BMI Paper

Skill-based Routing Policies in Call Centers

by

Martijn Onderwater

January 2010

BMI Paper

Skill-based Routing Policies in Call Centers

Author:
Martijn Onderwater

Supervisor:
Drs. Dennis Roubos

January 2010

VU University Amsterdam
Faculty of Sciences
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands

Preface

This paper is part of the curriculum of the master program *Business Mathematics and Informatics* at the VU University in Amsterdam. Students are expected to select a subject of their own interest and write a formal paper on that subject. The paper should contain both practical and theoretical aspects, and should at least deal with two of the three BMI areas: Business, Mathematics and Informatics.

I would like to thank my supervisor, Drs. Dennis Roubos, for his continuous support, endless patience and creative ideas. His guidance was essential for finishing this paper.

Abstract

Modern call centers deal with a wide variety of service requests from customers. This is partly due to the increasingly divers range of products that companies have, but also because of the many ways that a call center can be contacted (telephone, email, IM, etc.). In the past, agents were trained to deal with all possible types of calls that could arrive at the call center. The large number of call types arriving at modern call centers makes it practically impossible to fully train each agent. A typical agent only has the skills to deal with a limited number of call types.

This has caused a need for methods that describe which call should be serviced by which agent. These methods should also take several service level constraints into account, e.g., prioritizing premium customers, minimizing waiting times of customers and minimizing idle times of agents. The term *Skill-based routing* (SBR) is commonly used for such methods.

Modern SBR software packages use heuristic methods that can be configured by managers. In this paper, we will experiment with some of these heuristics. We will also attempt to formulate and apply a mathematical framework that will allow us to improve a heuristic policy. It is our aim to compare the mathematical approach to the heuristic methods, show the improvement and investigate its usefulness in practice.

In this paper we will show that the mathematical approach has some potential. For a small call center example, we describe how we came very close to improving a heuristic policy, although time limitations stopped us from actually achieving it. At the same time, we discuss some practical aspects that will arise if the mathematical approach is ever applied in a real world call center. Taking all these aspects into consideration, we conclude the mathematical approach has potential, but is a long way from maturity.

Contents

Chapter 1

Introduction

Classical call centers were large rooms where service representatives (called 'agents') wearing telephone headsets would sit in front of computer screens. Customers called these call centers with, e.g, questions about products or complaints. Service was focused on keeping the agents busy, thereby trying to keep waiting times as short as possible for customers. The calls that were handled by the call center were usually of similar type, making agents' work very repetitive.

The situation today is quite different. The variety of calls that arrive at a call center is much wider. This is partly due to the growing number of services offered by companies, but also because technological advances have made it possible to contact a call center not just via the phone, but also via fax, email, text-messages, forums and instant messaging. Managers have realized that their call centers are not just there to handle complaints, but are key to increasing customer satisfaction. The classical call center has evolved into the modern Customer Contact Center. There have even emerged companies that specialize in running a customer contact center, allowing other companies to outsource their in house contact center.

The classical call center used a Automatic Call Distributor (ACD) to assign a call to an agent, as soon as it became available. In the modern customer contact center, an agent can not take just any call, because he/she might not have the necessary skills. Therefore, a new system is needed that can find out what skills are needed for a particular call and which agents can handle this type of call. With this information, the system can route the call to the correct agent. This is commonly known as *Skill-based Routing* (SBR).

Modern SBR systems combine various sources of information to identify the type of call. Examples are: the telephone number that the customer

calls from, the number which the customer has dialed, existing customer records and an Interactive Voice Response menu. Based on this information, the SBR system may, e.g., decide that the current call has priority over other calls, because it is a premium customer. Or perhaps the call needs to handled by a German speaking agent, an agent with IT skills, a sales person or somebody with access to financial systems.

Some advantages of SBR are (from the product sheet of Nortel's [1] Symposium software, Nortel (2003)):

- Call resolution improves, because calls are directed to the most appropriately skilled agent the first time.
- Overall call processing time is faster, hand-offs and wait time are minimized, cost per call is reduced, and the customer receives more knowledgeable and efficient service.
- Shortening call durations increases the number of calls agents can handle, which in turn enables the contact center to grow in call volume without hiring more people and to reduce the number of calls abandoned by frustrated callers on hold.
- The ability to route and prioritize calls based on required skill set reduces agent training time and saves on training costs.
- Recognizing agents as individuals with unique skills increases agent morale and sense of personal investment, which is often reflected in lower turnover and enhanced job satisfaction and service extended to callers.
- The ability to understand and leverage the unique skills of agents gives managers the tools to offer incentives to agents based on their willingness to learn new skills and expand the scope of the calls they are able to effectively handle.

The same product sheet also claims

- In real world implementations, skill-based routing has been shown to increase agent productivity by more than 25 percent.
- One utility customer reported 98 percent first-call resolution and expedited agent training after implementing skill-based routing.

[1] On January 14, 2009, Nortel filed for Chapter 11 protection as a result of the financial crisis. In June 2009 it announced that it would stop operations and sell off all of its business units. Avaya bought Nortel's Enterprise Solutions business unit in July 2009, which includes the Symposium Call Center Server. Integration of both companies is still underway.

- A customer combined skill-based routing with specialized 800 numbers to reduce hold times by 89 percent and wait times by 75 percent, while at the same doubling the volume of calls handled by their agents.
- A major Las Vegas hotel and casino reported that abandoned calls dropped from 20+ percent to 4 percent when skill-based routing was introduced, due to reduced hold times and direct connection to the agents best able to satisfy callers' requirements.
- One customer was able to handle a 30 percent increase in call volume with zero staff growth.

So the advantages of and need for SBR systems is apparent, but its development is still in its infancy. In this paper we will start by looking at some heuristic ideas that are in use in todays SBR systems (chapter 2). In chapter 3 we introduce *Markov Decision Processes* as a mathematical framework. Chapter 4 deals with some practical issues concerned with this framework and shows how these can be addressed. Computer experiments with the discussed framework are in chapter 5. The paper finishes with a discussion of the contents and results of this paper and some ideas for further research in chapter 6.

Chapter 2

Heuristic skill-based routing

Figure 2.1 below shows some examples of a small call center (from Garnet & Mandelbaum (2000), where they are named "I", "V", "N", "X", "W" and "M" designs). The open rectangles on top are the queues for the specified call types and the circles are groups of agents. So in, e.g., the "M" design there are 2 types of calls and 3 groups of agents. Group 1 can handle calls of type 1, group 3 can handle calls of type 2 and group 2 can handle both call types.

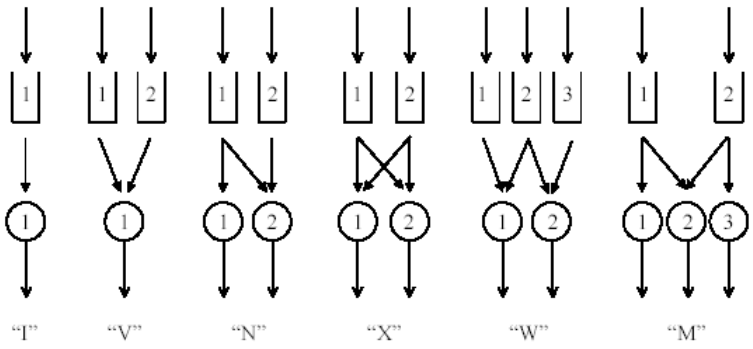


Figure 2.1: The five canonical designs.

The "I" and "V" designs are well known from queueing theory and are discussed extensively in, e.g., Koole (2008) and Bhat (2008). They have no need for skill-based routing. The other four designs all use some form of skill-based routing, although the routing policy is not specified. In, e.g., the "N" design, the routing policy could be chosen as:

- Let group 2 handle type 1 calls only if there are more than a certain number of type 1 call waiting in the queue. This policy protects the type 1 calls, perhaps because these customers pay more for their service than type 2 customers.

- Let group 2 handle type 1 calls only if there are no type 2 calls waiting. This would minimize idle-time among the group 2 agents.
- Let group 2 always handle type 1 calls, and switch to type 2 calls if there are no more calls of type 1. So type 1 customers are given absolute priority, at the expense of the type 2 customers.

There are of course other possible choices for the routing policy. Note that these policies are perfectly understandable for a call center with only a limited number of call types and groups, but they do not scale directly to larger situations. That is why modern SBR systems use heuristic routing policies that can be formulated in a more general way. They fall into two categories: the *Call Selection Policy* and the *Agent Selection Policy*. The call selection policy specifies which type of call an agent should take when he becomes available and the agent selection policy determines which agent group (if any) should handle an arriving call of a certain type. An agent selection policy and a call selection policy together from "the" routing policy. In the next few sections, we will discuss some of these heuristic policies.

2.1 Call selection

Fixed Priority Routing (FP)
FP assigns a fixed priority to each call type. When an agent becomes available, the call with the highest priority is chosen. Usually, the priorities correspond to numbers $1, 2, 3, \ldots$, where 1 is the highest priority. FP is sometimes also called Head-of-the-Line, or HOL. This policy is studied in various situations in Kleinrock (1975) and Koole (2008).

Shortest Remaining Processing Time (SRPT)
As the name suggests, SRPT selects the call with the shortest service time. According to Schrage & Miller (1966), this policy minimizes the average waiting time. It does mean that calls with a long service time are punished in favor of the shorter calls, and this situation is not always desirable. See Bansal & Harchol-Balter (2001) for more details.

Longest Queue (LQ)
This policy selects the first call from the longest queue that can be served by the agent that becomes available.

Time Function Scheduling (TFS)
TFS offers a more general framework for routing policies. Costs are associated with the amount of time that the first call in the queue of a certain call type has been in the system. The call with the lowest costs is served next. An example of this is the so-called *Generalized* $c\mu$ (or $Gc\mu$) rule. Delay costs

are quantified in terms of (convex, increasing) functions $C_i(t)$, where $C_i(t)$ is the costs incurred by a type i call that spends t units of time in the system.

Let μ_{ij} be the reciprocal of the average service time of call type i by agent group j, with $\mu_{ij} = 0$ if agent group j does not have the skills to serve a call of type i. The $Gc\mu$ rule then states that when an agent in group j becomes available, the first call in the queue of type i^* should be served, where i^* is calculated from

$$i^* = \underset{i}{\operatorname{argmax}}\, C_i'(W_i(t))\mu_{ij}.$$

Here, $W_i(t)$ is the waiting time of the first call of type i in the queue at time t and C_i' is the derivative of C_i. See Gans, Koole & Mandelbaum (2003) for more information on the $Gc\mu$ rule. Other applications of TFS can be found in Koole & Pot (2006).

Credit rule (CR)
This rule was described in a previous BMI paper by Marengo (2004). The idea of this rule is to select calls based on whether they are getting better or worse service than if each call type would be served by its own private queue. To do this, each call type k is assigned an amount of credit $U_k(t)$ using

$$U_k(t) = \frac{C_k - N_k(t)}{C_k - a_k}. \tag{2.1}$$

Here:

- $N_k(t)$ is the number of type k calls that are in the system at time t.
- a_k is the offered load of the queue corresponding to type k calls.
- C_k is the minimum number of agents with skill k (i.e., who can handle calls of type k) that would be needed if each call type had its own private queue. This can be calculated using the standard $M/M/c$ model (Erlang C). For this model, it is known that

  $$\mathbb{P}(W_q > t) = C(c,a)e^{-(c\mu-\lambda)t},$$

  with

  $$C(c,a) = \sum_{j=c}^{\infty} \pi(c) = \frac{a^c}{(c-1)!(c-a)}\pi(0)^{-1}.$$

  Here, λ is the parameter of the arrival (Poisson) arrival process, μ the reciprocal of the expected service time, $a = \lambda/\mu$ (the offered load) and

$$\pi(0)^{-1} = \sum_{j=0}^{c-1} \frac{a^j}{j!} + \frac{a^c}{(c-1)!(c-a)}.$$

So if we have a service level specified as $\mathbb{P}(W_q > t) < \alpha$, then we should take for C_k the smallest c such that

$$C(c, a)e^{-(c\mu-\lambda)t} < \alpha.$$

From queueing theory (see, e.g., Koole (2008)) we know that if the load $\rho_k = a_k/C_k < 1$, the Erlang C system is stable. Since we choose C_k large enough for this, we have $C_k > a_k$ and thus the denominator in equation (2.1) is always positive.

So if the numerator in equation (2.1) is positive, then $N_k(t) < C_k$ and thus there are still agents available to handle a type k call. Similarly, if $U_k(t) < 0$, then there is a shortage of type k agents. This leads to the following call selection rule: if an agent becomes available, select the call (from the set of calls for which he has the necessary skills) with the highest amount of credit.

Note that, since this call selection rule calculates C_k from the $M/M/c$ queue, it assumes a homogeneous Poisson arrival process and exponential service times. If either one of these assumptions do not hold in a call center, one should (if possible) reformulate this heuristic before applying it.

There is a similar rule for agent selection. We discuss this in the next section, where it is called *Value rule*.

2.2 Agent selection

Least Busy (LB)
This policy selects the agent group which is the least busy, i.e., the agent group for which the number of occupied agents divided by the total number of agents in that group is the smallest.

Hierarchical routing (HR)
Hierarchical Routing assigns priorities to agent groups, per call type. This is usually defined using a matrix, where each call type is a column and each agent group is a row. For example,

$$\pi = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \\ 2 & 0 & 1 \end{bmatrix}.$$

So agents from group 2 handle type 2 calls with priority 1, type 1 calls with priority 2 and type 3 calls with priority 3. Similarly, call type 2 is handled by agents from group 2 (priority 1) or group 1 (priority 2), but not by group 3 (denoted by the priority 0). See Koole & Pot (2006) for a description of the method.

Overflow routing (OR)
OR is an extension of HR, with the additional assumption that there exists an ordering of the priorities of the groups such that $\pi_{rj} > \pi_{sj}$ for $r > s$. For example,

$$\pi = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \\ 2 & 3 & 2 \end{bmatrix}.$$

Note that the values in the columns are increasing, showing the ordering of the priorities. Again, see Koole & Pot (2006) for more details.

Value rule (VR)
This rule (also from Marengo (2004)) approaches the agent selection problem similar to how the Credit Rule deals with the call selection problem. We start with some definitions:

- As before, C_k is the minimum number of agents with skill k that would be needed if each call type had its own private queue.
- $B_i(t)$ is the number of agents in group i that is available at time t.
- $A_{ik} = 1$ if agents from group i have skill k (and can thus answer calls of type k). $A_{ik} = 0$ otherwise.
- There are n call types and M groups.

Using these definitions, we see that $\sum_{i=1}^{M} A_{ik}B_i(t)$ is the total number of agents with skill k available at time t. Now define $Q_k(t)$ as

$$Q_k(t) = \frac{C_k}{\sum_{i=1}^{M} A_{ik}B_i(t)}.$$

So a high $Q_k(t)$ value means that there are few agents with skill k available. Each agent group i is now assigned a value $P_i(t)$ via

$$P_i(t) = \sum_{k=1}^{n} A_{ik}Q_k(t).$$

So a high P_i value means that agents from group i are busy. The agent selection rule now becomes: if a call of type k arrives, select an agent from group i that has the lowest P_i value. If there are no agents available with skill k, then the call is put in the appropriate queue.

Here also, it is worth emphasizing that C_k is calculated using an Erlang C model. So it assumes that customers arrive according to a homogeneous Poisson process and that the servers have exponential service times.

2.3 Beyond heuristics

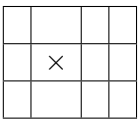
Besides heuristics, it is also possible to formulate a mathematical model for a call center and (in theory) to obtain a policy from that. In the next section we will discuss *Markov Decision Processes* as a mathematical framework for call centers.

Chapter 3

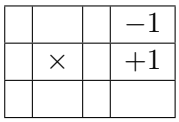
Markov Decision Processes

3.1 An example

We will explain the theory of Markov Decision Processes (MDP) using an example from Russel & Norvig (2002), although we change it slightly to suit our wishes. Consider a robot that lives in the following 3×4 world



The robot can move freely in this world and enter each of the squares (which are called 'states'), except for the state with the $\times$ in it. For convenience each of the states is referred to using its 'coördinates', i.e. the top left corner is $(1, 1)$ and the bottom right corner is $(3, 4)$. The goal of the robot is to reach the state $(1, 4)$ and to avoid $(2, 4)$. If the robot enters $(1, 4)$ it will receive a reward of 1, but if it enters $(2, 4)$ it will 'pay' 1. Most real world problems deal with costs, so we use the same terminology here. Hence, the reward of 1 is replaced by a costs of -1. With this, the world looks a bit like



The robot can move in either of the 4 directions of the wind: N, E, S and W (we will call these 'actions'). If the given direction happens to mean that a wall or the forbidden state $(2, 2)$ is hit, then the robot will return to its original position. Finally, each move will costs 0.02 (for, e.g., usage of power or fuel).

