Immune System for Virus Detection and Elimination

Rune Schmidt Jensen

IMM-THESIS-2002-08-31

Printed by IMM, DTU

Preface

This thesis is written in partial fulfilment of the requirements for obtaining the degree of Master of Science in Engineering at the Technical University of Denmark. The work has been carried out over a period of 7 months at the division for Computer Science and Engineering, CSE, in the department of Informatics and Mathematical Modelling, IMM, at the Technical University of Denmark, DTU.

Acknowledgements

I would like to thank my family for their support and encouragement during the time of completing my master degree.

I would also like to thank my friends for their support and all my co-students through my five years of study here at DTU.

Finally thanks go out to my supervisors, Robin Sharp and Jørgen Villadsen, for all their guidance and inspiration during the preparation of this master's thesis.

Lyngby, 31 August 2002
Rune Schmidt Jensen

Summary

In this thesis we consider the aspects of designing a computer immune system for virus detection and elimination using components and techniques found in the biological immune system. Already published proposals for constructing computer immune systems are described and analysed. Based on these analyses and a general introduction to modelling the biological immune system in a computer we design a computer immune system for virus detection.

In the modelling of the biological immune system we consider the use of three different kinds of *loose* matching: Hamming Distance, R-Contiguous Symbols, and Hidden Markov Models (HMMs). A complete and in depth introduction to the theory of HMMs will be given and the algorithms used in connexion with HMMs will be explained. A framework for representing the HMMs together with the algorithms are implemented in Java as part of the CIS package which is thought of as being a preliminary version of a computer immune system.

Experiments with virus infected programs and HMMs are presented. HMMs are trained on static code from non-infected programs and on traces of systems calls generated by executions of non-infected programs. The programs are infected with a virus and the HMMs ability to detect the infections are tested. It is concluded that HMMs successfully can detect virus infections in programs from static code and from traces of system calls generated by executions of programs.

Keywords: Biological Immune System, Computer Immune System, Hamming Distance, R-Contiguous Symbols, Hidden Markov Models, Virus Detection, Virus Elimination.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The immune system of the human body is a highly advanced, complex and robust system. It has the ability to kill almost any infectious agents and keep us humans strong and healthy. The immune system has evolved through millions of years, and some of the most basic principles of the immune system can be found in almost any animals and even plants. But what is it that makes the immune system so robust and is able to keep all of us alive, and would it not be perfect if we could adopt these features to make computer system more secure, reliable and robust?

More and more computer systems are today compromised by security flaws, infected by computer viruses, and denied servicing clients, all resulting in a loss of many man-hours and great sums of money spend trying to bring back the computer systems into normal state as before the attack. Would it not be nice if we were warned if something like this was about to happen or even better, the system was able to take some kind of action disabling or denying the intrusion, or preventing the outcome of the attack itself.

Many computer systems keep getting larger and more complex, constantly introducing new security flaws and ways of attacking the systems. Normally when software is released the first time, they will still have some small errors or even some security flaws. When these are found normally a patch will be available or the error will be fixed in a new version of the software. But by introducing a new version of the software new errors and new security flaws might be introduced. These will again compromise the system and a continuous circle of introducing new errors and security flaws has started; see figure 1.1 on the next page.

We need to figure out new ways to protect our systems because often software is released before proper testing has been carried out, and without proper testing the released software might include security flaws.

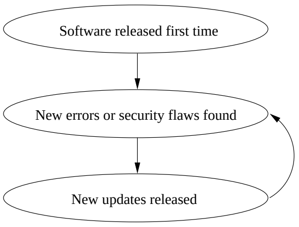

Software released first time
New errors or security flaws found
New updates released

Figure 1.1: The life cycle of software.

1.1 Protection Against Viruses

We will especially in this thesis look at how we can protect our systems from viruses. A virus is in context to a computer defined as a program that replicates itself by copying its own code into environments. The virus normally copy itself into environments like executable files, document files, boot sectors and networks. Through its replication the virus takes up system resources and often carry out some sort of damaging activity.

The most widely used method for detecting viruses is the virus scanner, which uses short sequences of code to identify particular viruses. The short sequences are also known as virus signatures. Unfortunately even small changes made to the viral code makes the virus scanner unable to detect already known viruses and the virus scanner often needs updates to recognise new viruses and new variants. Often human experts make the signatures by converting the binary code of the virus into assembler code, picking out sections of code that appear to be viral, and identifying the corresponding bytes in machine code. The analysis made by human expert is often tedious and time consuming, and the number of viruses are still increasing due to that fact that automated virus writing programs are easily available. This together with the increasing connectivity through the Internet results in greater spread of viruses than ever seen before.

We will in this thesis look at how we can use other methods than the virus scanner to recognise the viruses. We will especially look at how we can recognise viruses by their patterns of behaviour, using components and techniques seen in the biological immune system. With reference in already published proposal for constructing immune system and the techniques found in the biological immune system, we will design a computer immune system for virus detecting.

1.2 Overview of the Thesis

Chapter 2 will start out by giving an introduction to some of the central properties of the immune system. It will explain the components and techniques of human immune system and introduce some terms often used when referring to the immune system. Chapter 3 and 4 will describe some of the already made systems, which are inspired by some of the mechanisms in the biological immune system. We will especially in chapter 3 look at IBM's computer immune system for virus detection and elimination, and in chapter 4 look at two systems from the University of New Mexico (UNM) made for intrusion detection. In chapter 5 we will general look at how we could model the most important components and techniques from the biological immune system in a computer and in chapter 6 we look at how we could design a computer immune system for virus detection. In chapter 7 we look at some experimental results made with some of the software developed for a preliminary version of a computer immune system and finally in chapter 8 we propose future work and draw some conclusion from the thesis.

Attached to this thesis is also appendix A-D. Appendix A gives a more scientific introduction to the immune system than given in chapter 2 and presents the immune system of the human body from an immunological viewpoint. Appendix B gives an introduction to the different kinds of viruses and explain some of their properties. Appendix C gives a full and in depth introduction to the theory of Hidden Markov Models (HMM), presents the algorithms used in connexion with HMMs, and discuss implementation issues. Finally appendix D describes implementation, functionality test, and usage of the developed software.

Chapter 2

The Immune System of the Human Body

How does the immune system of the human body keep us strong and healthy and what kind of mechanisms are used to achieve this robustness and strength? This chapter will answer the questions and explain the mechanisms involved.

The immune system consists of billions of cells carrying out their own little task interacting locally with the environment. The interaction is done through chemical signals and bindings of proteins resulting in the cell differentiating and maybe releasing new signals to other cells. The billions of cells acting on their own make the immune system *highly distributed* and *error tolerant*. If one little cell does something wrong e.g. killing another healthy cell, the fault is not that big, because there are billions of others cells doing the right job of killing harmful cells. Furthermore, the cells most fit for the job will receive necessary resources for survival and reproduction, thereby assuring that only the best cells will survive; all others will die.

In the nature of being a highly distributed system comes a robustness against attacks on central points disabling the hole system at once. There is no central point in the immune system to attack, because there is no *central control*: all communication and stimulation is done through the local environment, and if you for instance removed a thousand cells, the immune system would still be able to cope with an infectious agent.

Another kind of robustness is the generation of millions of new cells every day – kill a million cells today and they will be back tomorrow! Some of these new cells originate from the bone marrow and are matured in the thymus, an organ just behind the breastbone, but others will be generated from existing cells dividing themselves. Especially cells which have encountered an infectious agent will be able to multiply themselves to enhance the destruction of the same kind of infectious agents; this is known as *clonal expansion*. Take for instance a virus which has infected a lot of healthy cells, and one of the cells of the immune system recognises the virus and knows how to destroy it. The cell which recognises the virus will then clone itself and thereby enhances the body's possibility of killing the virus. The idea behind clonal expansion is quite simple and used by all living beings: if you know how to achieve and handle some kind of resource, lets say the finding and preparation of food, you are able to survive and produce offspring. The clonal expansion enables the immune system to grow with the assignment, creating enough cells to cope with the intrusion.

The survival and clonal expansion of the fittest, leads us to another characteristic of the immune system, known as *immunologically memory*. The best cells to recognise and kill the virus fast and effective will proliferate into cells known as memory cells. These cells are especially good at killing viruses and can recognise them very fast. This enables the human body to be immune against already known diseases and we as human are not even able to notice that we are infected a second time with a virus, because the response is so fast and effective.

Primary Response: the immune system needs to learn how to fight the infectious agents, this take time and the response is therefore slow.

Secondary Response: the immune system already know how to fight the infectious agents, the response is fast.

As mention before, communication is done locally through chemical signals and bindings of proteins. This enables the cells of the immune system to "call for help" when needed. When an infection occur, the cells of the immune system will try contaminate it and prevent it from spreading to the hole body, but the cells will also start releasing chemical signals such as *cytokines* and *chemokines*. These two products will attract more cells of the immune system to the site of infection, thereby increasing the immune system's ability to withstand and fight the infection. This mechanism of attracting more cells to the site of infection, is clearly a great advantage of the immune system, enabling the body to respond faster and quicker when fighting and eliminating an infection.

To make it even harder for an infection to invade the body, the immune system consist of a *layered system of defence mechanisms*, each specialised to practise different types of protection. This makes the immune system more robust and strong, and only the most toughest and withstanding infections are able to reach the inner defence system. The first line of defence is the skin, built from tight junctions of cells, forming a seal against many infectious agents. If the skin is compromised by wounds or the infection has found its way through the respiratory parts of the body, it will be met by another line of defence: low pH-value and temperature. Low pH-value together with the body's temperature of 37 °C give bad living conditions for a lot of infectious agents. If these two first defence systems are compromised, a third defence system known as the innate immune system will be engaged. This defence system is denoted innate because it is inherited from our parents, and is able to recognise and eliminate a lot of known infectious agents. The last line of defence is the adaptive immune system, which is clearly the most interesting, because it is able to tell harmful substances from good ones and able to *adapt* itself to the given environment. Being able to adapt to different environments makes the immune system of every human being more unique: some humans are able to withstand a special kind of infection whereas others get sick. This *uniqueness* or *diversity* of every immune system is a feature that we are very interested in, it makes the systems harder to break. Normally when a system is compromised, it is because someone has found a security breach in the system. This security breach is then used to compromise thousand or even millions of other similar systems. All the systems are the same and have the same kind of security breach, enabling the intruder to use the same kind of procedure when breaking the systems. If every system were unique in some way, the intruder might be able to compromise a few systems, but would not be able to use the same procedure to compromise all the systems. This kind of problem is for example seen with Internet worms attached to emails, which are using the same kind of security flaw in a widely spread email reader program to replicate itself to other systems.

All the cells of the immune system repeatedly need stimulation from the surrounding environment. This enables the immune system to have a kind of distributed local control, regulating the number of cells and assuring that they are working correctly. If the cells are neglected the stimulation, they will die from programmed cell death, also known as apoptosis. With the immune system having a distributed local control it is stronger and more robust: no infectious agent is able to take complete control of the immune system, because there is no central control point to attack.

The immune system is a *dynamic* system, all the cells of the immune system are constantly circulating the human body, enabling cells to constantly meet with others cells, communicating and affecting each other. In this way the immune system is changing all the time, which is clearly also one of the great advantages of the immune system: how do you know where to strike a system if it is constantly changing? The immune system is also dynamic in the sense, that it can supply new cells when they are needed and dispatch or remove them when the treat is over. When an infection is met, the cells of the immune system start to release chemical signals attracting other cells to the site of infection, and by clonal expansion one cell can multiply itself into thousand of daughter cells each helping in fighting the infection. When the infection is eliminated the cells no longer needed will die by apoptosis because they no longer receive stimulation from the environment, or they will go to other destinations receiving stimulation to survive there instead.

The immune system is also clearly a *learning* system in the sense that it is able to remember previous encountered infectious agents, known as immunologically memory. But it is also a learning system in the sense that it is able to learn to distinguish between good and bad, this is often defined as distinguishing between *self* and *nonself.*

Self: harmless substances including the body's own cells.
Nonself: substances which are harmful to the body.

2.1 The Adaptive Immune System

The cells of the immune system, which are taught to distinguish between self and nonself are known as *lymphocytes*. The learning of the lymphocytes takes place in two different parts of the body: the thymus, which is an organ just behind the breastbone, and in the bone marrow. We normally distinguish the lymphocytes taught in the thymus from the ones taught in bone marrow because they have different kinds of purposes. The lymphocytes which are taught in the thymus are referred to as T-lymphocytes, from the T in Thymus, whereas the lymphocytes taught in the bone marrow are called B-lymphocytes, from the B in Bone marrow. The communication between the lymphocytes and other cells is done through receptors on the lymphocyte's cell surface. The receptors are able to bind to different kind of peptides, and when a certain number of receptors are bound, the lymphocytes will be activated to carry out some sort of action. To learn the lymphocytes to distinguish between self and nonself they are exposed to self peptides in the thymus and in the bone marrow. Those who react strongly to self peptides will be killed in a process known as *negative selection*, whereas those who are not able to recognise self peptides will survive in a process known as *positive selection*. Figure 2.1 illustrates the process of training the lymphocytes to distinguish between self and nonself.

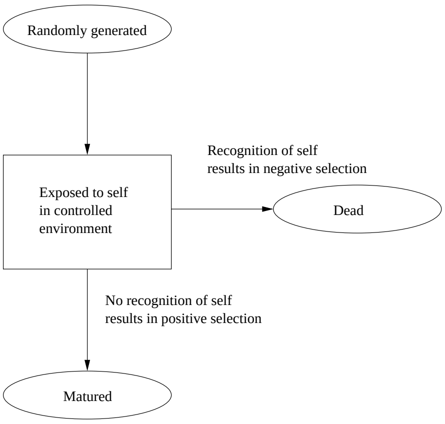

Randomly generated
Exposed to self in controlled environment
Recognition of self results in negative selection
Dead
No recognition of self results in positive selection
Matured

Figure 2.1: The positive and negative selection of lymphocytes.