If this would be the entire model, then reaching $(1, 4)$ and avoiding $(2, 4)$ is easy. Suppose that the robot starts in $(3, 1)$, then it can end up in $(1, 4)$ by doing 2 moves N and 3 moves E, after which it will pay of $-1 + 5 \cdot 0.02 = -0.9$ (so it will actually receive 0.9). Similar solutions hold for the other states.

Unfortunately, in reality if the robot moves N in $(3, 3)$ its wheels might spin somewhat and cause the robot to end up in $(3, 4)$ instead of $(2, 3)$. To model this, let the robot go toward its intended position with probability 0.8, and let it veer to the left or right with probability 0.1. Figure 3.1 illustrates this for the case that the action is E.

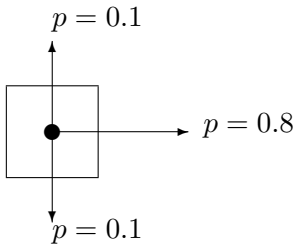


Figure 3.1: Probabilistic movement of the robot when the action is E.

By adding these deviations to the model, a probabilistic element is introduced into the formerly deterministic model. As a consequence it is no longer guaranteed that the robot will reach $(1, 4)$. It would be interesting to know which action should be taken in which state, in order to get the best outcome. And how do we measure the 'goodness' of an outcome? These questions will be addressed in the next few sections. But before we answer them, we will need some notation and background information on MDPs.

3.2 Definitions and notations

Discrete vs. continuous time
The model described in the previous section is an example of a discrete time MDP. The robot does not move continuously, but only at specific moments in time. In this paper we limit our attention to discrete time MDPs, because that is all that is needed to model the call-center problem of chapter 5. But continuous time MDPs are often used in practice, see the references in section 3.8.

Decision epoch
Each point in time where the system moves from one state into another is

called a decision epoch. Time will be denoted by t.

States
A state $s \in \mathcal{S}$ describes the dynamics of the system at the current decision epoch. For our robot, the state would be its location in the 3×4 world. The set of all states $\mathcal{S}$ may be either finite or infinite.

Actions
When the system reaches a certain state s, a decision has to be made as to which action to choose. A particular action is denoted by a and the set of all actions that can be chosen in state s by $\mathcal{A}_s$. Note that in general $\mathcal{A}_s$ depends on the current state s. Our robot model has $\mathcal{A}_s = \{N, E, S, W\}$ which is independent of s. But if it would, for example, be forbidden for the robot to hit the outer walls, then this dependence would also be present.

Transition Probabilities
When choosing an action $a \in \mathcal{A}_s$ the system transitions from state $s \in \mathcal{S}$ to some state $j \in \mathcal{S}$. Because of the probabilistic nature of the problems at hand, it is uncertain which state j this is. We do assume that we know (or can find) the probability of transitioning from state s to j when action a is taken: $p(j|s, a)$. As usual these probabilities satisfy

$$\sum_{j \in \mathcal{S}} p(j|s, a) = 1.$$

As an example, suppose that our robot is in state $s = (1, 1)$ and that we choose action $a = E$. There are now 3 possibilities for the next state: it can be

- $(2, 1)$ with $p(j = (2, 1)|s = (1, 1), a = E) = 0.1$,
- $(1, 1)$ with $p(j = (1, 1)|s = (1, 1), a = E) = 0.1$ (it bounces of the northern wall),
- $(1, 2)$ with $p(j = (1, 2)|s = (1, 1), a = E) = 0.8$.

Note that these probabilities do indeed sum to 1.

Costs
As a result of choosing the action a and thus moving from state s to state j, some costs are paid. We denote these costs by $c(s, a, j)$. Later on, we will see what the costs look like for our robot example.

Policies
A policy describes which action to take in which state. We use the notation π for the policy, so that $\pi(s)$ gives the action to be taken in state s. We will see an example of a policy in the next section, where we continue with the robot example.

Comparing policies
If we want to compare policies, then we need some kind of quality measure of a policy. There are three popular choices for this measure. The first one is called the *total expected costs criterion*, which basically sums all costs that are accrued along the way:

$$\sum_{t=0}^{\infty} \mathbb{E}c(s_t, \pi(s_t)). \tag{3.1}$$

Here, s_0 is the state that the system starts in. The expectation can be calculated using the transition probabilities

$$\mathbb{E}c(s_t, a) = \sum_{j \in \mathcal{S}} p(j|s_t, a) \cdot c(s_t, a, j).$$

Note that the sum in equation (3.1) is an infinite sum, so there might be problems with convergence. This criterion is mostly used in problems that have a clear terminal state, such as our robot example. If the terminal state is reached with probability 1 and no more costs are accrued afterward, then there are no issues with convergence of the infinite sum.

A second option is to use the *discounted total expected costs criterion*, which solves the convergence problem by discounting all costs to the current time. If we denote the discount factor by γ ($0 < \gamma < 1$), then the discounted total expected costs criterion leads to

$$\sum_{t=0}^{\infty} \gamma^t \mathbb{E}c(s_t, \pi(s_t)).$$

Note that if the expectation is bounded, then so is the infinite sum. This criterion is often taken in financial situations, where the time value of money is important. The third choice is the *average expected costs* criterion. It calculates a time average of the expected costs using

$$\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} \mathbb{E}c(s_t, \pi(s_t)).$$

Which criterion to choose, depends on the situation that the MDP is applied to. For our robot example, the total expected costs criterion is probably the most natural one to choose. But in this paper, we want to model a call

center, and for this situation the average expected costs criterion is a more appropriate choice. So we will use that criterion throughout this paper.

3.3 Policy evaluation

In the previous section, we chose to use the average expected costs criterion to measure the quality of a policy π. Define these costs as g^π, then

$$g^\pi := \lim_{T\to\infty} \frac{1}{T} \sum_{t=0}^{T} \mathbb{E}c(s_t, \pi(s_t)).$$

In practice g^π is found from the so-called *Poisson equations* (see chapter 7 of Bertsekas & Tsitsiklis (1996))

$$g^\pi + V^\pi(s) = \sum_{j\in\mathcal{S}} p(j|s,\pi(s)) \left[c(s,\pi(s),j) + V^\pi(j)\right] \qquad \forall s \in \mathcal{S}. \tag{3.2}$$

The $V^\pi(s)$ is called the *relative value function* and can be interpreted as the asymptotic difference in total costs that results from starting the process in state s instead of some reference state. We take $s = 0$ as the reference state and set $V^\pi(0) = 0$.

The Poisson equations are usually solved by iteratively updating $V_n^\pi(\cdot)$ from known values of $V_{n-1}^\pi(\cdot)$:

$$V_n^\pi(s) = \sum_{j\in\mathcal{S}} p(j|s,\pi(s)) \left[c(s,\pi(s),j) + V_{n-1}^\pi(j)\right] \qquad \forall s \in \mathcal{S}.$$

These new values are then made relative to the reference state using

$$V_n^\pi(s) \leftarrow V_n^\pi(s) - V_n^\pi(0) \qquad \forall s \in \mathcal{S}.$$

The average expected costs g^π are now approximated by filling in the reference state and the current values $V_n^\pi(s)$ in the Poisson equations (3.2)

$$\begin{aligned} g_n^\pi &= -V_n^\pi(0) + \sum_{j\in\mathcal{S}} p(j|0,\pi(0)) \left[c(0,\pi(0),j) + V_n^\pi(j)\right] \\ &= \sum_{j\in\mathcal{S}} p(j|0,\pi(0)) \left[c(0,\pi(0),j) + V_n^\pi(j)\right]. \end{aligned} \tag{3.3}$$

This method is known as *Policy Evaluation*:

Algorithm 1: Policy Evaluation

1. Set $n = 0$ and $V_0^\pi(s) = 0 \quad \forall s \in \mathcal{S}$
2. $n \longleftarrow n + 1$
3. $V_n^\pi(s) = \sum_{j \in \mathcal{S}} p(j|s, \pi(s)) \left[c(s, \pi(s), j) + V_{n-1}^\pi(j) \right] \qquad \forall s \in \mathcal{S}$
4. $V_n^\pi(s) \leftarrow V_n^\pi(s) - V_n^\pi(0) \qquad \forall s \in \mathcal{S}$
5. $g_n^\pi = \sum_{j \in \mathcal{S}} p(j|0, \pi(0)) \left[c(0, \pi(0), j) + V_{n-1}^\pi(j) \right]$
6. Stop if g_n^π has converged, otherwise go to step 2

Convergence of this method is guaranteed, provided that the model and policy are such that there exists a state that is reached with probability one, for all starting states. Furthermore, the resulting value function and average expected costs are then independent of the starting state s_0. See chapter 7 of Bertsekas & Tsitsiklis (1996).

Note that the update of $V_n^\pi(s)$ in step 3 is calculated from the previous estimates $V_{n-1}^\pi(j)$. Even though for some of the $V^\pi(j)$ we already have a better estimate $V_n^\pi(j)$, we do not use this information. This is known as *synchronous* updating. If we do use this information, then the update rule becomes

$$V^\pi(s) = \sum_{j \in \mathcal{S}} p(j|s, \pi(s)) \left[c(s, \pi(s), j) + V^\pi(j) \right]$$

which is known as *asynchronous* updating. One of the advantages of asynchronous updates is that only one storage vector is needed for $V^\pi(s)$, which can be quite useful for large problems. Another advantage is that a parallel implementation is possible, where each parallel component updates some of the states.

We can apply policy evaluation to our robot example. Suppose that we have the following policy

$\pi(s) =$

$\rightarrow$	$\rightarrow$	$\rightarrow$	
$\downarrow$	$\times$	$\rightarrow$	
$\rightarrow$	$\rightarrow$	$\uparrow$	$\uparrow$

So if we are in, e.g., $(3, 4)$, then the policy tells us to go N. Note that this is not a very good policy, since moving N from $(3, 4)$ brings the robot to the

square with 1 costs. For each state s there are costs of $c(s, \pi(s), j) = 0.02$ and the transition probabilities $p(j|s, \pi(s))$ are as described in section 3.2. We model two more properties of the two end states $(1, 4)$ and $(2, 4)$:

- When one of the end states is reached, the system will always remain in that end state (both states are called *absorbing*). For, e.g., the end state $(1, 4)$ this is denoted by $p(j = (1, 4)|s = (1, 4)) = 1$.
- When one of the end states is first reached, a one-off cost of either 1 or -1 is payed. All further transitions occur with no costs.

Appendix A shows Matlab code that uses (synchronous) policy evaluation to calculate g^π. The resulting value function is

$$V^\pi(s) = \begin{array}{|c|c|c|c|} \hline -0.5381 & -0.7494 & -0.7744 & -1 \\ \hline 0.9530 & \times & 0.8300 & 1 \\ \hline 0.9280 & 0.8998 & 0.8748 & 1.0083 \\ \hline \end{array}$$

The resulting average expected costs are $g^\pi = 0$. This makes sense since, in the long run, the robot will end up in one of the two end states. The first time it gets there, costs are paid. However, once the system is there, it remains there with no further costs. Hence, $g^\pi = 0$.

3.4 Value iteration

In the previous section we assumed that a policy was given and we saw a method to approximate the average expected costs. Usually we are more interested in what the best policy would be. One way to find such an optimal policy is to simply try all possible policies, i.e.,

$$\pi^* := \operatorname*{argmin}_\pi g^\pi.$$

In our robot example, this means that we have to evaluate at most $|\mathcal{A}|^{|\mathcal{S}|} = 4^{11} = 4194304$ policies. So even in a low-dimensional problem, the number of policies to evaluate is huge (exponentially large).

We tackle this problem using an iterative technique similar to the one used in the previous section to solve the Poisson equations (3.2). This time, we start with the *Bellman equations*

$$g^* + V^*(s) = \min_{a \in \mathcal{A}_s} \left\{ \sum_{j \in \mathcal{S}} p(j|s, a) \left[c(s, a, j) + V^*(j) \right] \right\} \qquad \forall s \in \mathcal{S} \qquad (3.4)$$

where g^* is defined by

$$g^* := \min_{\pi} g^{\pi}.$$

The Bellman equations (3.4) resemble the Poisson equations (3.2) used in policy evaluation, but the main difference is that the Bellman equations are no longer linear (because of the non-linear min operator). Fortunately, the iterative approach still works, resulting in $V_n^*(s)$ from which the g_n^* can be calculated. Once the g_n^* have converged, an optimal policy π^* can be found using

$$\pi^*(s) = \underset{a \in \mathcal{A}_s}{\operatorname{argmin}} \left\{ \sum_{j \in \mathcal{S}} p(j|s,a) \left[c(s,a,j) + V^*(j) \right] \right\} \qquad \forall s \in \mathcal{S}. \tag{3.5}$$

This is known as *Value Iteration*:

Algorithm 2: Value Iteration

1. Set $n = 0$ and $V_0^*(s) = 0 \quad \forall s \in \mathcal{S}$
2. $n \longleftarrow n + 1$
3. $V_n^*(s) = \min\limits_{a \in \mathcal{A}_s} \left\{ \sum\limits_{j \in \mathcal{S}} p(j|s,a) \left[c(s,a,j) + V_{n-1}^*(j) \right] \right\} \qquad \forall s \in \mathcal{S}$
4. $g_n^* = \sum\limits_{j \in \mathcal{S}} p(j|0,\pi(0)) \left[c(0,\pi(0),j) + V_{n-1}^*(j) \right]$
5. Stop if g_n^* has converged, otherwise go to step 2
6. Calculate an optimal policy from equation (3.5)

As with Policy Evaluation, updates can be done either synchronous or asynchronous. A proof of Value Iteration can be found in chapter 7 of Bertsekas & Tsitsiklis (1996).

Appendix A contains Matlab code of Value Iteration applied to the robot example. The resulting value function is

$$V^*(s) = \begin{array}{|c|c|c|c|} \hline -0.8994 & -0.9276 & -0.9526 & -1 \\ \hline -0.8744 & \times & -0.7731 & 1 \\ \hline -0.8463 & -0.8213 & -0.7937 & -0.5929 \\ \hline \end{array}$$

with the optimal policy

$$\pi^*(s) = \begin{array}{|c|c|c|c|}\hline \rightarrow & \rightarrow & \rightarrow & \\ \hline \uparrow & \times & \leftarrow & \\ \hline \uparrow & \leftarrow & \leftarrow & \downarrow \\ \hline \end{array}$$

and average expected costs $g^* = 0$. Note that, with time average expected costs of 0, the robot has no incentive to move toward one of the end states quickly. Therefore, it takes the safest route possible, in order to avoid the state with cost 1. So in the $(3, 4)$ state, the robot will move south, since that is the only way that it can be sure to avoid $(2, 4)$ state. Similarly, the policy tells the robot to move west in the $(2, 3)$ state.

In Ng (2008), Value Iteration is applied to this example, but now with the total discounted expected costs criterion. To illustrate the impact of the criterion, we show below the resulting value function and optimal policy when using the total discounted expected costs criterion (with discount factor $\gamma = 0.99$):

$$V^*(s) = \begin{array}{|c|c|c|c|}\hline -0.86 & -0.90 & -0.93 & -1 \\ \hline -0.82 & \times & -0.69 & 1 \\ \hline -0.78 & -0.75 & -0.71 & -0.49 \\ \hline \end{array}$$

with optimal policy

$$\pi^*(s) = \begin{array}{|c|c|c|c|}\hline \rightarrow & \rightarrow & \rightarrow & \\ \hline \uparrow & \times & \uparrow & \\ \hline \uparrow & \leftarrow & \leftarrow & \leftarrow \\ \hline \end{array}$$

With the total discounted expected costs criterion, costs made in the first few steps have a relatively big impact on the total expected costs. Hence, in this case, the optimal policy takes a bit more risk in the states $(2, 3)$ and $(3, 4)$, trying to avoid costs in the beginning.

3.5 Policy iteration

Value Iteration approximates the optimal value function and then uses this to find an optimal policy. Policy iteration focuses more on the policy and uses the value function as a means to improve policies. The algorithm starts with a random policy and calculates its value using Policy Evaluation. This is called the *Policy evaluation step*. The value function resulting from the policy evaluation step can then be used to obtain an improved policy, using an approach similar to equation (3.5). This is the *Policy Improvement step*. The complete algorithm is:

Algorithm 3: Policy Iteration

1. Set $n = 0$ and choose an arbitrary policy π_0
2. $n \longleftarrow n + 1$
3. *Policy evaluation step*:
   Calculate $V^{\pi}_{n-1}(\cdot)$ using Policy Evaluation
4. *Policy improvement step*:
   Obtain a new policy π_n from

$$\pi_n(s) = \underset{a \in \mathcal{A}_s}{\operatorname{argmin}} \left\{ \sum_{j \in \mathcal{S}} p(j|s,a) \left[c(s,a,j) + V^{\pi}_{n-1}(j) \right] \right\} \qquad \forall s \in \mathcal{S}$$

5. Stop if $\pi_n = \pi_{n-1}$, otherwise go to step 2

This algorithm converges to an optimal policy, see chapter 7 of Bertsekas & Tsitsiklis (1996). Policy Iteration usually converges in just a few iterations, but each iteration involves a computationally expensive Policy Evaluation step.

If we apply Policy Iteration to the robot example (see appendix A for the source code), we get the same policy and value function as with Value Iteration:

$$\pi^*(s) = \begin{array}{|c|c|c|c|} \hline \rightarrow & \rightarrow & \rightarrow & \\ \hline \uparrow & \times & \leftarrow & \\ \hline \uparrow & \leftarrow & \leftarrow & \downarrow \\ \hline \end{array}$$

with the optimal value function

$$V^*(s) = \begin{array}{|c|c|c|c|} \hline -0.8994 & -0.9276 & -0.9526 & -1 \\ \hline -0.8744 & \times & -0.7731 & 1 \\ \hline -0.8463 & -0.8213 & -0.7937 & -0.5929 \\ \hline \end{array}$$

and average expected costs $g^* = 0$.

We used the Policy Evaluation method from section 3.3 in the algorithm above, but there are other methods to evaluate a policy. We will see one of them in the next section. This leads to a more general version of Policy Iteration, called *Generalized Policy Iteration*:

Algorithm 4: Generalized Policy Iteration

1. Set $n = 0$ and choose an arbitrary policy π_0
2. $n \longleftarrow n + 1$
3. *Policy evaluation step*:
   Calculate $V_{n-1}^{\pi}(\cdot)$ by evaluating π using 'some' algorithm
4. *Policy improvement step*:
   Obtain a new policy π_n from

$$\pi_n(s) = \underset{a \in \mathcal{A}_s}{\operatorname{argmin}} \left\{ \sum_{j \in \mathcal{S}} p(j|s,a) \left[c(s,a,j) + V_{n-1}^{\pi}(j) \right] \right\} \qquad \forall s \in \mathcal{S}$$

5. Stop if $\pi_n = \pi_{n-1}$, otherwise go to step 2

3.6 Temporal Differences

The methods of the previous sections all calculate an estimate of $V(s)$ by looking at the expected result of a particular action. This is illustrated in Figure 3.2 (adapted from slides belonging to Sutton & Barto (1998), where they discuss TD methods in the context of rewards instead of costs (hence the r_{t+1})). Temporal Difference (TD) methods take a different approach than the iterative methods discussed before. The idea is to simulate a number of sample paths, and investigate $V(s)$ along each sample path. At each transition in a sample path, an error term is calculated. This error is then propagated back to the previous states in the sample path. Figure 3.3 (also from Sutton & Barto (1998)) shows an example of one such sample path.

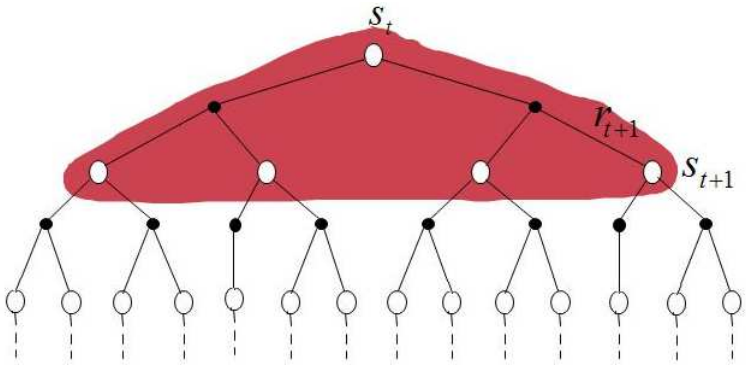


Figure 3.2: Approach of the methods from the previous sections.

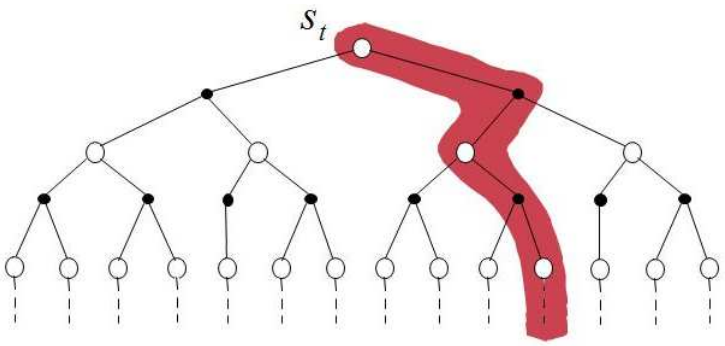


Figure 3.3: The approach of Temporal Difference methods.

We will show how this idea can be used to evaluate a given policy π. The algorithm is known in literature as *TD(λ)*, and is listed below. The parameter λ is a constant, with $\lambda \in [0, 1)$.

Algorithm 5: Policy Evaluation with TD(λ)

1. Initialize all the $V(s)$ arbitrarily
2. Generate a number of sample paths. Each sample path should start at randomly selected state and stop as soon as the reference state $s = 0$ is reached
3. Repeat for each sample path ω
   (a) Let $\omega = (s_1, s_1, \ldots, s_T)$
   (b) $e(s) := 0 \quad \forall s \in \mathcal{S}$
   (c) For $j = 1 \ldots T - 1$ do
      i. $d_j \longleftarrow c(s_j, \pi(s_j), s_{j+1}) + V(s_{j+1}) - V(s_j)$
      ii. $e(s_j) \longleftarrow e(s_j) + 1$
      iii. For all $s \in \{s_1, \ldots, s_j\}$
         $V(s) \longleftarrow V(s) + \alpha d_j e(s)$
         $e(s) \longleftarrow \lambda e(s)$

So the algorithm starts by initializing $V(s)$ and generating some sample paths using the given policy π. Then we calculate the temporal differences d_j, which represent the error between $c(s_j, \pi(s_j), s_{j+1}) + V(s_{j+1})$ and $V(s_j)$. In the remaining lines, this error is redistributed over previous states that were visited on the current sample path. For this, the *eligibility trace* $e(s)$ is used. This trace indicates what fraction of the current error d_j should be assigned to which state. At each transition from s_j to s_{j+1}, the state s_j is added to the eligibility trace via $e(s_j) \longleftarrow e(s_j) + 1$. Line 3(c)iii shows how

d_j is redistributed along the visited states in the sample path and how the eligibility trace is decayed by a factor λ. The parameter α is a step size and indicates how fast the d_j should be propagated. Here, we take α to be a constant, but in general it is also allowed to depend on j.

So with the parameter λ, we control how far the error d_j is redistributed among the previous states in the sample path. For $\lambda = 0$, the error d_j is assigned completely to s_j. As λ gets larger, more states become 'eligible' for a part of the error.

The TD(λ) algorithm for evaluating a policy can be plugged into Generalized Policy Iteration (Algorithm 4) to obtain a method for generating an optimal policy. There are also algorithms that avoid the use of Generalized Policy Iteration. Instead, they extend the idea behind TD(λ) and incorporate a method to find optimal policies while doing the simulations. See the references in section 3.8.

We will experiment with TD(λ) a lot in this paper, so we omit its application to the robot example here.

3.7 Terminology

Markov Decision Process is a term that is mostly used in the area of Operations Research. But the same subjects also arise in other fields, such as Control Theory (engineering and economics) and Reinforcement Learning (computer science). Hence the literature contains multiple names for the same techniques. For example, the Bellman equations are also known as the Hamiltonian equations, the Jacobian equations, the Hamilton-Jacobian equations or the Hamilton-Jacobian-Bellman equations. Another example is Value Iteration. In the field of Numerical Analysis the synchronous variant is known as Jacobi iteration, whilst the asynchronous version is know as Gauss-Seidel iteration. See, e.g., Burden & Faires (1997).

We will use the names from MDP, as this is the area that this paper is in. But the reader should be aware of the existence of various names, to avoid confusion and also to be able to read literature from the other fields. A nice overview is given in section 1.5 of Powell (2007).

3.8 Further reading

Other MDP algorithms
There are some variations on Policy Iteration. One of these is Multistage Lookahead Policy Iteration, which uses the m-step transition probabilities

and corresponding costs when calculating a new policy π_{n+1}. Using this lookahead may improve convergence, but as the lookahead gets larger the computational effort increases rapidly. See Bertsekas & Tsitsiklis (1996) for more information. Another variation is called Modified Policy Iteration, which tries to overcome the heavy computations involved in the Policy Evaluation step. It replaces this step by a limited number of Value Iteration steps, which obtains an estimate of the value function more quickly (but less accurate) than Policy Evaluation. See again Bertsekas & Tsitsiklis (1996) or Powell (2007) (where it is called Hybrid Value/Policy Iteration).

TD(λ) and variations
The TD(λ) algorithm that we described in section 3.6 is based on the *Backward view of TD(λ)*, from Sutton & Barto (1998). They also introduce the *Forward view of TD(λ)* and show that both methods are equivalent, although the backward view is more easily implemented. The same book also discusses TD(λ)-based methods that can be used to get an optimal policy, without the aid of Generalized Policy Iteration. These methods estimate 'state-action pair values' $Q(s, a)$ instead of 'state values' $V(s)$. Examples are *Sarsa(λ)*, Q-learning and R-learning. See Sutton & Barto (1998) for details, but be aware that their discussion focuses on MDPs with the discounted total expected costs criterion and that have a terminal state.

Continuous Time MDP
As discussed at the beginning of section 3.2 we limit our attention to Discrete Time MDPs. But there are some interesting applications of continuous time MDPs. In Bacon (2008) they are used for the control of traffic lights and in Martin (2003) for describing the dynamics of riding a bicycle. The classic Pendulum Problem can also be modeled using Continuous Time MDPs, see Lagoudakis & Parr (2003).

Partially Observable MDP
In the robot example, we assumed that the robot always knows in which state it is. So the environment is completely observable. If this is not the case, then the robot does not know what action $\pi(s)$ to take. This complicates the situation considerably. POMDPs are explained in Russel & Norvig (2002) in the context of the robot example. There is also a web page on POMDPs (see Cassandra (2003)) which contains papers, tutorials, problems from literature and sample code.

Constrained MDP
It is not always sufficient to minimize expected costs (or maximize expected reward), as we do in MDP. For instance in telecommunications, it is desirable to maximize throughput whilst keeping delays to a minimum. This leads to some additional constraints. And for, e.g., a delivery agent,

we need to maximize rewards obtained from making the deliveries, but also take into account some limitations for the total time spent en route. More on Constrained MDPs can be found in, e.g., Altman (1999).

3.9 Bibliographic notes

The iterative approach described in the previous sections is the central idea behind *Dynamic Programming*. This technique is often used in practice to solve a wide range of optimization problems. The term 'dynamic' refers to the decisions to be taken over time and the term 'programming' is a synonym for optimization. However, nowadays computers are often used to implement dynamic programming algorithms, so 'programming' is often thought of as 'computer programming'.

The foundations of the field of dynamic programming were laid by Richard Bellman in his text Bellman (1957). However, the paper by Shapley (1953) on stochastic games includes Value Iteration as a special case. In the 1960s, the basics for other computational methods (such as Policy Iteration) were developed in papers like Howard (1960), Manne (1960) and Blackwell (1962). The seventies continued with research on finite MDPs, resulting in books such as Derman (1970), Mine & Osaki (1970) and Ross (1970). Since then, thousands of papers have been published on MDPs and each year new books emerge.

A recent overall treatment of the field of MDP is given in Puterman (1994). The first two chapters of Bertsekas & Tsitsiklis (1996) also form a good introduction. Much of this chapter has been based on Powell (2007), which is very readable and also contains theory to be used in the following chapters. For students at the VU University Amsterdam, there is a course on MDP. The lectures notes are online (see Koole (2006)). There is also another BMI paper which explains the basics of MDP (as part of a text about the ideal racing line), see Beltman (2008).

Currently, MDPs are used to model and control large, complex, non-linear systems. Many of these applications use the variations of MDPs that were discussed in the previous section (CTMDPs, POMDPs, constrained MDPs). Solving these models is difficult and usually requires some sort of approximation method. These approximation methods are the subject of much recent research. We will discuss this in the next section.

Chapter 4

Approximate Dynamic Programming

In theory, we are now able to model a call center as an MDP, apply a dynamic programming algorithm (i.e., value iteration) and find a skill-based routing policy. But there are some practical limitations that arise when trying to do so:

1. Step 3 of Value Iteration tries, for each state s, all actions a in order to find the action that minimizes the expression between the braces. In practice, the state space S can be so large that looping over all states would take too much time on most modern computers. The problem is even worse, since the minimization has to be done in each iteration. This is what is commonly known in literature as *The curse of dimensionality.*

2. Step 3 of Value Iteration implies that there is some sort of array on the computer that can hold all of the $V(s)$. This is often impossible, because of the size of the state space.

We will discuss possible solutions to these issues and will also see some dynamic programming techniques that use these solutions. Because dynamic programming now no longer gives exact solutions, this field is referred to as *Approximate Dynamic Programming (ADP).*

4.1 Representative states

As mentioned, looping over all states in, e.g., Value Iteration is computationally not feasible. In practice, Value Iteration is not done over all states, but only over a representative subset. This *representative set of states* is denoted by $\widetilde{\mathcal{S}}$. Choosing the states that make up $\widetilde{\mathcal{S}}$ is usually done with some sort of prior knowledge about the practical system that

is being modeled. For instance, in section 5.1 we will experiment with the $M/M/c$ queue, where the state space consists of the number of persons in the system. From queueing theory, it is known that the system will be in the smaller states most of the time (see, e.g., Koole (2008)). Hence, if we take $\widetilde{\mathcal{S}} = \{0, 1, \ldots, 20\}$, then we concentrate value iteration on the smaller states. Of course, the approximation of the value function might be of very bad quality outside $\widetilde{\mathcal{S}}$. This needs to be kept in mind when interpreting or using the resulting value function.

Another approach is to determine the set of representative states via some form of random sampling. This sampling could be completely random, but then the resulting states need not be representative. Usually, the sampling is done with the transition probabilities. This could be done online (while Value Iteration is running) or offline (using, e.g., a simulation before running Value Iteration).

Sometimes, asynchronous Value Iteration can provide a solution. If the state space can be divided up into smaller parts and value iteration is feasible on each part, then Value Iteration can be performed asynchronously and in a distributed setting. This idea may also benefit from the increasing amounts of computing power that becomes available via providers such as Amazon and Google.

4.2 Approximating the value function

Storing the value function on a computer is not always possible, due to the fact that the size of the state space ($|\mathcal{S}|$) is too large. A solution to this problem is to approximate the value function $V(s)$ by a fixed structure that is characterized by a parameter vector r, with $|r| \ll |\mathcal{S}|$. Then only the parameter vector r needs to be stored. Such an approximation is usually denoted by $\widetilde{V}(s; r)$. The choice of the approximating structure can be based on, e.g., ease of computation or some prior knowledge of the real value function. We will discuss some possible choices below.