The receptors of the lymphocytes are created by randomly gene rearrangements in the cells, enabling the lymphocytes to recognise almost anything. Each lymphocyte is thought to have around 30.000 receptors on the cell surface, and each receptor on a lymphocyte has the same specificity, meaning that each receptor recognises the same peptides and are alike. Only 10-100 of these receptors needs to be bound to peptides before a lymphocyte will be activated and can carry out some sort of action. The thymus is thought to generate over 10^7 T-lymphocytes every day, but only $2-4\%$ of these will survive the negative selection. This is because the process of positive selection also requires that the lymphocyte's receptors are able to receive signals and react on these, and clearly not many randomly generated receptors are capable of this. The immune system of the human body is though to constantly having 10^8 lymphocytes circulating in the body at any time, again giving an impression of how big and complex the immune system is.

Each lymphocyte in the human body follow the same life cycle, they are created and taught to distinguish between self and nonself, the ones who recognise the self peptides will die whereas the others will be released to circulate the body. While circulating the body some might recognise peptides and get activated, the activation will result in clonal expansion and all the daughter cells will carry out their specific actions. The best of the daughter cells will be selected as memory cells, whereas the others will die. Figure 2.2 illustrates the life cycle of a lymphocyte.

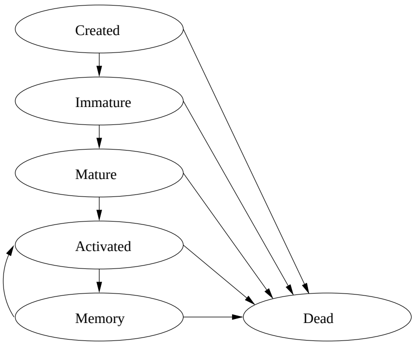


Figure 2.2: The lymphocyte's life cycle.

When circulating the body the lymphocytes constantly need signals from the environment to survive, otherwise they will die from apoptosis. They are like ticking bombs, they constantly need to be reset, if not to blow up and fall apart.

The description below will explain the states of the lymphocyte's life cycle more exact:

Created: Both the B-lymphocyte and the T-lymphocyte originate from stem cells in bone marrow, the T-lymphocyte travels to the thymus to mature whereas the B-lymphocyte stays in the bone marrow. Their receptors are developed through randomly rearrangements of the stem cell's receptor genes.

Immature: In a controlled environment the lymphocytes are exposed to self, the ones who react strongly to self peptides or are not able to receive stimulation signals through their receptors will die in a process known as negative selection. The ones who are able to receive signals and does not react strongly to self peptides will survive in a process known as positive selection. The surviving naive lymphocytes will now be released to circulate in the body.

Matured: Circulating the body the naive lymphocyte's receptors might bind to peptides and get activated. The lymphocytes released from the thymus and the bone marrow will not bind to any self peptides, they have undergone the positive selection, and are therefore only expected to bind to nonself peptides. If some self peptides have not been present in the thymus or in the bone marrow the lymphocytes might react on self peptides resulting in an autoimmune response. Several receptors on the lymphocytes cell surface need to be bound before the lymphocyte will be activated. The binding of self peptides does not have to be exact, but the stronger the binding is, the stronger a signal is send to the lymphocyte, and the less receptors need to be bound to activate the lymphocyte. The term *affinity* is often used as describing the strength of binding the peptides: the better affinity of the receptors the stronger a signal will be send.

Activated: When enough receptors on the lymphocyte have been bound to peptides, usual around 10-100, the signal from the receptors to the lymphocyte will be strong enough to activate it. When activated the lymphocyte will start to enlarge and divide into thousand of daughter cells - the clonal expansion. The daughter cells are also known as effector cells, because these are the cells which are going to help the immune system in fighting the infection. The actions carried out by the effector cells are quite different: the effector cells evolved from the B-lymphocyte will release antibodies, whereas the cells evolved from the T-lymphocyte will help in activation of other cells or killing the virus infected cells to stop the virus from replicating further.

Memory: When the infection is eliminated, a few effector cells – the ones with the highest affinity – will receive further stimulation from the surrounding environment to survive. All the others effector cells, not receiving stimulation, will die. The few effector cells receiving stimulation are known as memory cells because they are able to remember the infection. The memory cells will have a very high affinity, and the memory cells evolved from the B-lymphocyte will undergo even further *affinity maturation*, enabling the immune system to strike even faster and harder the next time the infection is meet.

Dead: The lymphocytes are constantly facing the thread of not receiving stimulation from the local environment and thereby dying from programmed cell death. The lymphocytes need to receive signals through their receptors to survive and reproduce. This is quite a remarkable way of the immune system to assert that there is a constant number of lymphocytes present in the body, and to assure that the receptors of the lymphocytes are working, able to bind to peptides and able to activate the lymphocyte.

As mentioned before the recognition of infectious agents is done by binding of peptides to the lymphocyte's receptors. But not all infectious agents display their peptides to the lymphocytes, disabling the lymphocytes to recognise them directly. This is for example the case with a virus, which has infected a healthy cell. The infected cell needs to decompose the virus into small peptide fragments and display them on the surface, before the lymphocytes are able to recognise the virus. To display the small peptide fragments on the cell's surface, molecules known as *Major Histocompatibility Complexes* (MHC) are used by the cell. These molecules will take up the small peptide fragments in the cell and travel to the cell's surface, displaying them to the lymphocytes. Normally only the T-lymphocytes will bind to peptides displayed with MHC molecules, this is because they are taught in the thymus only to react on peptides displayed with MHC molecules. Whereas the B-lymphocytes are able to recognise all the other kinds of peptides. In this way the adaptive immune system has evolved the two different lymphocytes into recognising and responding to two different types of infectious agents. The T-lymphocytes recognise and respond to intracellular infectious agent's peptides displayed on cell's surfaces with MHC molecules, whereas B-lymphocytes recognise and respond to extracellular infectious agent.

2.2 Failures of the Immune System

All though the immune system seems to be the perfect defence system it also sometimes makes mistakes. The mistakes reveal themselves when the immune system erroneously kills some of the body's healthy cells, also known as an *autoimmune response*. However killing one or two healthy cells is not a big problem, because the body have plenty of other similar cells ready to take over and carry out the function of the killed cell. What really could be a big problem is if a lymphocyte has reached the state of a memory cell, recognising a specific kind of healthy cell as being bad. This kind of action could be very dangerous for the body, because the memory cell gets activated really easy and quick, enabling it to strike very fast and hard against the recognised healthy cells. So before any lymphocyte reach the state of being a memory cell, the immune system needs be very sure that the recognised cell really is an infectious agent. The immune systems way of solving this kind of problem is to have an extra confirmation from others cells, telling them whether the recognised cell is bad or god. The extra confirmation comes from T-helper cells, which are actually T-lymphocytes especially evolved to help in activating other cells.

Another kind of failure the immune system can cause, is when it is not able to recognise the infectious agents. This could for example happen if a virus has infected a healthy cell, and the healthy cell is not able to express the virus's peptides on its surface. If this is the case the T-lymphocytes are not able to see that the cell is infected, because the virus's peptides need to be expressed on the cell's surface before the T-lymphocyte's receptors can recognise them. But it could also happen if the T-lymphocytes simply can not recognise the virus peptides displayed by the MHC molecules as being harmful.

When the immune system fails, we often distinguish between two different types of failures:

False Positive: We recognise the substance as being harmful, but it is not.

False Negative: We do not recognise the substance as being harmful, but it is.

One could imagine how easy it would be, causing the immune system to make false positives and false negative if for instance the thymus or the bone marrow was corrupted. If for example a part of self was removed from the thymus or the bone marrow, then the newly trained lymphocytes would recognise the removed part of self as nonself, resulting in elimination of cells important for the body. And in the same way, if a part of nonself had found its way into the thymus or the bone marrow, then the newly trained lymphocytes would not be able to recognise cells which where actually harmful to the body. Clearly the thymus and the bone marrow are fragile points in the immune system if corrupted, because all the lymphocytes need to go there to train. Even though this might seem as a central point to where the immune system could be fragile, there is no indication in the literature [1] on immuno biology that indicates this kind of attack. Failures of the immune system are more known to happen if the infectious agents are able to constantly change their structure, enabling them to hide and survive from an immune response. Or if the immune system has some kind of inherited failure such as gene defects, making them unable to recognise a specific kind of infectious agents. Or if the infectious agents are able to infect the cells belonging to the immune system, resulting in the immune system slowly killing itself, making it more easier for any kind of infection to defeat the immune system.

Chapter 3

IBM's Computer Immune System

We will in this chapter look at a commercial immune system for virus detection and elimination made by the company IBM. We take reference in some of the articles published on anti virus research for immune systems on IBM's web page: http://www.research.ibm.com/antivirus/SciPapers.htm; see [2–6].

IBM have developed the immune system over several years and released it as a part of their anti virus product in the beginning of 1997. Today IBM's immune system is used in cooperation with another anti virus firm, Symantec.

The chapter will start out by a short introduction to IBM's immune system for virus detection and elimination. Here we will describe which components of the biological immune system that the researchers from IBM have focused on. Then we give a short overview of the system and the steps towards automating a response to an unknown virus. The third section will describe how the system detects an unknown virus, and the final section will go into depth with the automated response engaged by the system to produce a prescription for detecting and eliminating a new unknown virus.

3.1 Introduction

The researchers of IBM's immune system take reference in both the innate and the adaptive immune system, stating that a computer immune system most include components from both these systems. They see the innate immune system as a way of sensing abnormalities in a generic way and the adaptive immune system as a way of identifying viruses very specific and use this precise identification to detect and eliminate them.

The innate immune system focus on detecting the presence of a broad range of unspecific viruses and ships them on to the adaptive immune system which automatically derive specific prescriptions for detecting and removing the virus. The innate system is implemented at the client PC, whereas the adaptive immune system resides in a central virus lab at IBM due to performance, implementation, and security issues.

The prescriptions derived in the virus lab are by analogy with the term known as immunological memory and the immune system's ability to withstand previous encountered viruses. The process of deriving the prescription is by analogy with the process of producing the huge amount of immune cells and antibodies that the biological immune system uses to eliminate the virus. The passing of the prescription to the infected machine and the availability of the prescription through updates from a web site is in analogy with the term known as clonal expansion.

3.2 Overview

The computer running IBM's anti virus product is connected through a network to a central computer which analyses the viruses. The anti virus product consists of monitor program which uses a variety of heuristic based methods to observe system behaviour, suspicious changes to programs and family signatures to detect the present of a virus. Family signatures are prescriptions that catch all different members of a viral family, including unknown ones. The monitor program looks for viruses such as file viruses, boot-sector viruses, and macro viruses; see appendix B on page 105 for a further description of these terms. When detecting an abnormality in the system the anti virus product scans for known viruses; if the virus is known the virus is eliminated, if the virus is unknown a copy of the infected file is send over the network to the virus-analysing computer. The software on the virus-analysing machine lures the virus into infecting decoy programs, bringing the viral code out of the hiding. Any infected decoy program is automatically analysed and a prescription for recognising and eliminating the virus is produced. The prescription is sent back to the infected machine and the anti virus program is instructed to locate and eliminate all instances of the virus. The prescription is also added to IBM's virus database and made available from their web site. The computer immune system carries out of the following steps:

1. Detect abnormality and scan for known viruses, if virus is known all instances is eliminated.
2. If the virus is unknown a copy of the virus is captured and sent to a central computer.
3. The virus is analysed and a prescription for detecting and eliminating the virus is automatically derived.
4. The prescription is send back to the user's computer and the anti virus product is instructed to eliminate all instances of the virus.

5. The prescription is send to other computers and made available on a web page.

The first step in the list is seen as an analogy to the innate immune system, whereas the third step is seen as an analogy to the adaptive immune system. These two steps will be described more fully in the next two sections.

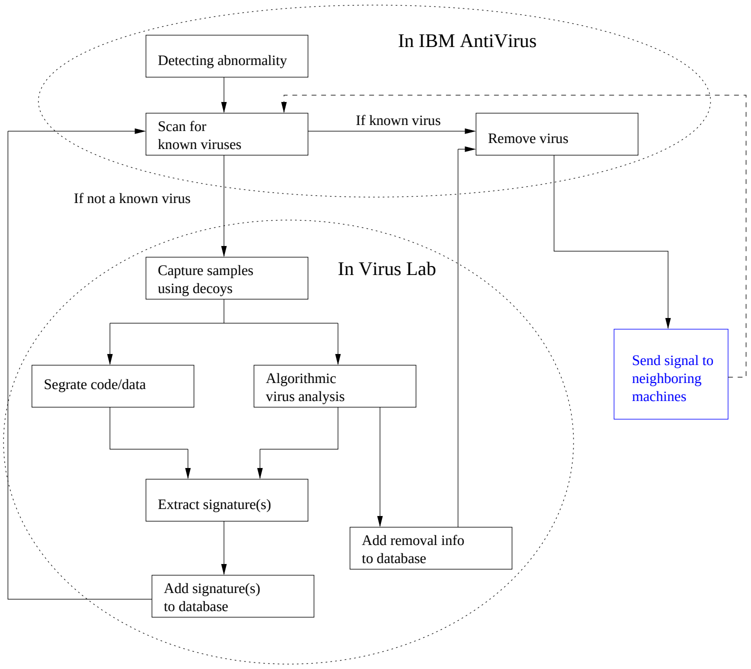


Figure 3.1: Overview of IBM's computer immune system. The component for sending signals to the neighbouring machines is not yet implemented.

To get a better overview of the immune system figure 3.1 sketches the components of the proposed computer immune system. It is label proposed because a component for sending signals to neighbouring machines is not yet implemented in the current system. The missing component is thought to assist in sending out prescriptions to neighbouring machines. When an infected machine receives a prescription and an instruction to eliminate a virus, the idea is that the infected machine should send the prescription on to the neighbouring machines, thereby hoping to stop the virus from replicating further through the network and increase the responds time for detecting and eliminating the virus.

Figure 3.1 also sketches the producing of prescriptions in a more detailed way than explained until now, we will come back to this in section 3.4 on the next page, but first we describe how the computer immune system is able to detect the abnormalities.

3.3 Detecting Abnormality: the Innate Immune System

IBM's immune system uses different kinds of techniques for detecting viral activity. To discover file viruses they use *generic disinfection* based on "mathematical fingerprints", and a classifier system which detects viruses based on machine code carrying out tasks specific to file infectors. For boot sector viruses they use a classifier system based on machine code carrying out tasks specific to boot sector infectors. For macro viruses they use fuzzy detection with simple variants of already known ones.