4.2.1 Linear combination of basis functions

This is a fairly general idea that approximates $V(s)$ by

$$\widetilde{V}(s; r) = \sum_{i=1}^{k} r_i \phi_i(s),$$

where $\phi_i(s)$ are the so-called basis functions. Or, as they are often called in the Artificial Intelligence community, *features*. Choosing the basis functions is somewhat of an art form and usually requires both experience and

substantial knowledge of the problem that is being modeled. An example of this can be found in Restrepo, Henderson & Topaloglu (2008), where the deployment of ambulances is considered. The basis functions there are created by, e.g., modeling missed calls and unreachable calls.

It is also possible to choose basis functions from a general class of functions. Examples of such classes are:

Polynomials
Suppose s is not a vector, e.g, $s \in \mathbb{N}$, then $\phi_i(s) = s^{i-1}$ is often chosen. Hence, $\widetilde{V}(s;r)$ is a polynomial of degree $k-1$. This choice is motivated by the fact that a function can be approximated by its k-th degree Taylor polynomial. The method can be easily extended to the situations with a higher-dimensional state space.

Piecewise constant functions
Here, the $\phi_i(s)$ are taken to be constant on a part of the state space, and 0 elsewhere. Besides the choice of the r_i, the partitioning of the state space needs to be decided upon. This technique is also known as *State Aggregation*, and was one of the first ideas to be applied to value function approximation.

Piecewise linear functions
In this case, the $\phi_i(s)$ are linear on the state space, but are allowed to change slope depending on the part of the state space.

Piecewise polynomials
The previous two choices for the basis functions are an example of piecewise polynomials (of degree 0 and 1, respectively). It is also possible to use polynomials of a higher degree. An example are the so-called *Splines*. With this technique, n-degree polynomials are constructed on each 'interval'. These polynomials are created in such a way that the first $(n-1)$ derivatives of the functions meeting in a 'split point of the interval' are equal. Hence, splines provide a smooth approximation to the value function. Note that, in general, this is not really necessary when using piecewise polynomials to approximate the value function. We are only interested in the value function at the split points, and have no need for the value function on the inside of the interval. So the approximation to the value function may just as well be discontinuous.

Sine functions
This choice is based on *Fourier theory*, which states that every function can be written as an infinite sum of sine functions.

Functions with small support
In general, it is advisable to take functions that are non-zero on only a small part of the state space (i.e., have small support). It ensures that when a component of the parameter vector r changes, it only effects the value function on a small part of the state space. This makes such a choice computationally efficient. Examples of functions with small support are *B-splines* (actually, all piecewise polynomials) and *Radial Basis Functions*. Polynomials and sine functions are typical examples of functions that do not have a small support.

4.2.2 Nonlinear combinations

It is also possible to look for an approximating structure to the value function that is not linear in its parameters. For example,

$$\widetilde{V}(s;r) = u \cdot e^{-v \cdot s} + w,$$

with $r = (u, v, w)$. In general, $\widetilde{V}(s;r)$ can consist of various nonlinear functions, such as trigonometric functions, max/min operators, logarithms and Gaussian type functions.

4.2.3 Neural Networks

A neural network is a technique that is inspired by the workings of the human brain. The elementary processing units of the brain are called *neurons*. Each neuron is connected to many other neurons and together they form a complex network. A tiny piece of this network can be seen in Figure 4.1, which is a drawing by Cajal (1904). The drawing shows some triangular shapes A, B, C, D, E (the cell-bodies of the neurons) and a lot of wires interconnecting the neurons. In reality, a region of one cubic millimeter contains over 10^4 cell bodies and kilometers of wire.

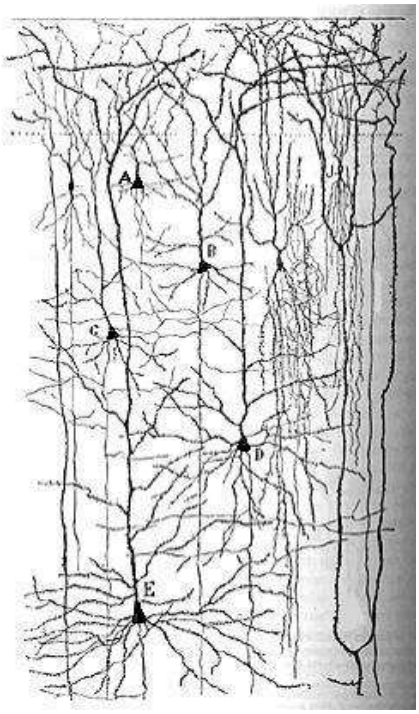


Figure 4.1: Drawing of a part of the human cortex.

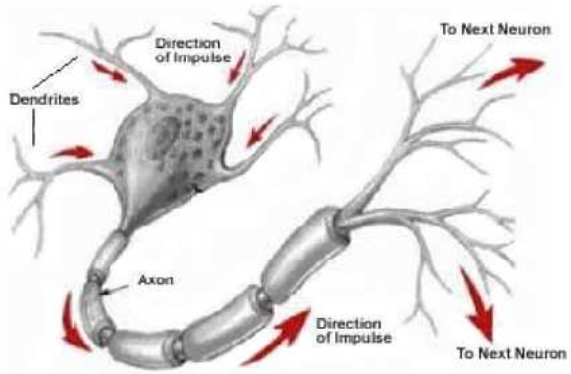


Figure 4.2: Schema of a neuron.

Figure 4.2 shows the schema of a neuron. A neuron consists of three parts: the dendrites, the soma (also called cell-body) and the axon. Roughly speaking, the dendrites are the input devices that deliver information from other neurons to the soma, using an electrochemical process. The soma is the 'central processing unit' which performs some processing step and then produces output. This output is then taken over by the axon and passed on to dendrites of other neurons.

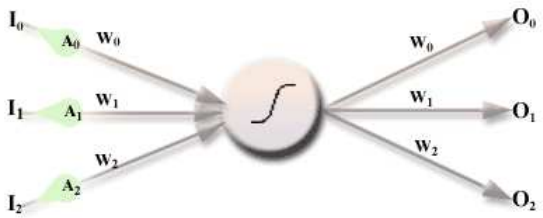


Figure 4.3: An artificial neuron.

An artificial neuron is shown in Figure 4.3. It has the same architecture as a real neuron: inputs, the neuron itself and some outputs. The processing is done using a function f, which takes the weighted sum of inputs and produces some outputs. The output of the node can be this weighted sum, i.e. $f(x) = x$. But there are other choices possible, e.g., *step function*, *sign function* and *sigmoid function*. See Figure 4.4.

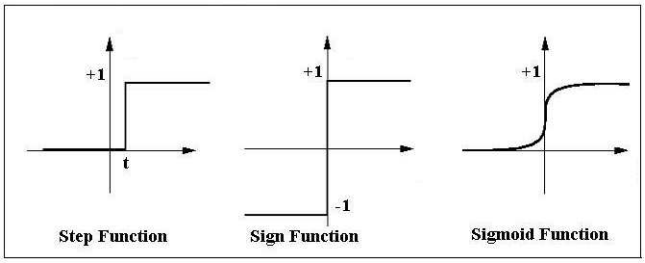


Figure 4.4: Some choices for the activation function.

When these neurons are combined, we get what is called a *Neural Network*. The neurons are organized in layers. The input neurons form the input layer, the output nodes the output layer. In between the input and output layer, multiple hidden layers can be created. An example of a neural network with one hidden layer can be seen in Figure 4.5

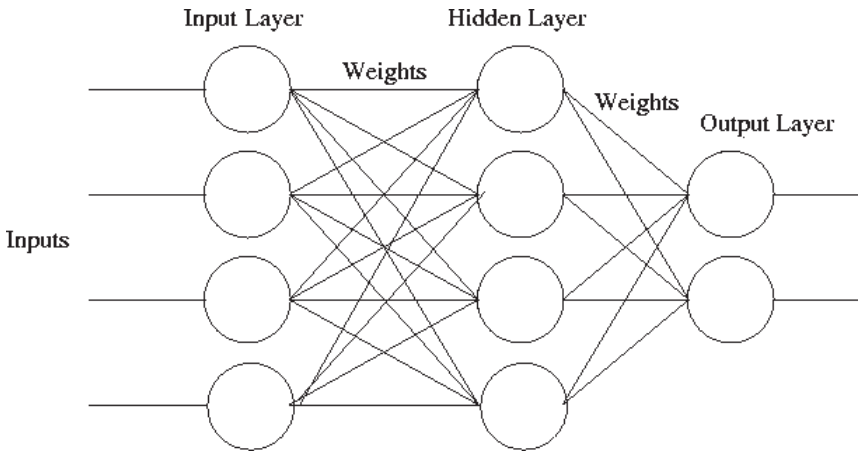


Figure 4.5: An example of a neural network.

Neural networks have been used for many purposes in the past, one of them being function approximation (see the references in section 4.4). In order for the network to approximate a given function, the weights need to be chosen such that the approximation is good. In practice, the weights are not chosen but learned by a training procedure. The idea is to give the neural network inputs for which we know the corresponding outputs (a training set). After the input has been processed by the network, the generated output can be compared to the desired output. The result of this comparison can then be used to update the weights in the network. This training procedure is summarized as follows:

Algorithm 6: Neural Network Training

1. Choose a network architecture and initialize all weights randomly
2. Take an input/output pair (I, O) from the training set
3. Enter I into the network, resulting in output $\tilde{O}$
4. Compare O and $\tilde{O}$ and update the weights in the network using a technique called *Backpropagation*
5. If there are examples left in the training set, go to 2. Otherwise, stop

The key to this algorithm is *Backpropagation*. We will not explain the details of this technique here, but it can be found in many textbooks on neural networks, e.g., Rojas (1996).

4.3 ADP Algorithms

4.3.1 Approximate Policy Evaluation

This algorithm is basically the same as regular Policy Evaluation, except for the use of the representative set of states $\widetilde{\mathcal{S}}$ and the updating of r instead of $V(s)$

Algorithm 7: Approximate Policy Evaluation

1. Set $n = 0$ and initialize r_0 randomly
2. $n \longleftarrow n + 1$
3. $\forall s \in \widetilde{\mathcal{S}}$:
$$\widehat{V}(s; r_{n-1}) = \left\{ \sum_{j \in \mathcal{S}} p(j|s, \pi(s)) \left[c(s, \pi(s), j) + \widetilde{V}(j; r_{n-1}) \right] \right\}$$
4. Update r_{n-1} to r_n using the approximating structure $\widetilde{V}(s; r_{n-1})$ and its new estimates $\widehat{V}(s; r_{n-1})$
5. $g_n^* = \sum_{j \in \mathcal{S}} p(j|0, \pi(0)) \left[c(0, \pi(0), j) + \widetilde{V}(j; r_n) \right]$
6. Stop if g_n^* has converged, otherwise go to step 2
7. Calculate a policy similar to equation (3.5)

The key to this algorithm is the update of r_{n-1} to r_n in step 4. The way that this is done depends on the approximating structure $\widetilde{V}(s; r_n)$. We will look at this in more detail for the various approximating structures that were introduced in the previous section.

Linear combination of basis functions
When $\widetilde{V}(s; r)$ is a linear combination of basis functions, then r_n is often calculated from

$$r_n = \underset{r}{\operatorname{argmin}} \sum_{s \in \widetilde{\mathcal{S}}} w_s \left(\widetilde{V}(s; r) - \widehat{V}(s; r_{n-1}) \right)^2. \tag{4.1}$$

This is a weighted least squares problem, which has been well studied in the past. It can be rewritten as

$$r_n = \left[A^T W A \right]^{-1} A^T W \cdot \widehat{V}(\cdot; r_{n-1}).$$

Here, the matrix A is given by

$$A = \begin{pmatrix} \phi_1(s_1) & \phi_2(s_1) & \dots & \phi_k(s_1) \\ \phi_1(s_2) & \phi_2(s_2) & \dots & \phi_k(s_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(s_R) & \phi_2(s_R) & \dots & \phi_k(s_R) \end{pmatrix}$$

with

$$\widetilde{\mathcal{S}} = \{s_1, s_2, \dots, s_R\}.$$

The weight matrix W is

$$W = \begin{pmatrix} w_1 & 0 & \dots & 0 \\ 0 & w_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & w_R \end{pmatrix}.$$

This method is guaranteed to result in the r that achieves the minimum.

Nonlinear combinations
When $\widetilde{V}(s;r)$ is a function that is nonlinear in the parameters r, then the minimization in equation (4.1) can still be used, although it is considerably more difficult. But nonlinear optimization has been studied extensively and there are several methods available that can do the minimization, although it is not guaranteed that they end up with a global minimum. It is typically a local minimum, but it at least provides some minimization. Examples of the available methods are *Bisection*, *Newton's Method*, *Quasi-Newton Methods*, *Gradient Descent Techniques* (also known as *Steepest Descent Techniques*) and *Nonlinear Conjugate Gradients*. See Burden & Faires (1997).

Choice of the weights
In practice, not all parts of the state space are equally important. The weights w_s in equation (4.1) are used to make sure that the resulting r_n provides a nice fit on an important part of the state space. For example, in Roubos & Bhulai (2009) ADP is applied to the $M/M/c$ queue. The state is defined as the number of customers in the system. From queueing theory, it is known that the system is more likely to be in the smaller states. Hence, the weights are taken as $w_s = \rho^s$, where $0 < \rho < 1$ is the load of the system.

Neural networks
When $\widetilde{V}(s;r)$ is a neural network, the parameters r are the weights on the various connections between the layers. Updating these weights is done in a training step, where the $\widehat{V}(\cdot; r_{n-1})$ serve as training set. The

resulting error in the right hand side of equation (4.1) is then spread over the various weights using *backpropagation*, which is essentially Gradient Descent applied to the neural network architecture (see Sutton & Barto (1998)). A detailed description of the backpropagation algorithm can be found in Rojas (1996).

4.3.2 Approximate Value Iteration

There is also a ADP variant of Value Iteration and (Generalized) Policy Iteration. They are extended in a similar way as we did for Approximate Policy Evaluation in the previous section.

Algorithm 8: Approximate Value Iteration

1. Set $n = 0$ and initialize r_0 randomly
2. $n \longleftarrow n + 1$
3. $\forall s \in \widetilde{\mathcal{S}}$:
$$\widehat{V}(s; r_{n-1}) = \min_{a \in \mathcal{A}_s} \left\{ \sum_{j \in \mathcal{S}} p(j|s,a) \left[c(s,a,j) + \widetilde{V}(j; r_{n-1}) \right] \right\}$$
4. Update r_{n-1} to r_n using the approximating structure $\widetilde{V}(s; r_{n-1})$ and its new estimates $\widehat{V}(s; r_{n-1})$
5. $g_n^* = \sum_{j \in \mathcal{S}} p(j|0, \pi(0)) \left[c(0, \pi(0), j) + \widetilde{V}(j; r_n) \right]$
6. Stop if g_n^* has converged, otherwise go to step 2
7. Calculate a policy similar to equation (3.5)

Algorithm 9: Approximate (Generalized) Policy Iteration

1. Set $n = 0$, initialize r_0 randomly and choose arbitrary policy π_0
2. $n \longleftarrow n + 1$
3. *Policy evaluation step*:
   Evaluate policy π_{n-1} on $\widetilde{\mathcal{S}}$, which gives $\widehat{V}(s; r_{n-1})$
4. Update r_{n-1} to r_n using the approximating structure $\widetilde{V}(s; r_{n-1})$ and its new estimates $\widehat{V}(s; r_{n-1})$
5. *Policy improvement step*:
   Obtain a new policy π_n from
   $$\pi_n(s) = \underset{a \in \mathcal{A}_s}{\operatorname{argmin}} \left\{ \sum_{j \in \mathcal{S}} p(j|s,a) \left[c(s,a,j) + \widetilde{V}(s; r_n) \right] \right\} \qquad \forall s \in \widetilde{\mathcal{S}}$$
6. Stop if $\pi_n = \pi_{n-1}$, otherwise go to step 2
7. Calculate g_n^* from
   $$g_n^* = \sum_{j \in \mathcal{S}} p(j|0, \pi(0)) \left[c(0, \pi(0), j) + \widetilde{V}(j; r_n) \right]$$
8. Obtain a policy similar to equation (3.5)

4.3.3 Bellman error

It is also possible to tune the parameter vector r in $\widetilde{V}(s; r)$ directly from the Bellman equations. To do this, we define the *Bellman error* as

$$D(s; r) = -g - \widetilde{V}(s; r) + \min_{a \in \mathcal{A}_s} \left\{ \sum_{j \in \mathcal{S}} p(j|s,a) \left[c(s,a,j) + \widetilde{V}(j; r) \right] \right\}, \qquad s \in \widetilde{\mathcal{S}}. \tag{4.2}$$

We are then interesting in finding r^* such that

$$r^* = \min_r \sum_{s \in \widetilde{\mathcal{S}}} \left[w_s D^2(s; r) \right],$$

where w_s are weights. Unfortunately, the g is unknown so we can not apply the above directly. This method could be combined with a short simulation run to estimate g, but then a 'reasonable' policy is needed to be able to run

the simulation. Another option is to start with a 'reasonable' policy and r and estimate g from

$$\tilde{g} = \sum_{j \in \mathcal{S}} p(j|0, \pi(0)) \left[c(0, \pi(0), j) + \widetilde{V}(s; r) \right]. \tag{4.3}$$

But finding a 'reasonable' policy and r is not very easy. Therefore, in practice, the simulation run is usually chosen as the desired method for estimating g.

So applying the Bellman error method is not very straightforward, because of the choice of g. But once g has been chosen, the minimization can be dealt with using the nonlinear optimization techniques discussed in section 4.3.1, e.g., gradient descent. For completeness, the entire algorithm:

Algorithm 10: Bellman Error Method

1. Choose a reasonable policy π
2. Approximate $\tilde{g}$ using, e.g., a short simulation run
3. Find an approximation to r^* from

   $$r^* = \min_r \sum_{s \in \widetilde{\mathcal{S}}} \left[w_s D^2(s; r) \right],$$

   which results in an approximation $\widetilde{V}(s; r^*)$ of the value function

A variation on this theme is to use a minimization that is not of the least squares type. For instance, one could use

$$r^* = \min_r \sum_{s \in \widetilde{\mathcal{S}}} \left| w_s D(s; r) \right|.$$

In the literature, algorithm 10 is usually used.

4.3.4 TD(λ), GPI and value function approximation

In section 3.6 we mentioned that the TD(λ) algorithm can be used in the Generalized Policy Iteration scheme (algorithm 4) to find an optimal policy. After doing so, this results in a estimate of the value function in all states that were visited on the various sample paths. These estimates can then be used to create an approximation of the value function, which, in turn, can be used for decision making in a real life system. In theory, this should work, but there are some practical considerations:

- **Number of sample paths in TD(λ)**
  The accuracy of the approximation of the value function in a particular state is determined by the amount of times that it is updated during all the sample paths. So using more sample paths gives higher accuracy, but also increased computation times.

- **Starting point of the sample paths**
  Each sample path should start at a random state and continue until it reaches the reference state. A sample path that is started 'far away' from the reference state will probably be longer than a sample path started 'closer' to the reference state. So the starting point has a direct influence on computation times and size of arrays. It also roughly determines which states will be visited often and thus affects accuracy of the TD(λ) method and thus also convergence of GPI.

- **Choice of λ in TD(λ)**
  A value for λ that is too small will not propagate errors far enough, and if it is too large the errors will propagate too far.

- **Choice of α in TD(λ)**
  The parameter α determines how quickly errors are propagated along the sample path. It can be a constant but ideally, this parameter should be large for inaccurate states and small for accurate states. That is why it is often related to the amount of times that a state is updated.

- **Convergence of GPI**
  GPI improves policies based on the value function that results from TD(λ). This works fine for states that were accurately approximated by TD(λ), but can not reasonably be expected to work in inaccurate states. So perhaps GPI should be stopped when two consecutive policies are equal in states that were visited 'often'. An alternative is to compute the average costs g_n^* and stop GPI when these values start increasing.

- **Which approximating structure to use**
  Do we take a neural network? It is known to be a universal approximator, so that makes it an attractive choice. But training it is quite slow. Or do we use some form of polynomial, hoping that the value function is of this form?

- **When to fit the approximating structure**
  If an approximating structure $\widetilde{V}(\cdot; r)$ is used, we need to determine when to fit it. Do we fit it after one sample path? Or after all sample paths? Or during a sample path? Or do we update a fit as soon as we decide that a state is accurate (i.e., when it has been updated a certain

number of times)? And do we use $\widetilde{V}(\cdot; r)$ in the TD(λ) algorithm, for instance to initialize the value function?

These choices are usually made based on experimental results and some domain knowledge. We will experiment with these issues in our examples in chapter 5.

4.4 Bibliographic notes

Research in ADP dates to the 1950s when "the curse of dimensionality" was first recognized in the operations research community (Bellman & Dreyfus (1959)). This "curse of dimensionality" was considered to be such a big problem, that the operations research community essentially left MDP for dead. But people from the field of artificial intelligence recognized the potential of MDPs for machine learning purposes and picked up the research on ADP. One of the earliest uses of ADP they came up with was training a computer to play a game of checkers (Samuel (1959)). The research that grew out of the artificial intelligence community is nicely summarized in the review by Kaelbling, Littman & Moore (1996) and the introductory textbook Sutton & Barto (1998).

As technical developments lead to increasingly powerful computers, people from the field of engineering control and the field of operations research picked up ADP again, realizing that the "curse of dimensionality" might be lifted. Extensive literature now exists within the engineering control community, see, e.g., White & Sofge (1992). At first they seemed to be unaware of the efforts that had already been put into ADP by the artificial intelligence community. This changed around the mid nineties, with, e.g., the publication of Bertsekas & Tsitsiklis (1996). They wrote from a control perspective, although the influences from operations research and artificial intelligence are apparent. More recently, papers were published that brought together the engineering control, artificial intelligence and operations research communities. See, e.g., the papers by Si, Barto, Powell & Wunsch (2004) and Tsitsiklis & Roy (1996).

Nowadays, ADP is applied to a wide variety of complex systems. These systems usually have large state spaces and severe nonlinearities. Some examples of such systems are (from Roy (2001)):

- Call Admission and Routing,
- Strategic Asset Allocation,
- Supply-Chain Management,

- Emissions Reduction,
- Semiconductor Wafer Fabrication.

Currently, much research is being done on how to approximate the value function. Chapters 7 and 11 of Powell (2007) discuss some methods, such as regression models, neural networks and (piecewise) linear approximations. In Deisenroth, Peters & Rasmussen (2008), Gaussian Processes are used as an approximating structure. Konidaris & Osentoski (2008) uses the Fourier Basis. Another recent idea is to iteratively update the basis functions (and not just the coëfficiënts), in order to better approximate the value functions. See, e.g., Mahadevan (2005).

Chapter 5

ADP in a call center

In this chapter we return to the situation of a call center and use it to experiment with techniques from this paper. We will look at three different call centers: we start with the classic $M/M/c$ queueing model, using it mostly to do some relatively easy experiments. Then we modify this example slightly to a special case of $M/M/2$ and show how an optimal policy can be obtained. The last example deals with a more general example of a call center.

5.1 An example: $M/M/c$

In this section, we will apply Approximate Policy Evaluation to the classic $M/M/c$ queueing model. This model has only one possible policy:

- **Agent selection**: arriving calls are either served by an agent (if one is available), or put in the queue.
- **Call selection**: when an agent becomes available, it serves the first customer in the queue (if one is available).

The $M/M/c$ model is not very realistic for a call center, since, e.g., it can handle only one type of call. But it can serve as an excellent model to illustrate the workings of the techniques from this paper, in particular because there exists a closed-form expression for the value function.

In order to be able to do policy evaluation, we need to specify the transition probabilities. But the $M/M/c$ queueing model is a continuous model, so this needs to be converted to a discrete time model. Also, we need to choose a costs function. Both issues are discussed below.

5.1.1 Uniformization

Uniformization is a technique developed by Jensen (1953) to convert continuous time Markov chains into discrete time Markov chains. It is

explained in various textbooks, e.g., Tijms (2003) and Puterman (1994). The explanation here is based on Koole (2006).

In continuous time Markov chains, the system leaves states at a rate that can vary per state. The idea behind uniformization is to add dummy self-transitions to each state. These are created in such a way that the transition rate out of a state becomes equal for all states. Hence, the expected amount of time that the system spends in a state is constant. With this property, the system can be viewed as a discrete event system.

Mathematically, this is formalized as follows. Suppose that $x \in \mathcal{S}$, $y \in \mathcal{S}$ and define the rate of transition from state x to state y in the continuous time Markov chain as $\lambda(x, y)$. Now choose a γ such that

$$\sum_y \lambda(x, y) \leq \gamma.$$

We construct the new process with transition rates $\lambda'(x, y)$ by setting

$$\lambda'(x, y) = \lambda(x, y) \qquad \forall x \neq y.$$

The dummy transition rates from x to itself are now taken such that that all rates out of x sum to γ:

$$\lambda'(x, x) = \gamma - \sum_{x \neq y} \lambda(x.y) \qquad \forall x \in \mathcal{S}.$$

This new process has expected transition time $1/\gamma$. The original transition rates are usually scale by γ, so that the $\lambda'(x, y)$ can be interpreted as the transition probabilities and a transition occurs each time step.

We will explain this technique for the $M/M/c$ queue. Here, arrivals occur according to a Poisson process and the service time distribution is exponential. We will denote the parameter of the arrival process by λ and the parameter of the exponential service time distribution by μ. The model is schematically depicted in Figure 5.1.

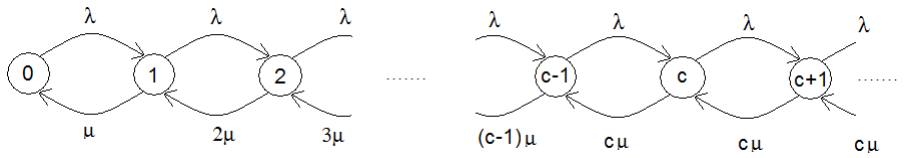


Figure 5.1: Schema of the $M/M/c$ birth-death process.

The first step of uniformization is to scale the transition rates by γ. From the schema it is clear that $\gamma = \lambda + c\mu$. Hence,

$$\lambda \leftarrow \frac{\lambda}{\gamma}, \qquad \mu \leftarrow \frac{\mu}{\gamma}.$$

When that is done, the dummy self-transitions are added:

$$\lambda'(x,x) = 1 - \lambda - \min(c,x)\mu \qquad x \in \mathcal{S} = \mathbb{N}_0.$$

In discrete time Markov chains it is more natural to talk about transition probabilities instead of transition rates. With the usual notation of $p(y|x)$ for the probability of a transition from state x to state y, the transition probabilities become

$$\begin{cases} p(x-1|x) &= \min(c,x)\mu \qquad x > 1 \\ p(x+1|x) &= \lambda \qquad x \geq 0 \\ p(x|x) &= 1 - \lambda - \min(c,x)\mu \qquad x \geq 0 \end{cases}$$

The update rule for Policy Evaluation becomes

$$\begin{aligned} \widehat{V}(n;s) &= c(s) + \lambda \widetilde{V}(s+1; r_{n-1}) \\ &\quad + \min(s,c)\mu \widetilde{V}([s-1]^+; r_{n-1}) \\ &\quad + (1 - \lambda - \min(s,c)\mu)\widetilde{V}(s; r_{n-1}). \end{aligned}$$

5.1.2 The costs function

Until now we have always dealt with the situation where costs were associated with transitions from one state to another. In the case of call centers, it is more natural to consider costs that depend on the amount of time that was spent in a certain state. For instance, with the $M/M/c$ queue, holding costs may be associated with the amount of time a customer spends in the queue. Suppose that these holding costs are h per time unit, then the costs function

$$c(s) = hs$$

reflects the costs of being in state s. Note the difference with the previous notation of the costs function: $c(s, \pi(s), j)$.

If we set $h = 1$, then the costs function $c(s) = s$ leads to a special interpretation of the expected average costs g. Costs correspond to queue lengths, so g is the average queue length (usually denote by $\mathbb{E}L_q$ in queueing theory). So minimizing the costs leads to a minimal average queue length, which is an important key performance indicator for a call center. This illustrates a crucial role of the costs function: it provides a way to control the process and steer solutions away from undesirable states. Therefore, the costs function

should be chosen with care, otherwise ADP techniques might come up with very unpractical results.

5.1.3 The real value function

For the $M/M/c$ queue there exists an analytic expression for the value function. Following Bhulai & Koole (2003), we use the number of customers in the system as state space, so $\mathcal{S} = \mathbb{N}_0$ (although there are other possibilities, see the section about *state disaggregation* in Roubos & Bhulai (2009)). The costs function is

$$c(s) = s.$$

For $0 \leq s \leq c$ the value function is then given by

$$V(s) = \frac{g}{\lambda} \sum_{i=1}^{s} F(i) - \frac{1}{\lambda} \sum_{i=1}^{s} (i-1) F(i-1), \tag{5.1}$$

where $F(i)$ is defined as

$$F(i) = \sum_{k=0}^{i-1} \frac{(x-1)!}{(x-k+1)!} \left(\frac{\mu}{\lambda}\right)^k$$

and the average expected costs g can be calculated from

$$g = \left[\frac{1}{\rho} F(c) + \frac{1}{1-\rho}\right]^{-1} \cdot \left[cF(c) + \frac{c\rho}{1-\rho} + \frac{\rho}{(1-\rho)^2}\right] \tag{5.2}$$

For $s \geq c+1$, $V(s)$ is given by

$$\begin{aligned} V(s) \quad = \quad & \tfrac{-(s-c)\rho}{1-\rho} \tfrac{g}{\lambda} + V(c) + \\ & \left(\tfrac{(s-c)(s-c+1)\rho}{2(1-\rho)} + \tfrac{(s-c)(\rho+c(1-\rho))\rho}{(1-\rho)^2} \right) \tfrac{1}{\lambda} \end{aligned} \tag{5.3}$$

5.1.4 Approximating the value function

We will start by confirming some of the results that were obtained in Roubos & Bhulai (2009). There, a second degree polynomial is used as an approximating structure for the value function. The set of representative states is taken to be $\widetilde{\mathcal{S}} = \{0, 1, \ldots, 20\}$. We use the same setup, but will try various approximating structures:

- **Second degree polynomial**. This is implemented in Matlab using a weighted version of the *polyfit* function. See Canós (2003). The weights are chosen as described in section 4.3.1, i.e., $w_s = \rho^s$.

- **Neural network with 1 hidden layer**. We used Netlab, a neural network package by Nabney & Bishop (2003).
- **Third degree (cubic) splines**. Matlab has some built-in functions for dealing with piecewise polynomials. We used *interp1* to create the splines and *ppval* to evaluate the results.

These structures were discussed in section 4.3.1. The results can be seen in Table 5.1 and the corresponding source code is in appendix B (after (un)commenting the correct lines).

λ	μ	c	g	$g(V^1)$	g_{pol}	g_{nn}	g_{spl}
4	2	8	2.00	2.00	1.9836	2.0353	1.9828
10	8	5	1.25	1.28	1.2444	1.2950	1.2472
8	2	16	4.00	4.01	3.9260	3.8492	3.9268
5	1	10	5.04	5.08	4.9328	4.9899	4.9628
3	2	3	1.74	1.70	1.6582	1.7506	1.7179
10	4	5	2.63	2.68	2.5369	2.4614	2.6039
15	5	4	4.53	4.37	4.2247	4.2846	4.2796
3	2	2	3.43	3.33	3.2068	3.4862	3.2130
9	3	4	4.53	4.36	4.2247	4.3106	4.2796

Table 5.1: Results for $M/M/c$ queue.

The first three columns show the parameters λ, μ and c of the $M/M/c$ model. The fourth column contains g, obtained by Roubos & Bhulai (2009) from equation (5.2). Column five ($g(V^1)$) lists the results that they found when using a second degree polynomial. The last three columns contain the approximations of g that resulted from the source code in appendix B. As can be seen, all structures find reasonable estimates of the correct g. The neural network and cubic spline outperform the polynomial, but it should be mentioned that training the neural network is a time consuming process. Fitting a spline or a polynomial is much faster.

5.1.5 Other stop criteria

In the previous section we stopped the simulations when two consecutive estimates of the average expected costs differed by less than 0.1%. Mathematically, this becomes:

$$\left| \frac{g_n - g_{n-1}}{g_{n-1}} \right| < 0.001. \tag{5.4}$$

In practice, this is not the best stop criterion to use. It might just be a coincidence that two consecutive approximations of g are close together. There are some other possibilities to choose from:

- In equation (5.4), the value 0.001 can be set to something smaller, e.g., 0.00001.
- Stop when equation (5.4) has occurred, say, k times consecutively.
- Stop when the maximum and minimum of the last k estimates are 'close' together.