The principle in *generic disinfection* is that an infected program can be restored to its original form from a fingerprint and the infected program. The algorithm will compute a fingerprint of every executable file on the system and save it in a database when installed first time. The fingerprint carry information on size, date and various checksums. When the system is scanned a fingerprint of each executable file is computed and compared with the one in the database. If the fingerprint has changed the file will be scanned for known viruses. If it contains a known virus the algorithm will try to repair the program and eliminate the virus from known prescriptions. If no known virus could be found in the program the algorithm tries to reconstruct the original program from the fingerprint and the modified program.

The principle in the classifier systems is to train the systems to recognise machine code which carries out tasks common to viral infection. The classifier systems can only detect the presence or absence of special features, it is not able to repair or eliminate the virus. Features are short byte sequences that are common to viruses. The classifier system is trained through a set of known viruses and a set of non-viruses. First the system obtains all the features from the set of known viruses, second it eliminates all the features in the system that appear in the set of non-viruses. In this way the system learns which features that are common to viruses and which are not. The number of times each feature occur in a sample is added up and multiplied by weights; if the result is greater than some given threshold the sample is considered to be infected with a virus.

3.4 Producing a Prescription: the Adaptive Immune System

When a copy of a possible virus is received at the virus lab it is first scanned with an up to date scanner because it is possible that the user's anti virus program is out of date with the newest virus signatures. If it is established that the copy could not be found among the set of known viruses it is encouraged to replicate itself. This is done in controlled environments appropriate for the virus. The software running on the virus analysing machine lures the virus into infecting "decoys", these are elements especially designed to be attractive for the virus to infect. The decoys are executable files like COM and EXE, floppy disk and hard disk images, or documents and spreadsheets. To lure the virus into infecting the decoys the operating system will in some way touch the decoys by for instance executing, reading, writing to, copying, opening or in some other way manipulate them. The decoys are placed where the most common programs and files are located in the system, this is typically in the root directory, system directories, current directory, directories in the path, etc. The decoys are periodically examined to see if the virus has infected any of them, and the decoys are also placed in several different environments like different versions of operating systems, different dates and so on to enhance and maybe trigger the infection.

Samples of the virus captured using the decoys are now used to analyse the virus. Several steps are taken in the process of generating a prescription.

Behavioural Analysis: Through the replication of the virus several log files have been written by the specially designed decoys and the algorithm luring the virus into infecting the decoys. These log files are analysed to provide a description on which kind of host type the virus infects: boot sectors, executable files, spreadsheets, and so on. It also provides information about how the virus behaves, does it go resident, is it polymorphic, does it have stealth capabilities, and so on; see appendix B on page 105 for a description on these terms if not familiar. Furthermore, the analysis will also describe under which conditions the virus will infect (executing, writing to, copying, etc.).

Code/Data Segregation: A modified chip emulator is used to reveal the virus's complete behaviour, all conditional branches in the virus code are taken so every path in the virus code is revealed. The portion of virus executed as code is identified and the rest is tagged as data. Portions represented as code are normally machine instructions with the exception of bytes representing addresses. Data is code representing numerical constants, character strings, screen images, addresses, etc.

Auto Sequencing: To be conservative they only use portions of the virus tagged as code in the further processing, this is because the data are inherently more likely to vary from one instance of the virus to another than the code portions are. Code from one sample is then compared with code from other samples using pattern matching. Infected regions in the code which tend to be constant are classified as being invariant regions. Various heuristics are used to identify the sections of the virus which are unlikely to vary from one instance of a sample to another. The output of the auto sequencing is byte sequences having a size of 16 to 32 bytes, which are guaranteed to be found in each instance of the virus. The byte sequences are denoted candidate signatures because one or few of them are used as the final signature. Effort is also made to locate the original code in the samples, these informations are used to repair other files infected with the concerned virus.

Automatic Signature Extraction: The purpose of this step is to find one or few signatures among the set of candidate signatures, which are least likely to lead to false positives, in other words those candidate signatures which are least likely to be found in a collection of normal, uninfected code (self). The normal process for finding the probability of observing a full candidate signature in a collection of normal and uninfected code is rather time consuming. The researchers therefore divide the candidate signatures into smaller byte sequences and use standard speech recognition techniques to observe and combine these probabilities into an estimate for each signature. The following procedure is carried out:

1. Form a list of all sequences of 3-5 bytes from a candidate signature.
2. Find the frequencies of observing these small sequences in over 20.000 ordinary uninfected programs.
3. Use simple formula's based on Markov Models to combine the frequencies into a probability estimate for each candidate signature.
4. Do step one to three for every candidate signature and select the one which has the lowest estimate of being found in the collection of ordinary programs.

Testing: The signature is tested on the decoys, and the repair method is tested to see that it corresponds to the original version of the decoy program. If test results are okay the signature and removal information are saved in a database and combined into a prescription which is send to the client's PC and made available from a web page as a new update to the anti virus product.

Every step in producing the prescription is automatic, human experts are only involved when the constant regions between every sample are so small that it is not possible to extract a candidate signature from them. But how good is the statistical method for extracting signatures and why is it a good idea. Tests made by the researchers show that the method can cut down the response time for a new virus from several days to a few hours or less. It takes time for humans to carry out the job and it is not always that interesting and amusing to find virus signatures from thousand of assembler code lines. Other test results also showed that the signatures automatically generated by the method are in fact better than the ones produced by human experts; they simply cause less false positives. The improved signatures together with the automated process of generating the prescriptions save money and time, and are more efficient, IBM therefore sees the computer immune system as a new and improved way of fighting computer viruses.

Chapter 4

UNM's Computer Immune Systems

Researchers at the University of New Mexico (UNM) have for over 10 years been occupied with incorporating many properties of the biological immune system into computer applications to make them smarter, better and more robust. They have developed applications for host based intrusion detection, network based intrusion detection, algorithms for distributable change detection systems, and methods for introducing diversity to improve host security. All applications use components and techniques from the biological immune system to a more or lesser extent.

We will in this chapter take a closer look at some of the applications, their design and the researchers considerations when modelling the components and mechanisms of the biological immune system. We will mainly look at the intrusion detection systems, starting with an application using system calls to detect host intrusions, and ending with an application using network connections and an artificial immune system to detect network intrusions.

4.1 Intrusion Detection using Sequences of System Calls

The researchers from UNM have developed an application for detecting intrusion using abnormalities in sequences of system calls. The application is built for a UNIX system and tested on both Linux and Sun operating systems. The application trace system calls in common UNIX programs and are for instance able to detect intrusions done with buffer overflows in programs.

The general idea is to built a profile of a normal behaviour for a program of interest. Any deviations from this normal behaviour are treated as abnormalities indicating a possible intrusion. First they build a database of normal behaviour through a training period and then they use this database to monitor the program's further behaviour.

4.1.1 Defining Normal Behaviour

To define the normal behaviour of a program the application scan traces of system calls generated by the program when executed. The application then builds a database of all unique sequences that occurred during these traces. To keep it simple each program monitored has its own database, and system calls generated by other programs which has been started from the existing program are not monitored.

If the application is used on different machines it is very important that every application has it own databases, making each application more robust against similar kinds of intrusions. The reason for taking this approach is also because it is often seen that computers with the same kind of configurations will have the same kind of vulnerability, giving each applications its own databases will prevent this.

To build a database for a program a window of size k is slided across the trace of all system calls made by running the program. The system calls observed within such a window is denoted a sequence, and only unique sequences are use in the further processing. The technique of extracting the unique sequences is illustrated on the trace of system calls: open, read, mmap, mmap, open, read, mmap with $k = 3$:

$$\underbrace{\text{open, read, mmap}}\text{, mmap, open, read, mmap}$$

$$\text{open, }\underbrace{\text{read, mmap, mmap}}\text{, open, read, mmap}$$

$$\text{open, read, }\underbrace{\text{mmap, mmap, open}}\text{, read, mmap}$$

$$\text{open, read, mmap, }\underbrace{\text{mmap, open, read}}\text{, mmap}$$

$$\text{open, read, mmap, mmap, }\underbrace{\text{open, read, mmap}}$$

These five sequences found by a window size of $k = 3$ results in the following four unique sequences:

open, read, mmap
read, mmap, mmap
mmap, mmap, open
mmap, open, read

These unique sequences are saved in the database as trees with each root node being a specific system call; see figure 4.1 on the next page. This saves spaces and makes it more efficient to look up the unique sequences when monitoring the program after the training period.

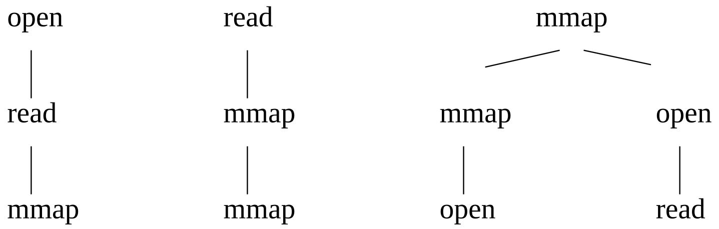


Figure 4.1: The unique sequences of system calls are saved as trees in the database.

4.1.2 Detecting Abnormal Behaviour

To detect abnormalities every sequence of length k generate by the program is checked with the database. If a sequence of length k is not found in the database it is considered to be a mismatch. The number of mismatches occurring in a new trace with respect to the length of the trace is an expression of how abnormal the trace is.

Generally they would like every abnormal trace to indicate an intrusion, but tests show that even new normal traces can cause mismatches because they simply did not occur during the training phase. The mismatch caused by normal traces often occurs during some kind of system problem, like running out of disk space, or because the executed program has caused an error resulting in an unusual execution sequence. To cope with this problem the researchers use Hamming Distance to compute how much a new sequence differs from a normal one. The reason for using Hamming Distance is that abnormal sequences due to intrusions often comes in local burst, and with Hamming Distance they are able to compute how closely these abnormalities are clumped [7, p.10], resulting in a better detection of burst caused by intrusions and lesser false positives.

4.1.3 Results

The application has been tested on three different UNIX programs: *sendmail*, *lpr*, and *wu.ftpd*. The *sendmail* program sends and receives mail, the *lpr* program prints to a printer, and the *ftpd* program transfer files between machines. To generate normal profiles for each of three programs, they exercised *sendmail* with 112 different messages, *lpr* with different kinds of existing and non-existing files together with symbolic links, and *ftpd* using all options from the man pages. The programs used to generate the traces were all non-patched programs vulnerable to already known successful intrusions.

To detect intrusions, attacks were carried out on the three non-patched UNIX programs. To summarise, the application were able to detect all abnormal behaviour caused by the attacks, including successful intrusions, failed intrusions, and unusual error conditions [7, p.18].

4.1.4 Analogy to the Biological Immune System

The analogy between this application and the biological immune system is the mechanism of distinguishing between self and nonself. The system learns over a period of time what is normal behaviour, and through change detection techniques it is able detect the abnormal behaviour. Furthermore, features like diversity, dynamic learning, and adaption found in the biological immune system are also seen in this application. The database build from the normal traces are unique with respect to others databases build on others machines. This results in a dissimilarity between every application running on different machines, making it harder to use the same kind of intrusion approach on every machine.

For more information on UNM's computer immune systems using system calls we refer to [7–10].

4.2 Network Intrusion Detections

The Network Intrusion Detection (NID) system developed by UNM is designed to detect network attacks by analysing and monitoring the network traffic. The system developed for detecting the intrusions is also named LISYS, which stands for Ligthweigth Intrusion detection SYStem.

Normally NID applications use signatures to detect the intrusions, the signatures are extracted from known attacks by human experts and are constantly being added to the NID's database as updates. The NID systems use the signatures to detect possible intrusions and will stop the intrusions by rejection or refusing the malicious network traffic. Other NID systems also detect abnormal behaviour using statistical analysis.

The basic idea in LISYS is to train the system on normal network traffic, known as self, trough a period of time. After training the system will be able detect all kind of abnormal behaviour, known as nonself, which the system was not exposed to during the training period. Instead of rejecting the network traffic as other NID systems does, LISYS will notify a human operator which will decide whether or not the abnormal behaviour is an intrusion attempt, the human operator will from here on decide which kind of action to take against the abnormal network traffic.

The structure of LISYS is built from another system called ARTIS, ARTificial Immune System, also designed by the researchers from UNM. The ARTIS system models most of the components and techniques from the biological immune system and could by used in a lot of different applications. We will therefore shortly describe ARTIS in a separate section before describing the LISYS system.

4.2.1 ARTIS

In ARTIS the detectors of the lymphocytes are modelled as binary strings, the detector contains the state of the lymphocyte which is either immature, mature, or memory. The detector also contains an indication of whether it is activated or not, when it last got activated, and how many matches it has accumulated. Finally, the detector contains a randomly generated binary string which represent the regions on the lymphocytes that binds to foreign substances.

ARTIS models the distributed environment of the biological immune system by placing the detectors in a graph with vertices. Each vertex can contain several detectors and the detectors can migrate from one vertex to another. To model the local environment of the biological immune system each detector is only able to interact with the other detectors in the same vertex.

To match with foreign substances they use a matching rule called *r-contiguous-bits*, which simply determines the maximum number of contiguous bits that two bit sequences have in common. The matching rule is described more in detail in section 5.7.2 on page 55.

To train the system on normal behaviour, they randomly generate the receptors so these could either bind to self or nonself. Afterwards the receptors are exposed to self. Those receptors which binds (match) to the set of self are killed in a process known as negative selection. The receptors that survive this process will change their state from being immature to mature and are now only able to bind (match) with nonself.

To model the surviving stimulation from the local environment each detector has a probability p_{death} of dying once it has reached the matured state. If the receptor dies it is replaced by a new randomly generated receptor, which again will be exposed to the set of self and undergo the process of negative selection. Generally all the receptors which dies are replaced by new randomly generated receptors, this enables the system to keep a constant rate of receptors just like the biological immune system which has a constant rate of lymphocytes.

When the matured receptor match with a foreign substance its match number is increased by some value, and when the match value exceeds an activation threshold the receptor becomes activated. When activated, the receptor has the possibility of advancing into a memory detector, but to keep the number of memory detectors constant, only a small fraction of the activated detectors will become memory detectors, all others will die. If the maximum number of memory detectors are reached the last recently activated memory detector will be replaced by the new one. In this way the ARTIS system is able to model the feature known as immunological memory from the biological immune system.

When the mature detector gets activated it will notify the human operator of the system to indicate that it has detected some abnormal behaviour. The human operator is now able to take the appropriate action against the abnormal behaviour and can eliminate the threat which caused the detectors to get activated.

4.2.2 LISYS

In the LISYS system the researchers have only decided to monitor TCP/IP network traffic, the sets of self and nonself are therefore represented by binary strings containing information on the TCP/IP network traffic. The representation of the TCP/IP traffic will only contain information about the network connection and not actual data. The representation of a network connection consist of a single 49-bit string containing source IP address, destination IP address, and TCP service or port.

Self is then defined as the set of network connections which are normally observed in the training period, and nonself is defined as the set of connections that are not normally observed in the network traffic. To model the distributed environment given as a graph with vertices in ARTIS, each vertex corresponds to a computer in a LAN, and the network connecting the computers corresponds to the graph. The local environment is modelled by a detector node that holds several detectors at the same time. It is assumed that the network is broadcast such that every computer sees every packet passing through the network.

To summarise LISYS implements most of the components and techniques given in ARTIS and use network connections to represent the set of self and nonself.

4.2.3 Results

LISYS was tested using data collected from 50 computers at the Computer Science Department of the New Mexico University. Through a period of 50 days over 2.3 million TCP/IP connections was collected, resulting in 3900 unique connections representing the self set. The system was then trained on the self set and afterwards exposed to seven different intrusions attempts. The intrusion attempts consisted of IP address probing and different kinds of port scanning. To summarise the system was able to detect all seven intrusions and the average of false positives generated by the system over 20 days was 1.76 pr. day, which is quite good for a NID system [11, p.18].

For more information on UNM's ARTIS and LISYS system we refer to [11] and [12]. For more general information about UNM's computer immune systems we refer to [13–19].

Chapter 5

Modelling a Computer Immune System

When modelling a computer immune system we take reference in the immune system of the human body. We try to imitate some of its components and mechanisms, hoping to build a system which will be able to make computer systems more secure and robust.

A computer immune system could be used for a lot of different kinds of applications when making computer system more secure and robust. The system could for instance be used in virus detection, network intrusion detections and in other kinds of change detection systems. We will therefore try to focus on modelling the system on such a high abstraction level that different kinds of applications could be built from the design.

We will in this chapter describe how the most important components and mechanisms of the immune system could be modelled in a computer. This chapter is by no means an instruction in how computer immune systems should be designed, because it only suggest how the most important components and techniques could be modelled. When designing and building a computer immune system not all components and techniques of the biological immune system are useful, because some elements might even be used in a completely different way than in the biological immune system. The goal is often not to model the complete biological immune system, but rather making smarter and better applications with inspiration from the biological immune system.

The chapter starts out by looking at the different layer of defence systems that the immune system consist of. After this we discuss some limitations and lightly specify what kind of functionality there is needed in such a system. Then we go more into detail with some of the components of the immune system, here we discuss how we could represent the huge amount of cells, model the circulation of the cells, and what *self* and *nonself* is in context to a computer. Finally we discuss the recognition done by the lymphocytes, different kinds of loose matching, elements needed to model lymphocytes, and how stimulation from the environment could be modelled in a computer system.

5.1 A Layered Defence System

One of the main characteristics of the immune system of the human body is that it is built from different layers of defence mechanisms. Each layer is specialised to handle different kinds of intrusions. The outer layer will filter of the most obvious intrusion attempts, whereas the inner most layer will take care of the most toughest and advanced intrusions.

The first defence layer of the human immune system, the skin, will only allow elements with the right size and format to go through. Only particles small enough and having the right structure will be absorbed and let in by the skin. Big particles with the wrong kind of structure will not be absorbed by the skin and are therefore denied or filtered out by the outermost layer. The second layer of defence, the physical conditions like pH-value and temperature, specify some conditions which are normal for the system; a temperature of 37 °C and a pH-value between 7.35 and 7.45 are normal for the human body. In the same way, we might be able find some conditions, which are similar or normal for the elements of our computer system, enabling us to deny access to those who have not got this kind of similarity or normality. The third defence system, the innate immune system, inherited from our parents, will filter out all known bad elements. This is a kind of knowledge that has been built up over a long time, experienced by others and is now given to us, enabling us to protect us better against known bad elements. The fourth defence layer, the adaptive immune system, has the ability to distinguish between self and nonself. In other words, it is able to learn what is good and what is bad. It is able to adapt to its environment, which makes it unique from other systems, and are able to remember previous encountered bad elements, enabling it to strike harder and faster the next time the bad elements are met. Figure 5.1 on the next page illustrates the four different defence layers.

If we were to design an immune system for virus detection, the outer most layer could for example check for the right format of incoming emails, files and so on. Does the email contain any attached files, and do we recognise this as any known format? Maybe the length of an attached file is specified in the email header, and does this length correspond to the real length of the file? In the same way we could check files on the disk, maybe some files have mysteriously changed their size, indicating a possible virus infection. Take for instance static executable programs, these seldom change size unless someone or something is tampering with the program. The same applies for the boot sector of a disk, which also seldom change size. The second layer could represent some kind of conditions set up by the user of the system. Maybe the user is only interested in

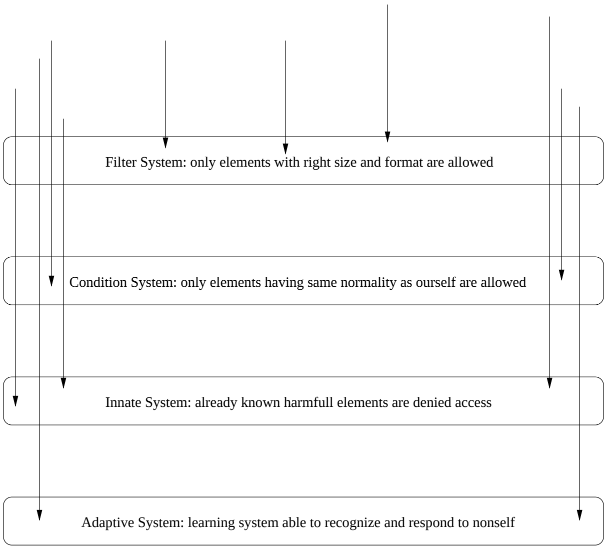

Filter System: only elements with right size and format are allowed
Condition System: only elements having same normality as ourself are allowed
Innate System: already known harmfull elements are denied access
Adaptive System: learning system able to recognize and respond to nonself

Figure 5.1: The immune system consist of four different defence layers, all carrying out their own specific job.

receiving files and emails from a certain number of trusted people, which the user normally communicate with. Or the user could for example set up an condition like: only receive emails with attached files from a certain amount of trusted people, enabling the user not to receive possible virus infected files attached to emails from unknown people. The third defence layer could for instance be a database with already known viruses. This is like most of todays virus scanners, which use virus definitions to find already known viruses in emails, files, and boot sectors. A virus definition is a kind of fingerprint of the virus, describing the virus in a short or compressed form. This could for instance be a certain block of instruction code, which is unique for the virus, and could always be found in files that the virus has infected. The last line of the defence is the adaptive system. Through a training period we teach the system not to recognise self, enabling the system to respond to nonself. Self is in this case all the normal emails, files, programs and so on, not infected with a virus. The system is now able to recognise everything which is not part of self, because we have taught the system to do so. If a virus then infects a program on the disk, the system will see this, because the program has now been changed and is no longer part of self. We will discuss the training, the distinguishing between self and nonself and matching later in this chapter.

If we instead where to make a immune system for network intrusion detecting, the first layer could filter of all network packets with the wrong kind of format of size. Maybe we are only interested in TCP/IP packets and not UDP packets, and maybe we know the sizes of network packets we would like to receive, enabling us to deny all wrong size packets. The second defence layer could deny all packets from known harmful net, subnet and hosts, or only receive packets from hosts which we are expecting to communicate with. The third layer could be a database of known network intrusions, describing how the packets of already known intrusion normally look like. The last defence system, the adaptive, would be taught in a training period not to recognise self packets, thereby enabling the system to recognise and respond to unknown packets and mysterious behaviour in network activity.

5.2 Limitations

A computer has only a limited amount of resources and trying to simulate billions of cells interacting with each other, carrying out their own little task, seems to be quite impossible on one of todays computers. We need to design a system which is able to work within the limited amount of resources in a computer. In other words: the system needs to be small.

The system we are designing also needs to be effective. Making a computer system more secure is not always at the highest priority, serving clients and carrying out different kinds of tasks might have a higher priority. What good is it, to have a system which is highly secure, but not capable of carrying out any tasks because all resources are used to make the system more secure?

Furthermore, it needs to be precise and correct. Who will use a system which gives false alarms and is not capable of reporting potential intrusions. It is okay for the human immune system to kill a single healthy cell and make small errors, but if a file is deleted due to the computer immune system erroneously reporting, the consequence could be catastrophic. Many of the files in a computer system are normally very important and have specific purposes for the system or the user, whereas the cells of the body are more error tolerant. It is therefore strictly necessary for the system to be precise and correct when reporting alarms to the user. In other words the system must have a small rate of false negatives and false positives.

In the design of the computer immune system we will mostly try to simulate and adopt the mechanisms of the adaptive immune system. This is because, it is the adaptive immune system, which has all the nice properties that we are interested in. It is the adaptive immune system which enables us to learn, remember and recognise new infectious agents and has the properties of being unique, highly distributed, dynamic, adaptable, no central control and so on. We will especially look at the lymphocytes of the adaptive immune system, because these are the main players of the adaptive immune system and carry out most the mechanisms, which give us the above mentioned properties.

5.3 Functionality

The main functionality of the system is to decide whether some kind of input data is normal or not. The input data could be of different kinds, e.g. files, emails or network packets. The system will read the input date and carry out some kind of action if it seems to be abnormal or suspect. The action taken from the system could be anything from just reporting the abnormality to completely removing or denying the input data from the computer. Figure 5.2 shows the overall design of such a system.

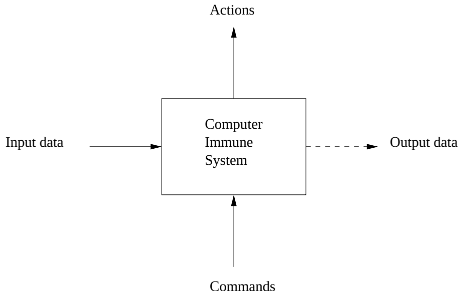


Figure 5.2: The overall design of the computer immune system. The action carried out by the system could be anything from reporting an abnormality to deleting the input data. The commands could for example be parameter adjusting or change of system operation.

The system could be implemented as a network filter, reading all the incoming data from the network before sending it on to the system, assuring that all data from the network is okay. The input data would then be network packets, and the output data a subset of the incoming packets where all abnormal packets were removed. The system could also just be implemented as to read the input data from other places, such as a hard disk with a lot of files. If the system was to be used in detecting viruses, an analysis of how data flows through the computer system might be good idea, especially incoming data from diskettes, CD-Roms and emails, where viruses normally spread from, should be used as input data to the system.

As seen on figure 5.2 we also need some way of controlling or telling the system how to operate. This could for example be commands like ignoring or thrusting input data or adjusting some kind of parameters for the system. We might need to tell the system that the incoming data could be trusted as normal or is safe in some way. This functionality is for example needed to simulate a trusted environment, like the thymus or the bone marrow in the immune system of the human body, where elements are exposed to self in teaching them to recognise nonself. The adjusting of parameters could for example be setting the number of active components or the size of some kind of threshold.

5.4 Representing the Huge Amount of Cells

When designing the system we choose only to look at the adaptive immune system as described in section 5.2 on page 44. Furthermore, we only choose to try to simulate the working of the lymphocytes, because these are the main players of the adaptive immune system.

Every lymphocyte carries out their own little task communicating locally with the environment and goes through different kinds of states, enabling them to carry out different kinds of actions. There are millions of lymphocytes present at all the time in the body acting on their own, making the immune system highly distributed. The most obvious way to represent this kind of highly distributed behaviour, would be by several processes or threads running at the same time, each representing a lymphocyte. Clearly this will demand a huge amount of resources, if we were to have millions of processes or threads running at the same time. To handle this problem we could either (1) distribute the total number of lymphocytes onto several machines, (2) lower the total number of lymphocytes, enabling us to run on one machine, or (3) represent the lymphocytes in another way than processes or threads, also enabling us to run on one machine; see figure 5.3.

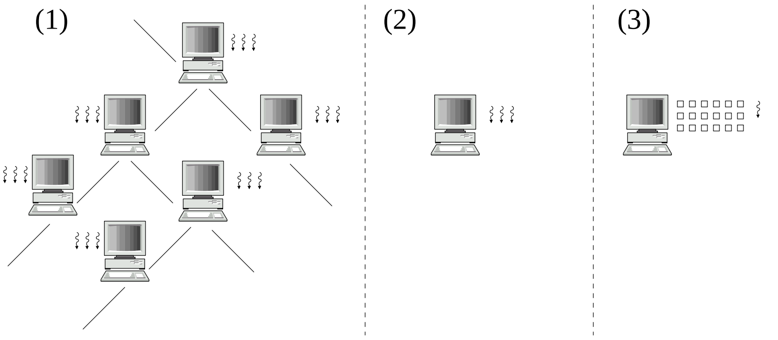


Figure 5.3: Three different ways of handling the huge number of highly distributed lymphocytes: (1) distribute onto several machines, (2) lower total number, or (3) use another representation.

Even if we choose to represent the lymphocytes by another way than processes and threads, we will still need a huge amount of memory on one machine to simulate millions of them. Furthermore, we also need some way of making the lymphocytes act on their own, to achieve the highly distributed system that we are interested in. Though, this problem could be solved by implementing a scheduler, which in turn would choose a lymphocyte to carry out its task.