These choices provide a bit more confidence that Value Iteration is stopped upon actual convergence. But since, in general, we do not have the exact value function, non of the stop criteria above guarantee that convergence has occurred. They are heuristic choices.

We experimented with these options on the three approximating structures in Table 5.1 and were able to improve most of the entries in the last three columns. For instance, the neural network for the parameters $\lambda = 10, \mu = 4$ and $c = 5$ results in $g_{nn} = 2.4614$. If we change the stopcriterion and demand equation (5.4) to occur 10 times, then this results in $g_{nn} = 2.6289$, which is a somewhat better approximation of the required 2.63.

5.1.6 Performance outside $\widetilde{\mathcal{S}}$

We chose the set of representative states as $\widetilde{\mathcal{S}} = \{0, 1, \ldots, 20\}$, based on the considerations in section 4.1. Since we fit the approximating structure of the value function to these states, we can not reasonably expect performance to be very good outside $\widetilde{\mathcal{S}}$. But it is still interesting to see what happens there. We investigate this with the $M/M/c$ queue with parameters $\lambda = 4, \mu = 2$ and $c = 8$. As can be seen from Table 5.1, all three methods give quite accurate results for g. Figure 5.2 shows a plot of the three approximating structures, each trained on $\widetilde{\mathcal{S}} = \{0, 1, \ldots, 20\}$, but plotted on $[0, 40]$.

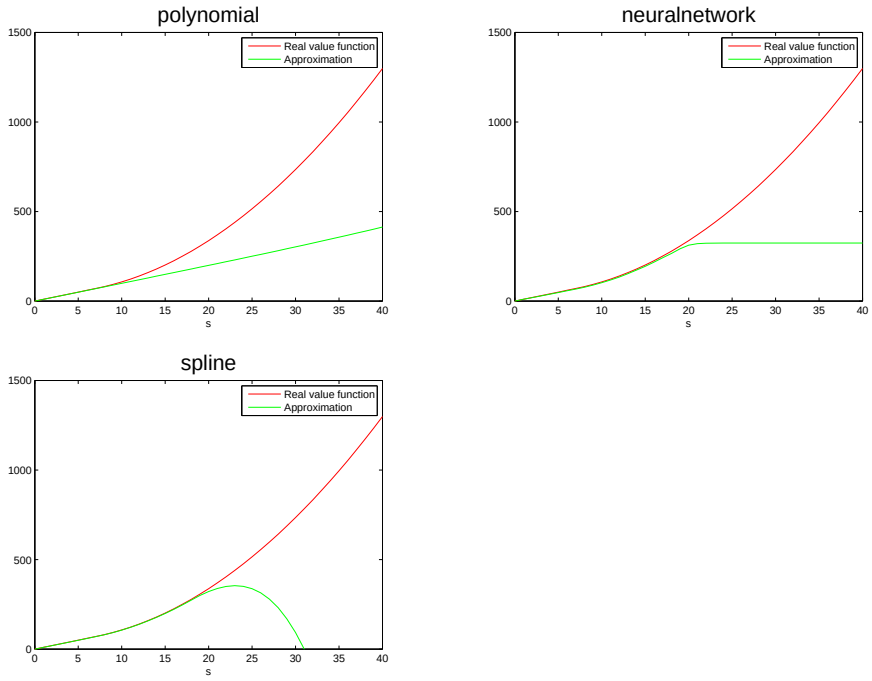


Figure 5.2: The three approximating structures on $[0, 40]$.

The polynomial produces a value function that looks like a straight line. This is to be expected, since the real value function is essentially a straight line for the first few states. And because of the weights, these states are the ones that the polynomial is fitted to. Both the neural network and the spline approximate the value function well on $\widetilde{\mathcal{S}}$, but fail outside the representative set of states. So, as expected, performance outside $\widetilde{\mathcal{S}}$ is not very good. Hence, Figure 5.2 emphasizes again the importance of choosing the states that are in $\widetilde{\mathcal{S}}$. They should be truly representative for $\mathcal{S}$, because if the value function is needed at a point outside $\widetilde{\mathcal{S}}$, then the approximating structure can not be trusted.

5.1.7 Changing $\widetilde{\mathcal{S}}$

Above, we used a single interval $[0, 20]$ to fit the approximations. It would be interesting to see what happens if we try to fit the value function on various disjunct parts of the state space. In the $M/M/c$ example, we could try to fit the value function on the intervals $[0, 10] \cup [20, 30] \cup [40, 50]$ and see how the approximations behave in between the intervals. It seems reasonable to expect that the approximating structure should be able to capture the true value function on the states in between the intervals. Figure 5.3 shows the results.

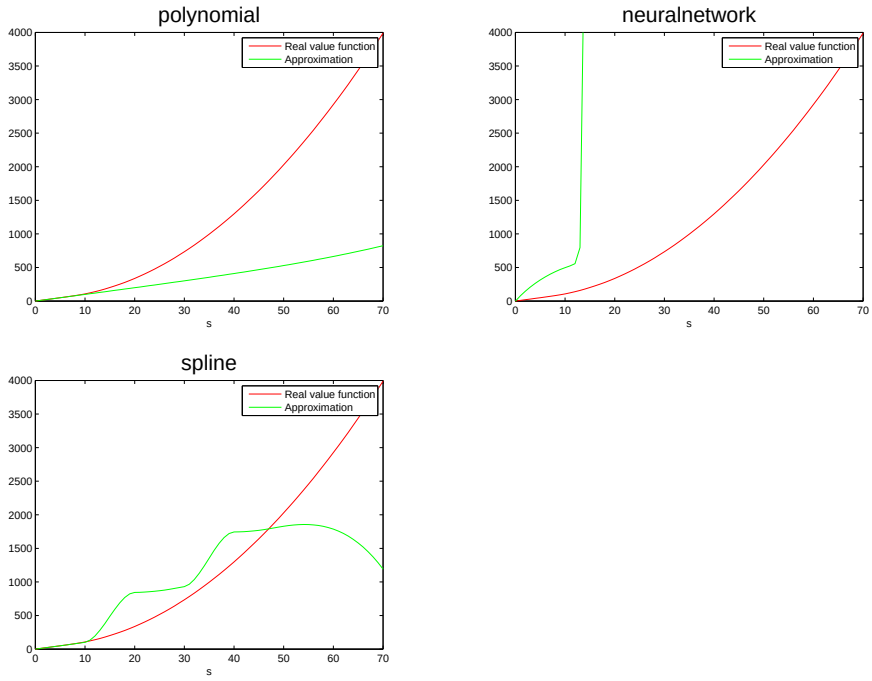


Figure 5.3: The three structures trained on $[0, 10] \cup [20, 30] \cup [40, 50]$.

Unfortunately, non of the structures is able to learn the correct value function from the disjunct intervals. Performance on $[0, 10]$ seems to be reasonable, but after that, the situation deteriorates quickly, even on intervals that were part of the fitting process. We experimented a bit more with this idea, because the results seemed somewhat surprising and disappointing to us. But after trying various other parameters, intervals and structures, we can only conclude that training on disjunct intervals has no beneficial effects on the approximations. The only consequent behavior we noticed was that the approximations were usually good on the first few states. This is probably because we took $s = 0$ as the reference state and set $V(s) = 0$. We can not give a reasonable explanation for the bad performance on the larger states in $\widetilde{\mathcal{S}}$.

5.1.8 Temporal Difference Learning

We also implemented the TD(λ) algorithm for the $M/M/c$ example. First, we investigate the influence of the constant α. Figure 5.4 shows the simulation results for various values of α. They were generated with 1000 sample paths and parameter $\lambda = 0$ for TD(λ). The $M/M/c$ parameters are as in the previous section, i.e., $\lambda = 4, \mu = 2$ and $c = 8$.

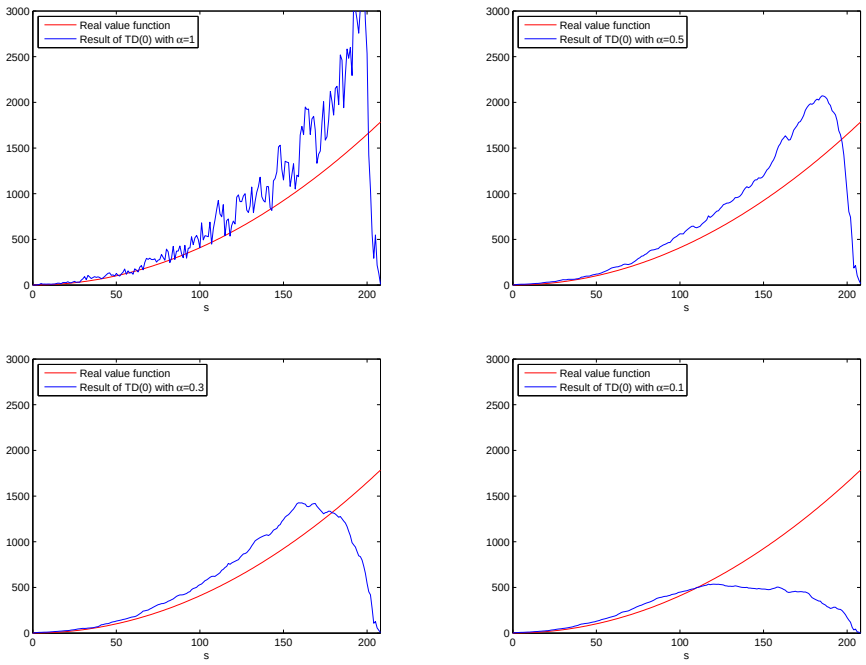


Figure 5.4: TD(0) for $\alpha = 1$, 0.5, 0.3 and 0.1.

So α has a smoothing effect, but as α gets smaller the temporal differences d_j are not propagated anymore, leading to a decrease in accuracy. For now, we fix the value of α to 0.5 and continue with an investigation of the parameter λ of TD(λ). Figure 5.5 shows the results.

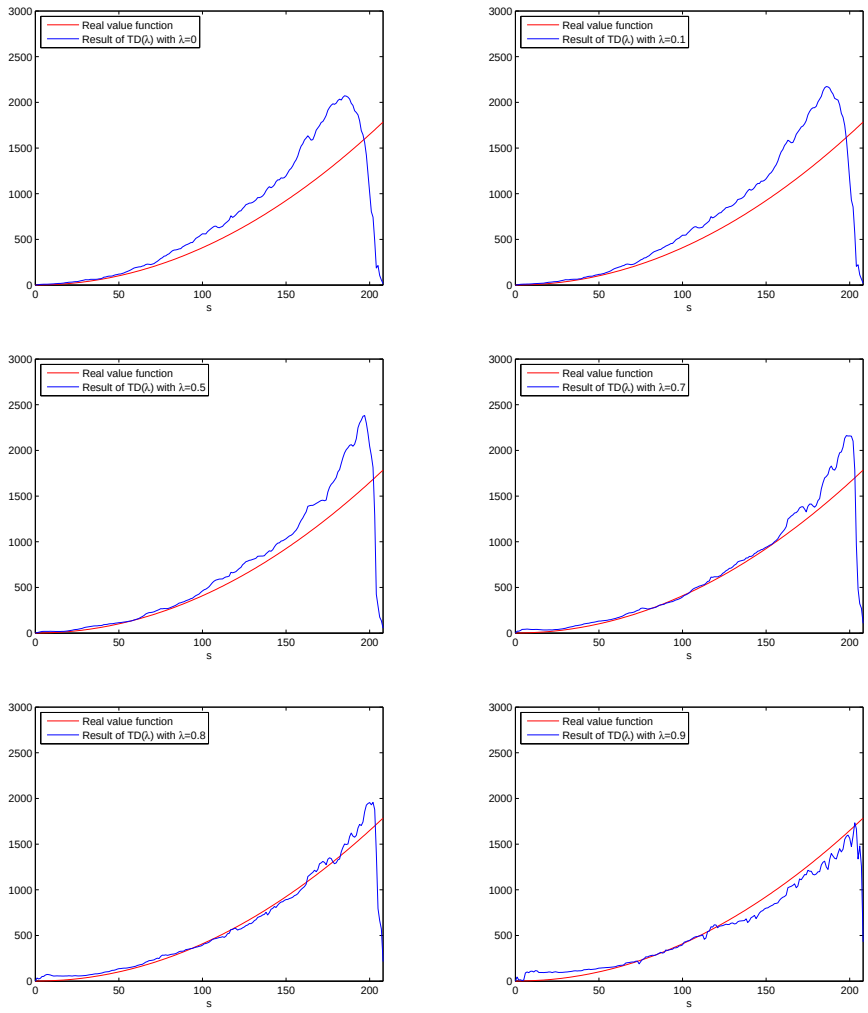


Figure 5.5: TD(λ) for $\lambda = 0$, 0.1, 0.5, 0.7, 0.8 and 0.9.

As you can see in the figure, propagating the d_j further down the sample path has a considerable positive effect on the simulation results. Although for values of $\lambda \geq 0.8$, the estimates for smaller states become more inaccurate. But in general, TD learning does a nice job.

5.2 An example: $M/M/2$ with control

In the $M/M/c$ example there is only one policy, so looking for an optimal policy with Generalized Policy Iteration is not very useful. To include control into the example, we modify the example somewhat. We assume that there are only 2 servers available, a fast server which serves with rate

μ_1 and a slow server that serves with rate μ_2. Of course, $\mu_1 > \mu_2$. At each decision epoch (which can be an arrival or a service completion), a decision needs to be made about which server to use. In Koole (1995) it is shown that the optimal policy is of threshold type, i.e., the slow server is only used if there are a certain number of customers in the system after the decision epoch.

We follow the approach of Koole (1995) and define a state as a pair (x, i), where x is the number of customers in the queue and at the first server, and $i \in \{0, 1\}$ the number of customers at the second server. The costs function that we used is $c(x, i) = x + i$.

The TD(λ) algorithm can now be applied together with the Generalized Policy Iteration scheme. We used a step size $\alpha = 0.001$ with 50000 sample paths (randomly initialized somewhere between the states $(0, 0)$ and $(40, 1)$) and $\lambda = 0.9$ as parameters for the TD(λ) method. The Generalized Policy Iteration scheme was stopped when two consecutive policies were the same in the states (x, i) with $x \leq 20$. These choices were made based on experiments, practical considerations and simulation time limitations. See also the discussion in section 4.3.4.

This does indeed result in a policy of threshold type. Table 5.2 shows some of the thresholds that were obtained from the TD(λ) method, together with those obtained from Value Iteration. As a reference, it also contains a heuristic value for the threshold, based on the observation that server 1 is μ_1/μ_2 times as fast as server 2. So if the system does not have a very high load, the threshold should be somewhere near μ_1/μ_2. Note that in Table 5.2, λ is the parameter of the Poisson Process with which the arrivals occur. The source code for this example can be found in appendix C

λ	μ_1	μ_2	T_{td}	T_{vi}	T_{heur}
12	24	2	7	7	12
8	10	2	4	3	5
5	8	2	4	3	4
5	8	1	6	5	8
4	20	4	5	5	5
4	16	2	8	7	8
4	10	2	4	4	5
3	7	2	4	3	3.5
2	10	1	10	9	10
2	8	2	4	4	4
2	6	2	3	3	3
1	2	1	2	2	2

Table 5.2: Threshold values obtained from TD(λ) (T_{td}), Value Iteration (T_{vi}) and a heuristic (T_{heur}).

So we see that most of the obtained thresholds agree with each other. For some entries, T_{td} and T_{vi} differ slightly, which is probably because both policies are optimal. In Bradtke & Duff (1994), a similar experiment is done.

5.3 An example: multi-skill call center

Now that we have some feeling for the techniques that were discussed in this paper, it is time to move to a more general example of a call center. Consider a call center with three types of calls and three groups of agents: one group for type 1 calls, one group for type 3 calls and one group of generalists that can handle all three call types. Schematically, it is depicted in figure 5.6.