Each of the three solutions in figure 5.3, of representation the huge amount of lymphocytes, has advantages and disadvantages. Clearly the first solution needs several machines and some kind of communication protocol, which enables the lymphocytes to interact over the network, but then we also have the possibility of simulating all the lymphocytes. The second solution is bound to the number of processes or threads that we are allowed to run on one machine, the small number of lymphocytes may affect the performance of our system, resulting in high rate of false negatives, because we are not able to recognise all the infectious agents. The third solution will properly allow us to have more lymphocytes than the second solution, but we are still limited by the memory available on one computer and will also have to simulate the working of the lymphocytes by implementing some kind of scheduler. Solution two and three have the advantages, that they are able to run on one machine only. Surely, a mix of all three solutions could be used as well.

5.5 Circulation of Cells in the Body

The lymphocytes of the immune system are constantly circulating in the body, enabling the lymphocytes to meet and interact with each other. In some way we should try to simulate this kind of behaviour, to achieve a more error tolerant system with no central control and to induce the autonomous system with local interacting that we are interested in. We need to simulate that the lymphocytes are located at specific points, that they can move to other nearby locations and that they are only able to communicate with the local environment. One way of doing this, is to positioning them in a limited two dimensional space with a x and y value representing their location; see figure 5.4.

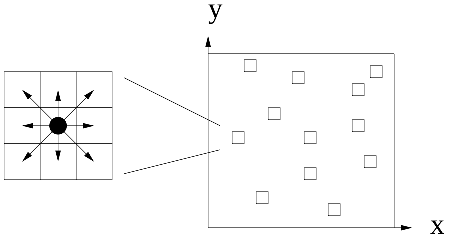


Figure 5.4: Placement of lymphocytes in a limited two dimensional space. Each lymphocyte is only able to interact with the surrounding environment.

To simulate the interaction with the local environment, we could restrict each lymphocyte only to communicate with adjacent cells as depicted to the left in figure 5.4. If the system were to be distributed onto several machines, each machine could have their own limited two dimensional space, and only the lymphocytes near the boundary of the space were allowed to travel to the nearby machines.

To extend the model further, we could of course also just use a three dimensional space, to represent the placement of the lymphocytes, enabling the lymphocytes to travel into 26 different directions, instead of just 8 as depicted to the left in figure 5.4 on the page before. If a three dimensional space is used, the interacting with other cells would also increase, because the lymphocyte would now have up to 26 adjacent cells to interact with.

Quite another way of representing the placement of the lymphocytes, is to use a graph with vertices and edges. Each vertex would either be empty or contain a lymphocyte. To simulate the local environment, every vertex must be connected to one or more vertices, enabling the lymphocytes to travel to other vertices or interact with other cells located at the connected vertices; see figure 5.5.

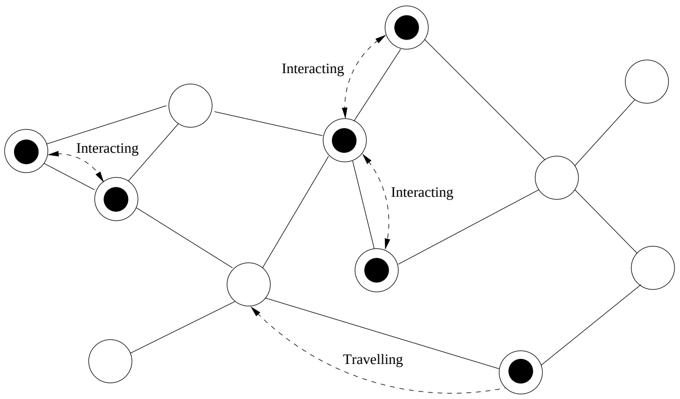


Figure 5.5: A graph with vertices and edges is used to simulate the lymphocytes placement.

When using a graph to represent the placement of the cells, we are not limited by an upper amount of nearby cells to interact with, as with the two or three dimensional space described above. But the memory needed to represent the graph will be quite huge compared to the two or three dimensional space. In the worst case, if the graph is dense we will need in the order of $|V|^2$ memory units, where V is the set of vertices in the graph. If the graph instead is sparse we only need in the order of $|E|$ memory units, where the E is the set of edges. The two or three dimensional space model will only need memory units for representing the boundary of the space, e.g. the upper and lower limit for the x, y and z directions. Another disadvantage with the graph model, is that we need to generate the graph in some way, and what should the requirements for such a graph be: how many edges should every vertex have, should the number of edges for every vertex be different or the same and how do we decide which vertices are connected to which?

5.6 Self and Nonself

When designing our immune system we need to consider how to train the lymphocytes not to recognise self. But what is self, how do we represent it, are there any problems with this kind of representation and could the set of self be compromised in some way, making our system less robust and more fragile?

Seen from a lymphocyte's point of view there are good and bad elements in the body. The good elements are referred to as self whereas the bad elements are referred to as nonself. Just like a human child, the lymphocytes need to be taught what is correct behaviour and what is not. If a child is taught by a bad or ignorant parent it might damage the character of the child, making it unable to distinguish between right or wrong. In the same way the lymphocytes need to be taught correctly to respond to right and wrong. The teaching of lymphocytes is carried out in the thymus and in the bone marrow of the human body where they are exposed to self peptides. Through processes known as negative selection and positive selection, see figure 2.1 on page 24, all the lymphocytes which respond strongly to self peptides are killed, thereby ensuring that the surviving lymphocytes will not react on and kill what is part of self.

If we look at all the self peptides of the human body, which the lymphocytes are exposed to in the thymus or in the bone marrow, we can define them as the set of self denoted S. The set S is a subset of the universe denoted U, $S \subseteq U$, and the nonself set denoted N, is then defined as the complement to S, as $N = \overline{S} = U - S$. Normally the set S is much smaller than the set N, because the body only has a certain amount of self peptides, whereas the set N is defined as everything else than self. Figure 5.6 shows the sets of self and nonself.

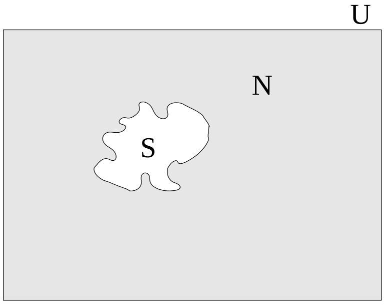


Figure 5.6: The nonself set N is defined as the complement to the self set S.

The literature on immuno biology [1] do not state the size of self or nonself, just that 10^8 specificities of the nonself set are represented by lymphocytes in the body at any time and that experiments with young adult mice have shown that 10^6 new T-lymphocytes leave the thymus every day. So, an indication of how big the size of self and nonself should be in a computer immune system, if we tried to simulate the immune system of the human body completely, seems to be unclear. Furthermore, given the limited amount of resources that a computer system has, see the discussing in section 5.4 on page 46, we would properly not be able to represent all of the immune system's cells. The only thing we can see from the literature [1], is that it seems like there is a connection between the size of the self and the nonself sets and the number of lymphocytes circulating the body, so we might keep that in mind when deciding upon the number of lymphocytes and the size of the self and nonself sets.

So, how should the set of self be represented? Well, the incoming data to the system could consist of files, emails, network packets or other kinds of data. These could be represented by streams of bits and we could therefore also represent the self and nonself sets by streams of bits. To make things easier and more effective on the computer, we might prefer to represent the bit streams by byte or word sequences because the computer is designed to work with byte and word aligned blocks of bits. But how long the bit streams should be, what they should contain and if they are allowed to be of different length is very application specific. And before deciding upon such a matter, an analysis of the application for which the computer immune system is going to be used for, must be carried out. If we for instance were going to make a system for network intrusion detection, we would take a closer look at the incoming network packets from the network. What kind of information is it that we are interested in? The sending host, the receiving host, the time, the type of packet, maybe which kind of port is used, is it a broadcast packets and so on. These informations could be extracted from each incoming packets and put together to represent an element in our universe U. If we were to make a computer immune system for virus detection we could for instance make an analysis of how virus infects programs, is it in the beginning of the program, in the end of the program, only in the code segment of the program and not in the data segment, maybe we could make an flow diagram of the program and track the behaviour of the program instead of extracting information from the static program code and so on. Again these information could be put together into a bit stream representing an element in our universe U. This kind of extracting and decomposing information into smaller fragments is in a way also carried out in the immune system of the human body. Some of the cells in the immune system of the human body are able to decompose the infectious agents into peptide fragments and represent them to the T-lymphocytes by MHC molecules on their cell surface. The T-lymphocytes of the human body are in this way taught to recognise small nonself peptide fragments instead of bigger ones, making the recognition faster and very simple.

Another issue that we should address when discussing self and nonself, is that the definition on self in the human body quite seldom change, whereas we in a computer immune system properly would like to add new harmless data to the system. Doctors have for several years tried to solve this problem in the human body, when transplanting animal organs into human bodies, by holding the immune system down to stop the body from rejecting the transplant. Another way for the doctors to stop the body from rejecting the transplant is by giving the patient cells from the bone marrow of the transplant provider, and in this way try to change the definition of self in the human body, hoping that newly created cells of the immune system will not attack the new transplant. The problem with changing the definition of self, is that there are still a lot of cells circulating the body, which are still able to recognise the new part of self as nonself, resulting in an autoimmune response. Furthermore, by allowing a redefinition of self in our computer immune system, we might introduce a possible security hole, because the unattended user might include harmful data into the definition of self, disabling the newly created lymphocytes from recognising the harmful data. Clearly the most secure solution is not to allow for a redefinition of self, and thereby assuring that no harmful data will ever be a part of self. But if we allow for a redefinition of self, we need to figure out a way to stop the already created lymphocytes from recognising the new part of self. We could design our system to make the lymphocytes return to the place where they were exposed to self, or in some way let the definition of self come to the lymphocytes, enabling us to kill the lymphocytes responding to the new part of self. As explained in section 2 on page 23 the lymphocytes are repeatedly receiving stimulation from the environment, to assure that the total number of lymphocytes are constant and that the receptors on lymphocytes are working. Through this stimulation we could also exposed the lymphocytes to the new part of self, such that the stimulation now have three purposes: keeping the total number of lymphocytes in the system constant, assuring that the receptors of the lymphocytes work, and killing the lymphocytes responding to the new part of self. As mentioned before this opens a major possible security hole in the system, and the user really needs to be sure that new data included in self is not harmful in any way. Figure 5.7 on the next page shows how the immune system of the human body really works, whereas figure 5.8 on the following page illustrates a new kind of system, where new data is allowed to be included in the set of self.

Another way of simulating the immune system of the body, is to have a more static system, where the system first goes through a training period and then afterwards is exposed to new incoming data. This kind of system could especially be used in network intrusion detection, where a fixed number of randomly generated lymphocytes are exposed to normal harmless network packets over a training period and then afterwards will monitor all new incoming network packets to detect any kind of abnormal behaviour in the network traffic.

1. A fixed number of lymphocytes are randomly generated.
2. Over a training period the lymphocytes are exposed to normal harmless data (self), and the lymphocytes recognising any of these data will be eliminated in the process of negative selection.
3. After the training period, the lymphocytes will be set to monitor all incoming data.
4. If any incoming data is recognised, is must be harmful data (nonself) and some kind of action is taken.

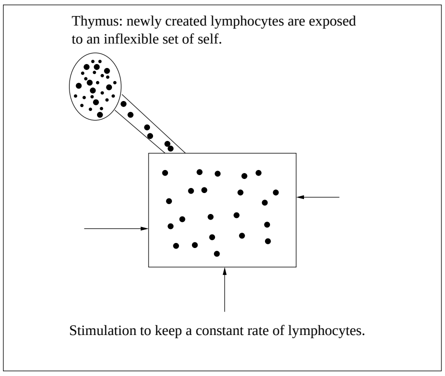

Thymus: newly created lymphocytes are exposed
to an inflexible set of self.
Stimulation to keep a constant rate of lymphocytes.

Figure 5.7: The lymphocytes are repeatedly generated, exposed to an inflexible set of self in a controlled environment and released to circulated the system. The lymphocytes are kept at a constant rate by stimulation from the local environment.

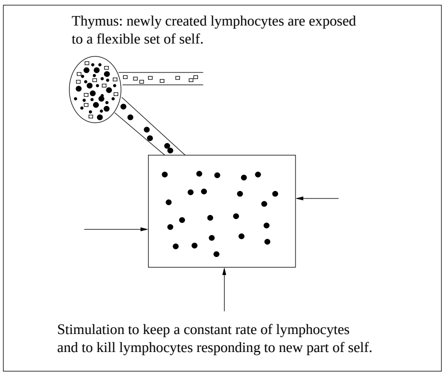

Thymus: newly created lymphocytes are exposed
to a flexible set of self.
Stimulation to keep a constant rate of lymphocytes
and to kill lymphocytes responding to new part of self.

Figure 5.8: The lymphocytes are repeatedly generated, exposed to a flexible set of self in a controlled environment and released to circulated the system. The lymphocytes are kept at a constant rate and lymphocytes responding to new part of self are killed by stimulation from the local environment.

This kind of system is more static, because no newly generated lymphocytes will be added to the system once the training period is over. When a suspect network packet is recognised, the system could notify an operator, which could then decide whether it is an intrusion attempt or just a false positive. If it is a false positive, the operator could decide to remove the lymphocyte from the system, to prevent getting more false positive of the same kind. By removing the lymphocyte from the system, the set of self is in a way automatically redefined, because this type of network packets is properly not going to be recognised by the system any more. The process of having an extra verification, from some kind of operator, before engaging an immune response, is also seen in the immune system of the human body, where the B-lymphocyte needs extra verifications from the T-lymphocytes, before the B-lymphocyte will engage the immune response.

5.7 Matching

When the lymphocytes in the human immune system recognise an infectious agent, its is due to binding of peptides to the lymphocyte's receptors. The B-lymphocyte will bind to peptides displayed on the infectious agents cell surface, whereas the T-lymphocyte will bind to peptides displayed by MHC molecules on the cell's surface. The binding of the peptides to the lymphocyte's receptors is not a perfect match, but more like a loose match. The term known as affinity describes the strength of the binding between a peptide and a receptor. The stronger the binding between the peptide and the receptor is, the stronger a signal is send to the core of the lymphocyte. When enough receptors are bound, the signals to the core of the lymphocyte are so strong, that the lymphocyte will start to differentiate – the lymphocyte has now recognised an infectious agent.

How should we simulate this kind of loose matching and the strength of the match, the affinity? What are we going to match, and what are our requirements to the matching algorithms?

The input data could be decomposed into small fragments, or in some way be compressed, to speed up the matching algorithm. We could compress the data by using some kind of hash function or checksum representing the original data, or we could extract only the important parts of the input data if we knew its format and structure, e.g. with network packets, where we for instance only are interested in the header. The incoming data therefore needs to go through some kind of filter as illustrated in figure 5.9 on the next page to make it easier for the matching algorithm to work.

Whether we are going to match compressed, extracted or decomposed data, it will in the end just be sequences of bits, bytes or words. So, when discussing the matching algorithms in this section we will just assume that we are going to match sequences of *discrete symbols*, which could be anything like bits, bytes, chars, colours and so on. The output of the filter will be objects with references to the original bit stream, and symbol sequences, which we are going to match with. A single bit stream may lead to several objects when going through the filter, depending on the format and the structure of the incoming data and the application for which we are going to use the immune computer system for.

Through our search for finding different ways of doing the loose matching, we have found three quite different approaches: Hamming Distance, R-Contiguous Symbols, and Hidden Markov Models. We will in the following three subsections introduce these approaches and discuss their complexity.

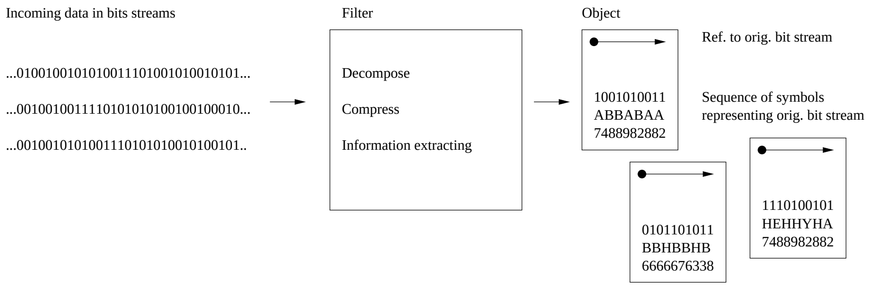


Figure 5.9: The incoming data will be decomposed, compressed or information extracted, before matching in the system will be carried out. The filter output objects with symbols sequences representing the original bit stream and a reference to the original bit stream.

5.7.1 Hamming Distance

The Hamming Distance is the number of places in which two sequences of symbols with equal length differ, or in other words, the number of symbols, that are needed to be changed in order to make two sequences alike. E.g. given two sequences of symbols, A and B, over the alphabet $\{0, 1\}$:

$$\begin{array}{rcl} A & = & 0110101010 \\ B & = & 01\underline{0101}1010 \end{array}$$

we need to change 0101 in sequence B to make the two sequences alike, therefore the Hamming Distance of A and B is four.

A more sophisticated way of Hamming Distance, is the Levenshtein Distance, also known as Edit Distance, able to match two sequences with different length. Instead of just counting the places where the two sequences differ, it also count the number places where one of the sequences has a symbol and the other does not. Take for example the two sequences:

$$\begin{array}{rcl} A & = & 0110101010 \\ B & = & 1000 \end{array}$$

Here the Levenshtein distance is six because we need to insert six symbols into the B sequence before the two sequences are alike:

$$\begin{array}{rcl} A & = & 0110101010 \\ B & = & \;\;1\;\;0\;\;0\;\;0 \end{array}$$

The Hamming Distance takes in the order of T comparisons giving an upper asymptotic running time of $\mathcal{O}(T)$, where T is the length of one of the two sequences. Whereas the algorithm for finding the Levenshtein Distance has an upper asymptotic running time of $\mathcal{O}(TS)$, where T is the length of the first sequence and S the length of the second sequence. If we are going to make a system where all the sequences have the same length, clearly the Hamming Distance should be used.

To find the affinity of the match when using Hamming Distance or Levenshtein Distance, we simply divide the distance with the length of the longest sequence and subtract this from 1. For the Hamming Distance we define the affinity of the match as:

$$P_{ma} \quad = \quad 1 - d/T,$$

where d is the Hamming Distance and T is the length of one of the sequences. For the Levenshtein Distance we define the affinity of the match as:

$$P_{ma} \quad = \quad 1 - d/\max(T, S),$$

where d is the Levenshtein Distance and T and S are the length of the two sequences.

5.7.2 R-Contiguous Symbols

The second matching approach, R-Contiguous Symbols, is the number of places where contiguous symbols of two sequences are the same. Two sequences, A and B, will match with r contiguous symbols, if r contiguous symbols in A are also found in B. Lets take an example:

$$\begin{array}{rcl} A & = & \underline{011010}1010 \\ B & = & 0101\underline{011010} \end{array}$$

Here the sequences A and B are defined over the alphabet $\{0, 1\}$ and match with the six contiguous symbols: 011010.

The approach could also be extended to match two symbol sequences of different length. Here we would just slide the smallest sequence of symbols, one symbol at the time, below the bigger symbol sequence to find the maximum number of r contiguous symbols. If for instance we have the two sequence $A = 10011100101$ and $B = 10101$, we would find the maximum number of r contiguous symbols as given in table 5.1 on the following page

From table 5.1 on the next page, we can see that A and B match with four contiguous symbols.

If the T is the length of the longest sequence and S is the length of the smallest sequence we need to shift the smallest symbol sequence T times and for every time we must compare S symbols, resulting in a upper asymptotic running time of $\mathcal{O}(TS)$. If the two symbol sequences are of the same length, the running time would be $\mathcal{O}(T^2)$.

match	r	match	r	match	r
10011100101 $\underline{10}101$	2	10011100101 $101\underline{01}$	2	10011100101 $\underline{101}01$	3
10011100101 $1\underline{01}01$	2	10011100101 $\underline{10}101$	2	10011100101 10101	0
10011100101 $10\underline{1}01$	1	10011100101 $1\underline{0101}$	4	10011100101 $\underline{1}0101$	1
10011100101 $10\underline{10}1$	2	10011100101 10101	0		

Table 5.1: Finding the maximum number of r contiguous symbols from the two sequences 10011100101 and 10101.

To find the affinity of the match, we simply divide r by the length of the longest symbol sequence:

$$P_{ma} \quad = \quad r/\max(T, S).$$

If the two symbol sequences are of the same length, we simply use:

$$P_{ma} \quad = \quad r/T.$$

5.7.3 Hidden Markov Models

Using a Hidden Markov Model (HMM) to do the loose matching is quite a different approach than the Hamming Distance and the R-Contiguous Symbols method. A HMM is a state machine where all the state transitions have fixed probabilities, and every state in the model have a probability distribution for observing the symbols in the state concerned. Figure 5.10 on the facing page illustrates a HMM where the two symbols A and B can be observed in each of the states S_1, S_2, and S_3.

With a HMM we are able to express that some state transitions are more likely to happen than others, for instance in the HMM given in figure 5.10 on the next page we can see that it is more likely to make a state transition from state S_1 to S_2 (3/5) than it is to make a state transition from state S_1 to S_3 (1/5). Furthermore, with the HMM we are also able to express how likely it is to observe different kinds of symbols in each state, from figure 5.10 on the facing page we can for instance see that it is more likely to observe symbol B (3/5) than it is to observe symbol A (2/5) in state S_1.

HMMs are often used in the area of speech recognition and biological sequence analysis searching for patterns in e.g. DNA. Normally the HMM is randomly generated and then trained or re-estimated to maximise the probability of observing a certain sequence of symbols, the trained HMM is then used to match with other symbol sequences to figure out how well these match with the original one. The training of the HMM with the original symbol sequence takes in

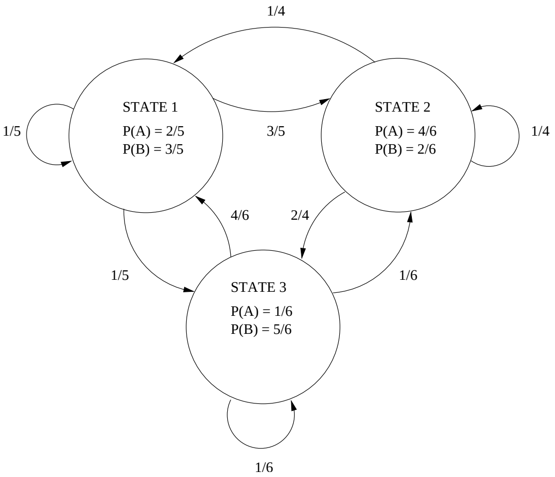


Figure 5.10: A Hidden Markov Model with 3 states and the symbols A and B.

the order $\mathcal{O}(N^2T)$ for every re-estimation of the model, where N denotes the number of states in the model and T the length of the original symbol sequence. The match with other symbol sequences each take in the order of $\mathcal{O}(N^2T_i)$, where T_i is the length of the i'th symbol sequence.

To figure out how well an observed sequence of symbols $O = O_1O_2 \cdots O_T$ matches with a specific HMM, an algorithm called Forward-Backward is used. The algorithm computes a forward value, which is the probability of observing the sequence $O = O_1O_2 \cdots O_t$ and being in state S_i at time t.

step 1: Initialisation.

$$\alpha_1(i) = \pi_i b_i(O_1), \qquad 1 \leq i \leq N \tag{5.1}$$

step 2: Induction.

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i)a_{ij}\right] b_j(O_{t+1}), \qquad 1 \leq j \leq N$$
$$1 \leq t \leq T-1. \tag{5.2}$$

Given the forward value $\alpha_T(i)$ for every state S_i, we can now calculate the probability of observing the sequence O in the HMM λ as:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i). \tag{5.3}$$

While the Forward-Backward algorithm computes the probability of observing the sequence $O = O_1O_2 \cdots O_t$ and being in state S_i at time t, another algorithm called Backward-Forward can compute the probability of having observed the sequence $O_{t+1}O_{t+2}\cdots O_T$ given we are in state S_i at time t:

step 1: Initialisation.

$$\beta_T(i) = 1, \qquad 1 \leq i \leq N \tag{5.4}$$

step 2: Induction.

$$\begin{aligned}\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1})\beta_{t+1}(j), \qquad & 1 \leq i \leq N \\ & t = T-1, T-2, \cdots, 1.\end{aligned} \tag{5.5}$$

From the alpha and beta values, we are able to compute the probability of being in state S_i at time t given the observation sequence $O = O_1O_2\cdots O_T$ and the model λ:

$$P(q_t = S_i|O,\lambda) \quad = \quad \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)}, \tag{5.6}$$

this probability is denoted $\gamma_t(i)$. We are also able to compute the joint probability of being in state S_i at time t and in state S_j at time $t+1$ given the observation sequence $O = O_1O_2\cdots O_T$ and the model λ:

$$P(q_t = S_i, q_{t+1} = S_j|O,\lambda) \quad = \quad \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)}, \tag{5.7}$$

this probability is denoted $\xi_t(i,j)$.

To train a HMM for a symbol sequence O we use an algorithm called Baum-Welch, it iteratively re-estimate the parameters of the HMM such that the probability of observing the sequence O in the HMM is maximised:

$$\overline{\pi}_i \quad = \quad \gamma_1(i) \tag{5.8}$$

$$\overline{a}_{ij} \quad = \quad \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\gamma_t(i)} \tag{5.9}$$

$$\overline{b}_j(k) \quad = \quad \frac{\sum_{\substack{t=1 \\ O_t=v_k}}^{T}\gamma_t(j)}{\sum_{t=1}^{T}\gamma_t(j)}. \tag{5.10}$$

Here $\overline{\pi}_i$ denotes the re-estimated initial probability of observing the symbol O_1 in state S_i, $\overline{a}_{ij}$ denotes the re-estimated probability of making a transition from state S_i to state S_j, and $\overline{b}_j(k)$ denotes the re-estimate probability of observing the symbol v_k in state S_j where v_k is the k'th symbol in the set of all symbols.

When training a HMM for a certain symbol sequence the probabilities in the HMM are maximised to observe exactly that symbol sequence. And by calculating the probability of observing any other symbol sequences in the trained

HMM, we can get an estimate of how well any of these other sequences match with the sequence that the HMM is trained for. If for instance a HMM is denoted λ and trained for the observation sequence A, we can find the affinity of the match between A and any other sequence B as:

$$P_{ma} = \frac{P(B|\lambda)}{P(A|\lambda)}$$

Normally the probabilities computed by Forward-Backward and Backward-Forward algorithms are to small to be represented within the boundary of double precision values in a computer. The values are therefore scaled using slightly modified versions of the Forward-Backward and Backward-Forward algorithms. Furthermore, the probability of observing a sequence in the HMM is often very small, so logarithmic probabilities are used instead. When using log probabilities instead of normal ones the affinity of the match between sequence A and B could then be computed as:

$$\log[P_{ma}] = \log[P(B|\lambda)] - \log[P(A|\lambda)].$$

A full and more in depth introduction to Hidden Markov Models is given in appendix C on page 107. Here we also discuss the techniques of using scaled algorithms to make sure that the probabilities can be represented within the boundary of double precision values.

5.7.4 Comparing Matching Approaches

In table 5.2 we have tabulated the running times for each of the three different matching approaches with respect to equal and different lengths of the symbol sequences.

Matching approach	Equal length	Different length
Hamming Distance	$\mathcal{O}(T)$	$\mathcal{O}(TS)$
R-Contiguous Symbols	$\mathcal{O}(T^2)$	$\mathcal{O}(TS)$
Hidden Markov Models	$\mathcal{O}(N^2T)$	$\mathcal{O}(N^2T)$

Table 5.2: Running times for three different matching approaches.

In the first two rows of the table 5.2 T denotes the length of the biggest symbol sequence whereas S denotes the length of the smallest sequence. In the third row T is the length of the symbol sequence that we are going to match the HMM with and N is the number of states in the HMM. Furthermore, the running time listed in third row does not take into account the time for training the HMM.