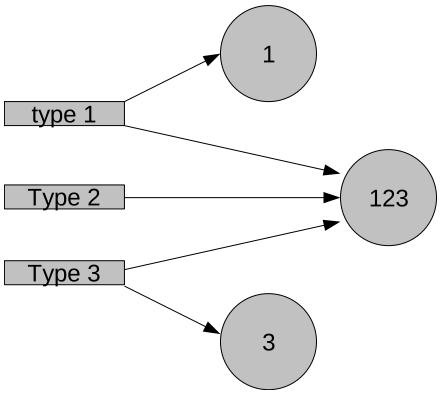


Figure 5.6: Call center example.

Each of the groups contains two agents. The service rates are listed in table 5.3. We assume that specialists have a higher service rate than generalists.

2	0	0
0	0	2
1	1	1

Table 5.3: Services rates of agent group (in the rows) vs call type (in the columns).

The arrival rate for each type is set to $\lambda_1 = \lambda_2 = \lambda_3 = \frac{2}{3}$. Because some of the heuristic policies discussed in chapter 2 assume a priority among the arriving types, we give priority to type 3, then type 2 and least priority to type 1 calls. We also reflect these priorities in the costs function: besides a costs of 1 for simply being in the system (which we assume equal for each call type), we add costs for the queue lengths of each call type. Call type 3 will have costs 3 for each customer in the corresponding queue, call type 2 has costs 2 for each customer in the queue and each customer in the queue of type 1 costs 1.

5.3.1 Comparing heuristic policies

We are now in a position to simulate the policies that were discussed in chapter 2. Table 5.4 shows the average costs of the various combinations of policies. They were obtained from a simulation of 100000 iterations. For reference, we also show the random policy. The source code is in appendix D.

		call selection				
		random	**FP**	**LQ**	**gcu**	**CR**
	random	2.0533	2.0904	403.0384	2.0912	2.0936
	HR	1.6934	1.6464	318.3202	1.6480	1.6477
agent-	**OR**	1.6934	1.6464	318.3202	1.6480	1.6477
selection	**LB**	1.6934	1.6464	318.3202	1.6480	1.6477
	VR	1.6934	1.6464	318.3202	1.6480	1.6477

Table 5.4: Average costs for various policies.

The table shows that the type of agent selection policy is not very important. This is caused by the costs function, which is focused on penalizing large queue lengths and does not distinguish between agent groups. We also see that most of the heuristic policies perform better than the random policy, proving their value. The **FP** policy is expected to work well, because we incorporated priorities into the example. The **LQ** policy performs badly, showing once again that a greedy strategy is not always best. **Gcu** and **CR**

give similar (and good) results. We should mention here that the C_i' in the **Gcu** policy were chosen to reflect the priorities in the model, so it also had some 'prior' knowledge.

We can emphasize the difference in performance between the policies better if we increase the arrival rates, because that causes larger queues and thus higher costs. If we fix the agent selection policy to **HR** and increase the arrival rates, then we get the average costs from table 5.5.

	random	**FP**	**LQ**	**gcu**	**CR**
$\lambda_i = \frac{5}{3}$	34.3259	110.8145	64505.3234	16.0478	110.8145
$\lambda_i = \frac{6}{3}$	3627.7653	2822.6822	75945.5193	486.9604	5058.0765
$\lambda_i = \frac{7}{3}$	7701.5039	6769.0164	84044.9723	4232.2200	9385.3620

Table 5.5: Average costs for various policies with higher arrival rate.

The results in table 5.5 clearly show the consequent good performance of the **gcu** policy. For completeness, we should mention that all the results from this section were obtained from one simulation run. If we would want to find the best policy, then we should do multiple simulations for each policy and compare the averages.

5.3.2 Improving a heuristic policy

We now attempt to use the MDP framework to improve one of the heuristic policies. We chose the combination of **HR** and **FP**, mainly because they are quite quick to use. The framework that we implemented in Matlab allows for the policies to be changed with minor effort, so this choice is not very important for now.

The state space that we need consists of vectors $s = (s_1, \ldots, s_9)$ such that

- s_1, s_2 and s_3 correspond to the number of customers of each type that is currently being served by agents,
- s_4, s_5 and s_6 correspond to the length of the queue of each call type,
- s_7, s_8, s_9 correspond to the number of agents in each group that is occupied.

Note that if we would choose to ignore the agent selection policy (maybe because of the results of the previous section), then the last 3 items do not

need to be in the state space. But we keep them in for now. Also note that the magnitude of the elements s_1, s_2, s_3, s_7, s_8 and s_9 is limited by the size of the agent groups. The elements s_4, s_5 and s_6 are the ones that can grow large.

Applying Value Iteration or Policy Iteration+Policy Evaluation to this problem is not possible, because we do not have the transition probabilities. Also, the size of arrays needed for storing the value function and policies is becoming a problem. If we limit the queue lengths to say L, then we need arrays of size $2 \cdot 2 \cdot 2 \cdot L \cdot L \cdot L \cdot 2 \cdot 2 \cdot 2 = 2^6 \cdot L^3$. And that is just for this tiny call center. If we want to find a way for improving heuristic policies in larger call centers, then we need to avoid using these arrays.

So our only option is to use TD(λ) combined with Generalized Policy Iteration (as an alternative to, e.g., Value Iteration) and create an approximation of the value function to prevent a large array. The section below discusses the results that we obtained.

Results

Before we could get started, we needed to address the TD(λ) related issues that were listed in section 4.3.4. The list below discusses these issues and some other optimalisations (for speed and storage) that we did.

- **Number of sample paths in TD(λ)**
  We experimented with this quite a lot and it seemed that 100000 sample paths is about the minimum amount we needed. Also, with our call center example, we could simulate these sample paths on our computer in a little over 1.5 hours. This was reasonable to work with.

- **Starting point of the sample paths**
  Simulations were started with 10 calls in the system, randomly distributed over the available agent groups and queues. This choice was made with the desired amount of 100000 sample paths and simulation time of $1 - 2$ hours in mind. Starting with significantly more than 10 calls in the system leads to longer sample paths and thus longer simulation times.

- **Choice of λ in TD(λ)**
  We set the λ to 0.9, which is probably quite high. But we suspected that with short and limited sample paths an aggressive strategy might be necessary.

- **Choice of α in TD(λ)**
  We took $\alpha = 0.1$. We wanted to set this as low as possible. But setting it any lower than this and the 100000 short sample paths did not give enough information to fit a good approximating structure.

- **Convergence of GPI**
  We did not worry about convergence of GPI. Our aim is to improve the heuristic policy, not to find the optimal policy, so we only do one iteration.

- **Which approximating structure do we use?**
  We started with a neural network, but discovered that it was quite difficult to train a good fit. We finally settled on a quadratic polynomial. More on this later.

- **When do we fit the approximating structure?**
  TD(λ) yields more accurate results as more sample paths are used. We decided to do as many sample paths as possible (100000) before fitting the approximation.

- **Storage of the value function**
  When TD(λ) is running its 100000 sample paths (and thus before an approximation of the value function exists), the value function needs to be stored. An array is not possible, so we stored the function values in a Hashtable (key/value pairs). The key is a string representation of the state vector, the value the corresponding function value. We used Java for the Hashtable implementation (java.util.Hashtable), because the Matlab *Container.Map* does not allow strings as key. The conversion of state vector to string was also done with a Java call, because the Matlab versions were much slower.

- **Storage of the policy**
  The use of an array in the policy improvement step of GPI (see step 4 of algorithm 4) is also not possible. But this problem is more easily prevented, by calculating optimal actions on the fly from the value function approximation. So we do not store the policy at all.

- **Length of the eligibility trace**
  The temporal differences in TD(λ) are distributed along the eligibility trace after each step on each sample path. This is a huge amount of effort and takes lots of time, even though the updates of the value function may be very small. To reduce the impact of this updating process, we updated states only if the element in the eligibility trace was greater than 0.01. This results in an eligibility trace of length 43. An added bonus is that this also reduces the amount of Java calls to the Hashtable.

With these choices made, we could start with the simulations. The steps below summarize our approach of using a combination of TD(λ), Generalized Policy Iteration and value function approximation:

1. Start with the heuristic policy (**HR** and **FP** in our case).
2. Evaluate this policy with TD(λ).
3. Approximate the resulting values with some approximating structure.
4. Simulate this approximation to the value function (recall that we do not save the improved policy explicitly).
5. See if the resulting average costs are similar to or perhaps even lower than the 1.6464 from table 5.4.

Evaluating the heuristic policy with TD(λ) gave no problems and resulted in some points distributed in 9 dimensional space. But that is when we ran into trouble. We needed to fit an approximating structure through these points that would result (when simulated) in average costs similar to the 1.6464 obtained by the heuristic policy itself. At first we used a Neural Network, because the other options that we programmed for the $M/M/c$ example did not directly scale to higher dimension space. Creating a good fit with a Neural Network turned out to be quite difficult. We tried training it on all points that resulted from TD(λ), only on 'frequently' visited states and on only 'large' points. Sometimes we got lucky and got a network that performed reasonably well, the best one resulting in average costs of about 12. But most of the produced networks performed badly, often with average costs higher than 40000. Inspection of these network revealed that there were two aspects that we would like have, but were seldomly learned:

- The 0 state has no costs, so we would like to have our approximating structure such that $\widetilde{V}(0;r) = 0$.
- Generally speaking, larger states correspond to higher costs. This holds especially for the elements s_4, s_5, s_6 of the state vector. They correspond to the queue lengths, which can be large and are punished in the costs function. So if we would plot $\widetilde{V}(s;r)$ along, e.g., the dimension of s_4 then we want to see an increasing function. Many of the networks that we trained were actually decreasing along some or all of these dimensions. So in a simulation run, the best action (which corresponds to the lowest value) was often to queue a call, ending in enormous average costs.

Because of these observations, we decided to try another approximating structure. We chose to fit a second degree polynomial of the form

$$\widetilde{V}(s;r) = \sum_{i=1}^{9} a_i s_i^2 + b_i s_i + \sum_{j>i} c_{ij} s_i s_j, \quad r = (a, b, c).$$

This structure already has the property $\widetilde{V}(0;r) = 0$. Also, checking whether $\widetilde{V}(s;r)$ is increasing along, e.g., s_4 can be done by inspecting if $a_4 > 0$. Fitting this function was done by minimizing the error function

$$E(r) = \sum_{k=1}^{K} w_s \left[\widetilde{V}(s_k;r) - \widehat{V}(s_k)\right]^2 \tag{5.5}$$

where K is the number of points that resulted from TD(λ). The s_k is the kth point and $\widehat{V}(s_k)$ its corresponding value (both obtained from the Hashtable in our case). The w_s are weights. At first, we used an unweighted fit ($w_s = 1, \forall s$). As with the neural networks, we tried fitting it on various subsets of the points that resulted from TD(λ). The fits were slightly better (average costs of 15000 were not uncommon), with the best getting average costs of about 6. But still, the determination of which subset to choose was made for this specific call center. We would like to have a more general method of fitting.

We hoped that choosing the weights correctly would help. We tried setting $w_s = \#visits/max(visits)$, so that often visited states (which we suspect to be accurate) get high weights and others a low weight. But this did not improve things. Another idea was to force a good fit on the smaller states by setting

$$w_s = \rho^{\sum_{i=1}^{9} s_i}, \quad \text{for some } \rho < 1.$$

This did not help either. The fit still performed considerably worse than the heuristic.

At this point, we decided to try to combine the best of both worlds. In stead of fitting an approximating structure on all states, we only did so on smaller states, which we suspected to be more accurate. For the larger states we used the heuristic policy. The results of this were more positive than the previous attempts. We used various definitions of 'small' and 'large' for the states, and each time the results were in the same order of magnitude as the score achieved by the heuristic policy on its own. For instance, using an unweighted polynomial fit of all data for the states s with $\sum_{i=1}^{9} s_i < 5$ (and the heuristic policy for the other states), gives a 'combi' policy with average costs 2.5721.

We also experimented with creating polynomial fits on some subsets of the states. That gave even more encouraging results. For instance, we created a polynomial fit on states that were visited during TD(λ) more than 100 times. The 'combi' policy that uses this fit on the states s with $\sum_{i=1}^{9} s_i < 5$ resulted in average costs of 1.6454, slightly lower than the

heuristic policy. To see if this was actually an improvement, we simulated both the 'combi' policy and the heuristic policy five times. The mean of the average costs for the 'combi' policy was 1.6938 and 1.6656 for the heuristic policy. So the 'combi' policy is not better. But, as mentioned before, this type of fit was chosen specifically for this call center example, so it can not be used directly for other examples.

So the 'combi' policy never significantly outperformed the heuristic policy, although it seems a interesting technique. Unfortunately we ran out of time to continue the 'investigation' any further. The sections below contain some ideas for future work and remarks about practical aspects of our approach.

Ideas for future work

So we did not manage to improve the heuristic policy, which leaves us slightly disappointed. But it would be unfair to say that the combination of TD(λ), GPI and value function approximation is insufficient for the task. We were able to create an approximating structure that gave average costs 6, which is in the same order of magnitude as the 1.6464 that resulted from the simulation of the policy. So the approximating structure does seems to capture the value function quite well. If we could just get the structure a bit more accurate, we could attempt to improve the heuristic policy (albeit with some patience). Also, the results from applying a 'combi' policy seemed hopeful.

So we still have some faith in the method and hope to get it working some day. If in the mean time somebody would like to continue this work, there are a few ideas and tips that might help:

- We experimented only slightly with the parameters of TD(λ) (λ and α). Perhaps tweaking of these parameters will allow for a better fit.
- The only approximating structures that we tried were neural networks and second degree polynomials. Perhaps there are other structures available. For instance, the Netlab package (see Nabney & Bishop (2003)) offers other variations of neural networks than just the multilayer feedforward network that we used.
- When experimenting with the approximating structure, it is convenient if the method of fitting can be tested on multiple datasets (i.e. multiple runs of the 100000 sample paths from TD(λ)). Matlab can save variables to disc, so we would advise creating a few datasets and storing them for later use.

- A lot of time can probably be gained by not programming in Matlab. The downside of this is that you can not use Matlab's large toolbox, which we used extensively for the approximating structures. But Matlab integrates with Java quite well, so perhaps this combination can be used better than we did.
- We did not pay very much attention to the choice of costs function. Perhaps a better choice can be made.
- The call center example that we treated was quite small and had a very low load. Perhaps the average costs obtained by the heuristic policy is already close to optimal. This would explain why there are four policies in table 5.5 that result in roughly the same average costs. Increasing the arrival rates may offer more space for improving a heuristic policy, but the downside is that simulations take much longer.

With these ideas, some time and perhaps a fresh pair of eyes, we feel confident that policy improvement is possible for this example.

Practical aspects

We would like to end this chapter with a few words on the prospects of the approach we used in a practical situation.

Our example consisted of only three call types and three agent groups, and already we struggled to get a sufficient amount of sample paths. Real call centers have dozens of call types and agent groups and continue to grow in size and diversity. With the current state of computer technology, simulating a large amount of sample paths seems wishful thinking at best.

Also, the simulations that are done by TD(λ) assume that the arrivals occur according to a homogeneous Poisson Process, which is rarely the case in practice. There is some evidence that arrivals in a small time slot (i.e. half an hour) are approximately homogeneous, so we would need a large number of approximating structures.

Then there is the costs function. Within the framework that we used, this is the only method we have of imposing constraints. We, for instance, punished large queue lengths (and thus long waiting times) with it. But there may be other constraints that we will not be able to incorporate into the costs function.

Another important fact is that most heuristic policies are quite easy to understand for managers. Our framework is as a black box to them, and it is probably not easy to convince them to change policies.

Taking all this into consideration, we feel that our approach is a long way from maturity, although that should stop nobody from researching it.

Chapter 6

Conclusions and recommendations

In this paper, we looked at various policies that can be used for skill-based routing. After an overview of some heuristic policies, we introduced Markov Decision Processes as a mathematical framework for skill-based routing. We saw how the classic algorithms Policy Evaluation, Value Iteration, and (Generalized) Policy Iteration can be used to solve an MDP and applied this to the robot example. We also showed the more recent method of Temporal Differences and explained how this can be plugged into the Generalized Policy Iteration scheme.

Then we moved on to some practical limitations of the algorithms and showed how these limitations may be solved. That lead us to the field of Approximate Dynamic Programming and the approximate versions of Policy Evaluation, Value Iteration, and (Generalized) Policy Iteration. We also discussed some of the issues connected with TD(λ) .