Clearly if we were going to make a system where all the symbol sequences where of equal lengths, the Hamming Distance would be preferred as it only has a running time in the order of $\mathcal{O}(T)$. If the symbols sequences where of different lengths and the number of the states in the HMMs could be kept at a low rate the HMMs seem to be the best approach.

We should keep in mind that this comparison only looks at how *fast* the matching could be carried out, and not how *good* it is. So when choosing the right matching approach for the system, we should also consider how good it is. To figure this out, we might do some tests with the three different approaches, look at experience made by others in similar kinds of applications, or simply choose the most intuitively correct one.

5.8 Lymphocytes

We have been looking at different approaches to simulate the circulating of lymphocytes in the system and the handling of the large amount of lymphocytes, but how should we actually represent the lymphocytes in our system, and what kind of functionality is needed?

Through our description of the immune system of the human body, we have learnt that the lymphocytes should be able to communicate with other cells, this could for instance be cells presenting an infectious agent for the lymphocytes or simply other cells trying to stimulate the lymphocytes in some way. The stimulation from other cells is needed by the lymphocytes to keep alive, and to ensure the functionality of the lymphocyte's receptors. Furthermore we discussed in section 5.6 on page 49, that it would be nice, if we in some way were able to expose the already mature lymphocytes to a new part of self, enabling us to change to definitions of self and nonself while the system was running.

We also discussed the different kinds of states that the lymphocyte could reside in during its life time: Created, Immature, Mature, Activated, Memory and Dead. How the receptors of the lymphocytes was randomly generated from gene rearrangements, exposed to self peptides and either died or survived through the processes of negative and positive selection. Then released to circulate the body, getting activated if 10-100 of its receptors where bound to foreign peptides, dividing itself into thousand of daughter cells to increase efficiency and release chemical signals to extract other cells to the site of infection. The best of the daughter cells, would then be chosen to memory cells, giving the body an ever lasting memory of the infectious agent and through affinity maturation the memory cell will be able to strike even harder and faster if the previous encountered infectious agent were met again. The affinity maturation will give the memory cells the ability to better match the infectious agent, resulting in a lower number of receptors needed to be bound before getting activated and engage the immune response.

When representing the lymphocytes in our system, we focus on what kind of elements there are needed to carry out the described functionality. As a minimum we should be able to represent the following:

- The lymphocyte's state.
- The specificity of the lymphocyte's receptor: a representation of the randomly generated receptors that we are going to match the infectious agents with.
- A value representing the numbers of receptors currently bound or a value representing the accumulated signal send to the core of the lymphocyte from the currently bound receptors.
- A value representing the threshold before getting activated.

The two main functionalities of the lymphocyte is to either travel or interact. The travelling is rather simple, but when interacting the lymphocyte should at least be able to carry out the following functionalities:

- Receive stimulation, this could for instance be commands like:
  - die when being exposed to part of self to simulate negative selection,
  - or die because we need to keep a constant rate of lymphocytes in the system.
- Go through different states.
- Recognise infectious agents.
- Divide itself to simulate clonal expansion.
- Lower activation threshold to simulate a possible faster activation when being a memory cell.
- Carry out some kind of affinity maturation.
- Release signals to attract others to the site of infection.
- Travel in a specific direction due to the receiving of signals from other cells.
- Carry out some kind of action when being activated.

5.9 Stimulation

To keep alive the lymphocytes repeatedly need stimulation from the local environment. This enables the immune system to have a kind distributed local control and enables it to keep the numbers of lymphocytes in the body at a constant rate. Furthermore, the lymphocytes can be stimulated by chemical signals like cytokines and chemokines to attract them to the site of infection. But how should we model the stimulation from the local environment in our system?

The stimulation could be modelled in a number of ways. One approach is to stimulate the lymphocytes from the environment they reside in. If a graph is used to represent the location of the lymphocytes, vertices in the graph could contain information on how many lymphocytes there were allowed to stay in the vertex, and for how long time they were allowed to stay. Furthermore, the vertex could contain a probability measure indicating how likely it would be for the lymphocyte to survive in that particular vertex, thereby enabling the system to keep a constant rate of lymphocytes in each vertex. A second approach is to model some kind of *stimulation* cells also circulating the system together with the lymphocytes. These cells would be able to interact with the lymphocytes and could keep a constant rate of them by neglecting stimulation to those which had been inactive for a long time and not doing any good. To attract lymphocytes to the site of infection the cells could also carry information on which direction the lymphocytes should travel to help in fighting an intrusion. A third approach is to try to let the lymphocytes handle all the stimulation themselves. The lymphocytes would have a probability measure of how likely it would be for them to survive when being in a specific state. As an example newly matured lymphocytes often have a high frequency of dying, whereas the memory lymphocytes often lives forever to support the immune system with the feature known as immunological memory.

Chapter 6

Computer Immune System for Virus Detection

With reference in the computer immune systems designed and implemented by IBM and UNM, we will in this chapter discuss how a system for virus detection could be designed. Components and mechanisms from the immune system, as discussed in the previous chapter, will be used to the extent that we see fit; we will not try to make a complete model of the immune system, but only use the parts that we find useful.

We start out by explaining how we have chosen to represent *self* instead of *nonself* in our computer immune system. As explained in the previous chapters we regard self as the elements which are not harmful to our system whereas nonself is regarded as elements which are harmful. Then we describe what kind of matching approach we have chosen to detect nonself and finally we discuss a static and a dynamic method for detecting nonself.

6.1 Representing Self instead of Nonself

The main purpose of the system is to distinguish between self and nonself. The immune system handles this job by randomly generating millions of cells and afterwards killing those cells reacting to self. In this way the cells of the immune system hold a complete set of nonself, and all substances which match this set are regarded as nonself and are therefore harmful. The following list shows the steps in the biological immune system for detecting nonself:

1. Randomly generate receptors.
2. Kill those receptors reacting to self.
3. Anything that match the receptors must be nonself.

In a computer system the set of nonself might be so large that we are not able to represent it within a computer, and the only possibility is therefore to try to represent the set of self instead. In such a system we do not randomly generate anything, but simply match with the set of self. The system then detects nonself by any substances which do not match self and in this way the system is able to tell what is right from wrong without having to represent the complete set of nonself. The list below shows the steps for detecting nonself in our system:

1. Match with self.
2. If it does not match with self it must be nonself.

But what is self and nonself in our system? Well, we have chosen to detect viruses from their patterns of behaviour. This means that self represents the normal behaviour which is not harmful and that nonself represents the abnormal behaviour which is harmful. Generally the computer immune system will hold a representation of normal behaviour and anything that deviates from this will be regarded as nonself.

6.2 Matching Approach

In section 5.7 on page 53 we discussed three different matching approaches: Hamming Distance, R-Contiguous Symbols, and Hidden Markov Models (HMMs). The Hamming Distance was used by the researchers from UNM to detect abnormal behaviour in sequences of system calls. The Hamming Distance gave them an expression for how closely the abnormalities in the system call sequences were clumped, resulting in fewer false positives. In the ARTIS and LISYS systems the researchers from UNM used the R-Contiguous Symbols approach to find abnormal behaviour in network connections. The argument for using this approach instead of Hamming Distance was that it more accurately simulated the detection of nonself in the biological immune system.

When matching in our system we focus on using HMMs. There are several good reasons for this. A HMM represents the behaviour of the sequence it is trained for, and by matching other sequences in the HMM we get an expression for how much these sequences deviate from the original behaviour. In this way we are able to detect if some sequences change behaviour or acts abnormal in some way. A second good reason for using HMMs is that we can train a HMM for multiple sequences and the HMM will then represent the average behaviour of these sequences. This means that we can represent the full set of self in one single HMM if we like. A third good reason for using HMMs is that we can match with sequences of different length and the execution time for doing this is faster than with Hamming Distance or R-Contiguous Symbols.

6.3 Static Analysis

The static analysis is designed to detect abnormal behaviour in programs due to virus infections but could easily be extended to also detect boot sector viruses and macro viruses found in word and excel documents. We call it a static analysis because we analyse the complete static code of a program.

The basic idea is to built a normal behaviour profile for all the normal and uninfected programs and then afterwards monitor the programs to see if any programs deviate from this profile. The normal behaviour profile is built by training one or several HMMs on the uninfected programs and the probability of observing the uninfected programs in the HMMs is saved. To detect abnormal behaviour the probability of observing the programs in the trained HMMs is computed and if any of these deviate from the saved probabilities we know that a program has changed behaviour. The general procedure looks like this:

Training:

1. Train HMM λ for program O.
2. Extract probability of observing program O in λ: $P(O|\lambda)$.
3. Save λ and $P(O|\lambda)$ in database.

Detecting:

1. Compute probability of observing the possibly infected program $\acute{O}$ in λ: $P(\acute{O}|\lambda)$.
2. If $P(\acute{O}|\lambda) \neq P(O|\lambda)$ the behaviour of the program has changed.

6.3.1 Building a Normal Behaviour Profile

To represent the normal behaviour of one or several programs we use HMMs. We can do this because HMMs say something about the behaviour of the sequences they are trained for. The beautiful feature of HMMs is that they can be trained for any sequences of symbols without having any knowledge about what the sequences represent and still be able to say something about the behaviour of the sequences.

The most obvious way to make a normal behaviour profile for a program is simply to train the HMM on the complete binary code of the program. In this way all information about the program will be represented in the HMM. But the bottle neck with using the complete binary code of the program to train the HMMs is that it takes a lot of time. So to make the system more efficient it might be a good idea to train the HMM on a representation of the program instead of the complete binary code of the program.

The binary code of a program often consists of a header, different kinds of identification information, and then code and data segments. What we are interested in here is the behaviour of the program, mostly reflected by the code segments but also by the data segments in the program. The code segments contain instructions which can be executed and the data segments contain data like constant values, character strings, data areas, screen images, etc. We can therefore choose to represent a program by its code and data segments, and still have the complete behaviour of the program. We could also go even further and only represent the program by its code segments, making an even smaller representation of the program and still representing most of the behaviour of the program.

Quite another approach to represent the programs in the system is by compressing them using already known compression algorithms. We could for instance use compression algorithms like zip or gzip. The only problem with this approach is that the compression algorithms might change the complete structure of the program in such a way that it is not possible for the HMMs to detect abnormal behaviour. The compression algorithms might rearrange some sequences or blocks of instructions in such a way that the behaviour of the program is completely destroyed.

When training the HMMs to represent the behaviour of uninfected programs several approaches could be chosen: Train one HMM for every program in the system, divide all the programs in the system into categories and train one HMM for the programs in every category, or simply train one HMM for all the programs in the system.

Training one HMM for every Program

Training a HMM for one program will iteratively maximise the probability of observing exactly that program in the HMM. The training iteratively makes the probability of observing the program in the HMM better and better until the probability of observing the program finally converges. When the training is over we will have a HMM representing the behaviour of the program and can compute the probability of how well the program matches the HMM. Off course we would like this probability to be as close to 1 as possible, but often this is not the case and we therefore have to save the value so we can compare it with subsequent probabilities of observing the possible infected program in the HMM. The procedure for training one HMM for every program looks like this:

Training:

1. Train HMM λ for program O.
2. Extract probability of observing program O in λ: $P(O|\lambda)$.
3. Save λ and $P(O|\lambda)$ in database.

The HMM λ will together with the probability $P(O|\lambda)$ make up the normal behaviour profile for the program O.

Training one HMM for several Programs

Training a HMM for several programs will iteratively maximise the average probability of observing each of the programs in the HMM. After training we should extract the probabilities of observing each of the programs in the HMM, so we can compared these values with subsequent observations of possible infected programs. The procedure for training several programs for one HMM look like this:

Training:

1. Train HMM λ for the programs $O^{(1)}, O^{(2)}, \cdots, O^{(n)}$.
2. Compute probability of observing $O^{(1)}, O^{(2)}, \cdots, O^{(n)}$ in λ: $P(O^{(1)}|\lambda), P(O^{(2)}|\lambda), \cdots, P(O^{(n)}|\lambda)$.
3. Save λ and $P(O^{(1)}|\lambda), P(O^{(2)}|\lambda), \cdots, P(O^{(n)}|\lambda)$ in database.

The HMM λ will together with the probabilities $P(O^{(i)}|\lambda)$ make up the normal behaviour profile for the ith program $O^{(i)}$.

6.3.2 Detecting Abnormal Behaviour

To check whether or not a program is infected with virus, we try to detect if the static code of the program has changed behaviour in some way. We find the HMM saved in the database which represents the normal behaviour of the program and compute the probability of observing the program in the HMM. If the computed probability has change from the probability saved in the normal behaviour profile we know that the program has changed behaviour. The procedure for detecting abnormal behaviour looks like this:

Detecting:

1. Compute probability of observing the possible infected program $\acute{O}$ in λ: $P(\acute{O}|\lambda)$.
2. If $P(\acute{O}|\lambda) \neq P(O|\lambda)$ the behaviour of the program has changed.

The procedure for detecting abnormal behaviour is the same whether or not the HMM is trained for one program or several programs.

An often arising question is how often we should check whether or not the program has changed its behaviour. Should this be done periodically or randomly or should we choose a completely different approach? Well, before a program can change its behaviour the binary code of the program must be changed in some way. The only way to change a program is by writing code to it and the most normal way of doing this, and in some systems the only way, is by using the operating system's `write` call. So by intercepting the operating system's `write` call on a program, we know that the program could in theory now contain a virus. In other words whenever a program has been written to, we check to see if it has changed its behaviour indicating a possible virus infection. If it is not possible to intercept the system calls or if the operating system allows writing to executable files without using the `write` call, we need to periodically test the programs for changed behaviour.

When a program is tested for abnormal behaviour we compute the probability of observing the program in the HMM and compare it with the probability saved in the normal behaviour profile as mentioned before. To get an expression of how well the program matches with the normal behaviour we can calculate the *match affinity*:

$$P_{ma} = \frac{P(\acute{O}|\lambda)}{P(O|\lambda)},$$

where $\acute{O}$ is the possible infected program, O the original program, and λ is the HMM trained for the original program.

In some systems the static behaviour of a program might be allowed to change a little bit, so in these system it is not enough simply to see if the probability has changed from the probability saved in the normal behaviour profile. Under these circumstances the match affinity could be used to find an expression for the size of the changed behaviour. This value could simply be found as: $1 - P_{ma}$.