In the last chapter we applied the techniques from this paper to three different call centers. The first two are the classroom examples of $M/M/c$ and a variation on $M/M/2$. Both are used mainly to illustrate the techniques and develop some feeling for them. The third example is a more realistic call center, although it is still quite small due to computation time limits. We compared the heuristic policies from chapter 2 and then attempted to improve on one of the heuristic policies. Unfortunately, we did not succeed.

The main goal of this paper was to improve a heuristic policy, which we did not manage to do. Although this was somewhat disappointing to us, we do feel that our approach has some potential. It was mainly a lack of time that stopped our progress. We have given a list with some ideas that

may be of use to people who would like to continue our work. This might help getting our approach to work on the example.

At the same time, we also discussed some practical aspects that will arise if our approach is ever applied in a real world call center. Taking all these aspects into consideration, we feel that our approach is a long way from maturity, although that should stop nobody from researching it.

Besides continuing our work on improving a heuristic policy, there are other interesting and related topics to explore. The mathematical framework of Markov Decision Processes has applications in many areas and not all of them face the curse of dimensionality. Section 3.8 contains an overview with some of those topics and corresponding references. Approximate Dynamic Programming is also a 'hot topic', see the references in section 4.4.

Appendix A

Robot example

The source code for the results of the robot example can be downloaded from the internet, see Onderwater (2010). The directory contains the following files:

- **robot.m** Main file for this example. It can be used to obtain the results from chapter 3.
- **policyEvaluation.m** Function that performs policy evaluation for a given policy and returns the value function.
- **valueIteration.m** Function that performs value iteration and returns the optimal policy and corresponding value function.
- **policyIteration.m** Function that performs policy iteration and returns the optimal policy and the corresponding value function.
- **getNewValue.m** Returns the value corresponding to taking a specific action in the square (row,col). Used by policyEvaluation.m, valueIteration.m and policyIteration.m.
- **getValueAfterMove.m** Returns value of a move from the square (oldRow, oldCol) to the square (newRow, newCol). Used by getNewValue.m.

Appendix B

$M/M/c$ example

The file *mmc.m* is the main file for the example in section 5.1. It can be downloaded from the internet, see Onderwater (2010). In the beginning of the file, it is possible to specify the $M/M/c$ parameters, the set of representative states and the approximating structure to be used. There are 4 possible approximating structures:

- **spline** Setting *aType* (and also the parameter in this case) to 'spline' creates a spline as approximating structure. The parameter for this structure can also be either of the interpolation methods supported by the Matlab function *interp1*, but we only used 'spline' in this paper.
- **polynomial** This corresponds to a weighted polynomial of degree 4. The weights should be given as a parameter for this method.
- **unweighted_polynomial** No parameters are needed for this structure.
- **neuralnetwork** The parameter for this method is the *options* array used by Matlab, where, e.g., the number of training iterations can be specified.

Uncommenting the correct lines in *mmc.m* will activate the desired approximating structure. The structures are created, updated and evaluated in the files *create_structure.m*, *update_structure.m* and *eval_structure.m* respectively.

At the end of *mmc.m* a figure is created of the trained approximation and the real value function. The real value function is obtained from *get_valuefunction_mmc.m* which implements the expressions from section 5.1.3.

Together, these files give the results from section 5.1. There is one more file, called *sim_mmc.m*. With this, the value for the average costs g of

the $M/M/c$ example can be simulated. The result should match the values in the fourth column of table 5.1 (which in turn were calculated from 5.2).

The results for section 5.1.8 can be obtained with the file *td_learning_mmc.m*. In the beginning of this file, the parameters for the $M/M/c$ model and TD(λ) are specified. Creation of a sample path is deferred to the file *create_sample_path_mmc.m*. sample paths are initialized somewhere in the interval $[0, 200]$ and simulated until state 0 is reached. The value of 200 can be changed by setting *initstate* to a different value in *td_learning_mmc.m*. At the end of this file, the resulting approximation of the value function is compared to the real value function (again using *get_valuefunction_mmc.m*).

Appendix C

TD(λ) for $M/M/2$ example

The file *mm2.m* (see the downloads at Onderwater (2010)) contains the code that produces the results in section 5.2. It starts by specifying the $M/M/2$ parameters and then applies value iteration to the problem. This is done via a call to *valueIterationMM2.m* and gives a value function and the three policies. The same problem is then solved with the combination of Generalized Policy Iteration and TD(λ) via a call to *generalized_policy_iteration.m*. The policy evaluation step is done with TD(λ), see *td_learning.m*. Creation of a sample path is again done in a separate file called *create_sample_path.m*. The final lines of *mm2.m* shows the threshold values of the policies found via value iteration, GPI+TD(λ) and the heuristic.

The file *grow.m* is used in *td_learning.m* to grow arrays (if necessary) after a new sample path was created. And *getW.m* is an auxiliary function for *valueIterationMM2.m*, see the references is section 5.2 for a description of what W is. Finally, there is a file called *policyEvaluation.m* which can do policy evaluation. It can be plugged into GPI as an alternative to TD(λ) and used to compare results.

Appendix D

Call center example

The main file for the results of section 5.3 is *cc.m* (see Onderwater (2010)). The first part of it configures the call center model that we want to use. The Matlab struct *model* contains the following fields:

- *model.numberOfCallTypes* The number of call types that the model has.
- *model.numberOfAgentGroups* The number of agent groups that the model has.
- *model.lambda* The total arrival rate of the Poisson Process.
- *model.mu* Matrix with service rates per group and call type (see the matrix in table 5.3).
- *model.agentGroups* Vector with the number of agents per agent group.
- *model.agentGroupNames* Names of the agent groups. Only used for debug purposes (initialized to *['1' '3' '123']* in our example).
- *model.probs* For splitting the arriving Poisson Process in one for each call type.
- *model.costs.inSystem* Vector with costs for being in the system per call type.
- *model.costs.inQueue* Vector with costs per customer in the queue of each call type.
- *model.td.alfa*, *model.td.lambda*, *model.td.nsamples*, *model.td.initstate* Parameters for TD(λ).

Agent and call selection policies are configured similarly. Each has a *name* field. The **HR**, **OR** and **FP** have a field called *priorities*. Both **VR** and **CR**

have *alpha* and t and **gcu** has *Cprime*. A 'combi' policy is configured using its *name*, *threshold*, *policyHeur* and *policyApp* fields. See the end of *cc.m* for an example. After configuring these structs, we can simulate policies (with *simulatePolicy.m*).

TD(λ) is implemented in *tdLearning.m*. It returns a Hashtable with the value function, from which we then extract *in* and *out* vectors. After this we initialize the *weights*. Then we configure the approximating structure, again with Matlab struct. The code is similar to what we used before and should be clear from the file. Note that if fitting the approximating structure would work, this procedure would be combined with the call to *tdLearning.m* to get a GPI scheme.

As before, there are files *createStructure.m*, *updateStructure.m* and *evalStructure.m* available to work with the approximations. For the Neural Network, the Netlab package is again used. For the polynomial, we also borrow some functionality from Netlab. We use the function *scg.m* (which implements Scaled Conjugate Gradients, see the part on 'nonlinear combinations' in section 4.3.1) to minimize the error function of equation 5.5. The error function itself and its gradient are implemented in *SSE.m* and *SSEgrad.m* respectively. There is also a small helper file *SSEtest.m*, which generates some data and uses the Netlab function *gradchek* to check the implementation of the function and its gradient.

For the simulation of a sample path, we use the file *createSamplePathCC.m*. Just as *simulatePolicy.m*, it relies on *createEventNode.m* and *createQueueNode.m* to create elements to be used in the event list and queue respectively. Actions to be taken are determined depending on the agent selection policy and call selection policy. These were configured before, but their actual implementation is in files *getAgentGroupFromPolicy.m* and *getCallTypeFromPolicy.m*. There is one helper file called *getCk.m*, which calculates the C_k used in **VR** and **CR** (see equation 2.1 and the explanation below it).

Finally, there are two Java files: *State.java* and its compiled version *State.class*. They contain a static method `public static String create(int[] values)` which creates the string representation of the state vector.

Bibliography

[1] Altman, E., *Constrained Markov Decision Processes*: Chapman & Hall, *1999*.

[2] Bacon, N., 'Planning in Continuous Domains with MDPs.' http://cs.anu.edu.au/student/projects/08S2/Reports/Neil%20Bacon.pdf, *2008*. Australian National University.

[3] Bansal, N. & M. Harchol-Balter, 'Analysis of SRPT scheduling: investigating unfairness.' In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 279–290: ACM, *2001*.

[4] Bellman, R. & S. Dreyfus, 'Functional Approximations and Dynamic Programming.' *Mathematical Tables and Other Aids to Computation*, **13**, pp. 247–251, *1959*.

[5] Bellman, R.E., *Dynamic Programming*: Princeton University Press, *1957*.

[6] Beltman, F., 'Optimization of ideal racing line.' *BMI Paper*, *2008*.

[7] Bertsekas, D. & J. Tsitsiklis, *Neuro-Dynamic Programming*: Athena Scientific, *1996*.

[8] Bhat, U. Narayan, *An Introduction to Queueing Theory: Modeling and Analysis in Applications*: Springer, *2008*.

[9] Bhulai, S. & G. Koole, 'On the structure of value functions for threshold policies in queueing models.' *J. Appl. Probab.*, **40** (3), pp. 613–622, *2003*.

[10] Blackwell, D., 'Discrete Dynamic Programming.' *Anals of Mathematical Statistics*, pp. 719–726, *1962*.

[11] Bradtke, S. & M. Duff, 'Reinforcement Learning Methods for Continuous-Time Markov Decision Problems.' In *NIPS*, pp. 393–400, *1994*.

[12] Burden, R.L. & J. Douglas Faires, *Numerical Analysis*: Brooks/Cole, *1997*.

[13] Cajal, S.R. Y, *Textura del Sistema Nervioso del Hombre y los Vertebrados*: N. Moya, Madrid, *1904*.

[14] Canós, A.J., 'Polifit3 - weighted version of matlab function polyfit.' http://www.mathworks.com/matlabcentral/fileexchange/4262, *2003*.

[15] Cassandra, A.R., , *2003*'Partially Observable Markov Decision Processes.' http://www.pomdp.org/.

[16] Deisenroth, M.P., J. Peters & C.E. Rasmussen, 'Approximate Dynamic Programming with Gaussian Processes.' *Proceedings of the 2008 American Control Conference*, *2008*.

[17] Derman, C., *Finite State Markovian Decision Processes*: Academic Press, New York, *1970*.

[18] Gans, N., G. Koole & A. Mandelbaum, 'Telephone Call Centers: Tutorial, Review, and Research Prospects.' *Manufacturing & Service Operations Management*, **5**, pp. 79–141, *2003*.

[19] Garnet, O. & A. Mandelbaum, 'An introduction to skill-based routing and its operational complexities.' *Teaching note, Technion*, *2000*. Downloadable from http://ie.technion.ac.il/serveng/Lectures/SBR.pdf.

[20] Howard, R.A., *Dynamic Programming and Markov Processes*: MIT Press, Cambridge, *1960*.

[21] Jensen, A., 'Markov Chains as an aid in the study of Markoff Process.' *Skand. Aktuarietidskr.*, **36**, pp. 87–91, *1953*.

[22] Kaelbling, L.P., M.L. Littman & A.W. Moore, 'Reinforcement learning: A survey.' *Journal of Artifical Intelligence Research*, **4**, pp. 237–285, *1996*.

[23] Kleinrock, L., *Queueing Systems, Vol. II: Computer Applications*: Wiley, *1975*.

[24] Konidaris, G. & S. Osentoski, 'Value Function Approximation in Reinforcement Learning using the Fourier Basis.' *Technical Report UM-CS-2008-19*, *2008*.

[25] Koole, G. & A. Pot, 'An Overview of Routing and Staffing Algorithms in Multi-skill Customer Contact Centers.' http://www.math.

vu.nl/~koole/articles/report06a/, *2006*. VU University Amsterdam.

[26] Koole, G., 'A Simple Proof of the Optimality of a Threshold Policy in a Two-Server Queueing System.' *Systems and Control Letters*, **26**, pp. 301–303, *1995*.

[27] ——— 'Lecture Notes Stochastic Optimization.' http://obp.math.vu.nl/edu/so/notes.pdf, *2006*. VU University Amsterdam.

[28] ——— 'Lecture Notes Applied Stochastic Modelling.' http://www.math.vu.nl/~koole/obp/obp.pdf, *2008*. VU University Amsterdam.

[29] Lagoudakis, M.G. & R. Parr, 'The Essential Dynamics Algorithm: Essential Results.' *Journal of Machine Learning Research*, **4**, pp. 1107–1149, *2003*.

[30] Mahadevan, S., 'Proto-value functions: developmental reinforcement learning.' In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pp. 553–560, New York, NY, USA: ACM, *2005*.

[31] Manne, A.S., 'Linear Programming and Sequential Decisions.' *Management Science*, pp. 259–267, *1960*.

[32] Marengo, N., 'Skill-based routing in multi-skill call centers.' *BMI Paper*, *2004*.

[33] Martin, M., 'The Essential Dynamics Algorithm: Essential Results.' http://groups.csail.mit.edu/lbr/lbr/lm/2003/AIM-2003-014.pdf, *2003*. Massachusetts Institute of Technology.

[34] Mine, H. & S. Osaki, *Markov Decision Processes*: American Elsevier, New York, *1970*.

[35] Nabney, I. & C. Bishop, 'Netlab - a neural networks package for Matlab.' http://www.ncrg.aston.ac.uk/netlab/index.php, *2003*.

[36] Ng, A., 'Machine Learning (CS 229), lecture 16.' http://www.youtube.com/watch?v=RtxI449ZjSc, *2008*. Stanford University.

[37] Nortel, 'Beyond ACD– The advantages of skill-based routing in today's contact centers.' *2003*.

[38] Onderwater, M., 'Sourcecode for the simulations in this paper.' http://www.martijn-onderwater.nl/bmi-paper/, *2010*.

[39] Powell, W. B., *Approximate Dynamic Programming*: Wiley, *2007*.

[40] Puterman, M., *Markov decision processes : discrete stochastic dynamic programming*: Wiley, *1994*.

[41] Restrepo, M., S.G. Henderson & H. Topaloglu, 'Approximate Dynamic Programming for Ambulance Redeployment.' *INFORMS Journal on Computing (to appear)*, *2008*.

[42] Rojas, R., *Neural Networks; A Systematic Introduction*: Springer, *1996*.

[43] Ross, K.W., *Applied Probability Models With Optimization Applications*: Holden-Day, San Francisco, *1970*.

[44] Roubos, D. & S. Bhulai, 'Approximate Dynamic Programming techniques for the control of time-varying queueing systems applied to call centers with abandonments and retrials.' *Probability in the Engineering and Informational Sciences (to appear)*, *2009*.

[45] Roy, B. Van, 'Neuro-dynamic Programming: Overview and recent trends.' In *Handbook of Markov Decision Processes: Methods and Applications*, pp. 431–459: Kluwer, Boston, *2001*.

[46] Russel, S. & P. Norvig, *Artificial Intelligence: A Modern Approach*: Prentice Hall, *2002*.

[47] Samuel, A.L., 'Some studies in machine learning using the game of checkers.' *IBM Journal of Research and Development*, **3**, pp. 211–229, *1959*.

[48] Schrage, L.E. & L.W. Miller, 'The queue M/G/1 with the shortest remaining processing time discipline..' *Operations Research*, **14**, pp. 670–684, *1966*.

[49] Shapley, L.S., 'Stochastic Games.' *Proceedings of the National Academy of Sciences*, pp. 1095–1100, *1953*.

[50] Si, J., A.G. Barto, W.B. Powell & D. Wunsch, *Handbook of Learning and Approximate Dynamic Programming*: IEEE Press, New York, *2004*.

[51] Sutton, R.S. & A. Barto, *Reinforcement learning*: MIT Press, Cambridge, *1998*.

[52] Tijms, H.C., *A first Course in Stochastic models*: Wiley, *2003*.

[53] Tsitsiklis, J.N. & B. Van Roy, 'Feature-based Methods for Large Scale Dynamic Programming.' *Machine Learning*, **22**, pp. 59–94, *1996*.

[54] White, D.A. & D.A. Sofge, *Handbook of Intelligent Control*: Von Nostrand Reinhold, New York, *1992*.