6.3.3 Discussion

The bad thing about the static analysis if using periodical tests on programs is that the virus might be able to infect several files before we are able to notice it. Another thing that the static analysis do not take into account is the installation of new programs on the system, this method assumes that we have a *static* system, and no new programs are added after the training has been carried out. It is possible to re-train a HMM to also include a new program, but before doing this one should really be sure that the new program is not infected with a virus!

6.4 Dynamic Analysis

In the dynamic analysis we scan traces of system calls to detect abnormal behaviour. This approach enables us to detect the virus while it is trying to execute itself and stop it before it actually infects other programs.

The basic idea is to detect deviations from normal behaving traces of system calls, just like the researchers from UNM did in their application for intrusion detection as explained in section 4.1 on page 35. The big difference here is that we are going to use HMMs to detect the abnormal behaviour instead of Hamming Distance and that the traces of system calls are not split into sequences. We do not split the traces into sequences because we believe that it removes some of the information about how the programs actually behave.

There are two approaches to do the dynamic analysis. The first approach is to try to simulate how the programs are going to be executed in a real environment. Each program is executed with all possible parameters and the traces of systems calls generated during this period are used to train the normal behaviour profile for the program. After the training period the normal behaviour profile is used to detect any abnormality whenever the program is executed. The second approach is to train the normal behaviour profile from normal traces of system calls generated by a program when it is actually executed by the users of the system. The training period is not determined by a certain number of traces as with the first approach. Here we simply keep on training the normal behaviour profile until it has settled down. The first couple of traces will probably deviate a lot from each other resulting in some huge deflections, but after a period we believe that the normal behaviour profile will be settled down and now represents the user's normal use of the program and thereby the normal behaviour of the program.

6.4.1 Learning from Synthetic Behaviour

We will in this subsection discuss how we can build a normal behaviour profile for a program by trying to simulate how the program would be executed under normal circumstances. Furthermore we describe how we detect any deviations from this normal behaviour profile.

Through a training period the program is executed with all kinds of different parameters and the traces of system calls generated are used to train a HMM representing the normal behaviour of the program. The average probability of observing normal behaving traces in the HMM is computed and saved together with the HMM. The average probability measure together with the HMM will represent the normal behaviour profile for the program. After the training period is done we are now able to detect any kind of abnormal behaviour by testing subsequent traces of the program in the HMM. This is done by obtaining the trace of system calls generated by the program, computing the probability of observing the trace in the HMM, and comparing this with the average probability measure saved in the normal behaviour profile. The procedure for training and detecting looks like this:

Training:

1. Execute program with all different kinds of parameters and obtain the traces of system calls generated: $O^{(1)}, O^{(2)}, \cdots, O^{(n)}$.
2. Train HMM λ on these traces.
3. Compute the average probability of observing the traces in λ: $P_{average} = \frac{\sum_{i=1}^{n} P(O^{(i)}|\lambda)}{n}$.
4. Save λ and $P_{average}$ in database.

Detecting:

1. Obtain the traces of system calls $\acute{O}$ whenever the program is executed.

2. Compute probability of observing $\acute{O}$ in λ: $P(\acute{O}|\lambda)$.
3. Compute the match affinity: $P_{ma} = \frac{P(\acute{O}|\lambda)}{P_{average}}$.
4. If P_{ma} is lees than some threshold the trace is thought of as being abnormal.

To be really sure that the detected abnormal behaviour is due to a virus infection, and not just some rare program execution trace that did not occur during the training period, we can accumulate subsequent abnormalities. If repeated abnormalities are detected within the traces of the same program, it is very likely that the program is infected with a virus, because once a program has been infected, all subsequent traces will reflect the infection of the virus. If a single rather small abnormality occurs, it is properly due to a program execution which was not simulated during the training period, in this case we should not mark the program as being infected with a virus. In other words repeated large abnormalities indicate a virus infection whereas small single occurring abnormalities indicate normal behaviour not simulated during the training period.

The technique of accumulating repeated abnormalities is also known from the biological immune system. Here 10-100 receptors of the lymphocytes need to match nonself before they will be activated and engage an immune response to kill the infection. As the receptors of the lymphocyte are bound, signals are send to the core of the lymphocyte, the stronger the binding is, the stronger a signal is sent, and the less signals are needed to activate the lymphocyte. In other words the lymphocytes have some kind of activation threshold; the signals sent to the core of the lymphocytes are accumulated and once they reach the activation threshold the lymphocytes are activated to engage the immune response.

6.4.2 Learning from Real Behaviour

In the above section we trained the normal behaviour profiles from traces generated by simulated program executions with all different kinds of parameters. In this way we simulated how the programs were normally going to be executed in a real environment. Quite another approach is to train the normal behaviour profiles from real executions of programs made in a real environment.

The first time a program is executed the trace of system calls generated by the program is used to train a HMM representing the normal behaviour of the program. All subsequent traces of the program are then used to re-train the HMM until it seems to be settled down. In the beginning many different traces will appear, but after some time all normal traces have occurred and the HMM will then represent the normal behaviour of the program.

Training:

1. Train HMM λ for the first trace $O^{(1)}$ and initialise the average probability $P_{average}$ to $P(O^{(1)}|\lambda)$.
2. Compute the probability of observing any subsequent trace $O^{(i)}$ in λ: $P(O^{(i)}|\lambda)$.

3. If $P(O^{(i)}|\lambda)$ differs a lot from $P_{average}$ then re-train λ to also include $O^{(i)}$ and set $P_{average} = \sum_{j=1}^{i} \frac{P(O^{(j)}|\lambda)}{i}$.
4. If the probability of the subsequent traces does not seem to differ a lot from $P_{average}$ the HMM has converged and we can stop training.
5. Save λ and $P_{average}$ in database.

The detection of abnormal behaviour is the same as described in section 6.4.1 on page 69.

A really big problem with learning from real behaviour is that we are not working in a controlled environment, this means that a virus might infect the program while we are training the HMM. We therefore have to monitor the environment very carefully while training to ensure that no virus infected traces are use to train the HMM. To do this we could train the HMM in an isolated environment where no virus is able to infect the programs. This means that the machine would not be allowed to be connected to the network and no new data would be allowed from CD-Roms, diskettes, and other peripheral devices. The disadvantage with this approach is that the normal behaviour profiles might be incomplete, especially for those programs which are used in connexion with the network and for those programs that use peripheral devices.

6.4.3 Discussion

The good thing about the dynamic analysis is that we are able to detect the virus relatively fast while it is executing its viral code. This means that we are able to detect the virus before it replicates itself to a lot of programs and stop it from carrying out disastrous actions corrupting the system. With todays systems a virus is often able to replicate itself to a lot of files and make a lot of damage before the user of the system notices it. When the user notices that something is wrong in the system, he will start a virus scanner to detect and remove the virus. Often this is too late because the virus has already replicated itself to several files and caused a lot of damage. Furthermore, the anti virus products are sometimes not even able to repair the programs and the user must re-install them on the system and in worst cases the virus have deleted files that are unable to retrieve. With the dynamic analysis we can build an automated system which is able to detect the virus while it is executing and have the possibility of stopping it before it spreads and makes further damage.

6.5 Conclusion

We have in this chapter described how we are going to represent self instead of nonself. We have chosen this approach because it is often easier to represent self in a computer system simply because the self set is much smaller than the nonself set. We also discussed which kind of matching approach we were going to use in our computer immune system. We chose the HMMs because they could represent the normal behaviour of sequences without having any knowledge about how to interpret the sequences. This means that we could use HMMs to represent the normal behaviour of a program from its static code, or from system calls generated by it, or any other kind of data representing the program. Furthermore, we indicated that HMMs could be trained on several sequences with different length, and that the execution time for matching with other sequences of different length was faster than any of the other proposed matching approaches.

Then we discussed some different ways of training the normal behaviour for one or several programs and how we could detect if any of the programs deviated from this normal behaviour. We discussed how we could train HMMs on a program's static behaviour by using parts of the program's binary code, and how we could train HMMs on a program's dynamic behaviour by using traces of system calls generated by the program.

But which of these approaches should we use in our computer immune system? Well, we do not know precisely, but in general the dynamic approach seems to be the best one, because we can detect the virus while it is trying to execute itself and before it infects other programs. As soon as any little trace of system calls deviates from the normal behaviour, we are able to detect it, because we can collect the traces while the program is executing. Furthermore, when training the HMMs on the traces of systems calls we believe it should be done in a real environment. In this way the computer immune system would be fully automated, the user do not have to train the HMMs on any simulated traces of programs executed with different kinds of parameters, and the HMMs would represent the true normal behaviour of the programs and not some kind of simulated behaviour that is never going to take place. In this way all computer immune systems would be unique if we trained them in the users own and real environments. If one virus found a way of not being detected in one system, it could not use the same approach in another system because all the systems are trained differently.

Chapter 7

Experimental Results

We will in this chapter describe and discuss some of the experimental results carried out on HMMs. A full and in depth introduction to HMMs can be found in appendix C on page 107. The software used for the experiments is implemented in Java and described further in appendix D on page 139.

Generally we use HMMs to build a normal behaviour profile for a program. The normal behaviour profile is used in detecting any abnormal changes made to the program. We train the HMMs in number of different ways, first we train a single HMM on the complete binary code of one program, second we use the complete binary code of several programs to train a single HMM, and last we use traces of system calls generated by one program to train a HMM. Two different kinds of changes are made to the files, first we try to randomly change a number of bytes on the program and second we try to infect the programs with a virus.

The programs used in these experiments are collected right after a new installation of windows 98 and have a maximum size of 30 KB. After collection, the sample programs are saved safely and the windows 98 installation is infected with a harmless virus known as *Apathy*. The *Apathy* virus is a file infector that works on windows 9x/NT systems and infects executable win32 files in the system directories. The virus will overwrite the original start of the executable file with a copy of itself and the original code will be copied to the end of the file. When an infected file is executed, the virus will copy itself to memory and start running as a separate process searching for other executable files to infect. Furthermore, the virus reconstructs the original code of the infected file in a temporary file and execute it as well, in this way the original code is not destroyed and the program will still be able to work correctly.

In the following sections we will use a term known as *log likelihood*. With the log likelihood we understand the 10 base logarithmic value of a probability measure. The term is often used in connexion with observing a sequence in a HMM. The probability of observing the sequence O in a HMM λ is given by $P(O|\lambda)$. The log likelihood of observing the sequence O in a HMM λ is given by $\log10[P(O|\lambda)]$.

As mentioned before we use the HMMs to represent the normal behaviour of programs. This means that we train a HMM on sequences representing the program's behaviour. In these experiments we use the complete binary code of a program and traces of system calls generated by the program when executed. Afterwards we test how well any sequences match with the normal behaviour. This is done by computing the log likelihood of observing a sequence in the HMM. As we shall see later on in this chapter, some sequences deviates so much from the normal behaviour that we can not represent the computed log likelihoods. This simply means that the computed log likelihoods of observing the sequences in the HMM are so small that we can not represent them within the boundary of double precision values in the computer.

To train a HMM for one or several sequences we use the re-estimation formulae implemented in the HMM software described in appendix D on page 139. The re-estimation formulae will iteratively maximise the log likelihoods of observing the sequences in HMM. Generally through all the experiments, the re-estimations of every HMM are carried out until a threshold of 10^{-3} is reached or 500 iterations have been carried out. A threshold of 10^{-3} simply means that we iteratively re-estimate until $log10[P(O|\lambda_{i+1})] - log10[P(O|\lambda_i)] < 10^{-3}$, where λ_i denotes the re-estimated HMM in the ith iteration and O denotes the byte sequence that we are training the HMM for. The threshold enables us to stop re-estimating when the log likelihood of observing the sequence in the HMM converges to a fixed value.

7.1 Randomly Made Changes in a Single Program

We will in this section describe how we trained 29 HMMs having from 1 to 29 states on a single program. We trained 29 HMMs with a different number of states to see how the number of states would effect the normal behaviour profile. All the experiments described in this section are carried out on a Pentium II 300 MHz machine running Linux Redhat with the Kaffe Virtual Machine 1.0.5 for executing java 1.1 byte-code.

The following three figures illustrate the training of the 29 HMMs for the xcopy program. The xcopy program from the windows 98 distribution is a 4KB program for copying files, directories and drives from one destination to another. Figure 7.1 on the next page illustrates the training of the 29 HMMs and shows how the log likelihood is increased for every iteration of the re-estimation of the HMMs. The lowest curve in the graph is the re-estimation of the HMM with 1 state, whereas the upper curve in the graph is the re-estimation of the HMM with 29 states. We can see how the log likelihood converges to a fixed value when a certain number of iterations has been reached.

Figure 7.2 on the facing page illustrates the log likelihood as a function over the number of states in the HMM. Here we can see how the log likelihood is

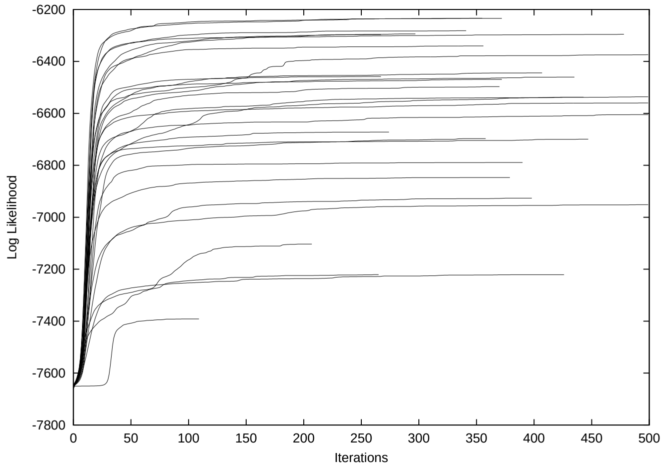


Figure 7.1: The log likelihood is increased at every iteration of the re-estimation formulae.

improved whenever the number of states are increased. We can also see the log likelihood seems to be converging as the number of states in the HMM are increased. This means that in the end it is not feasible to increase the number of states anymore because the log likelihood can not be improved any further.

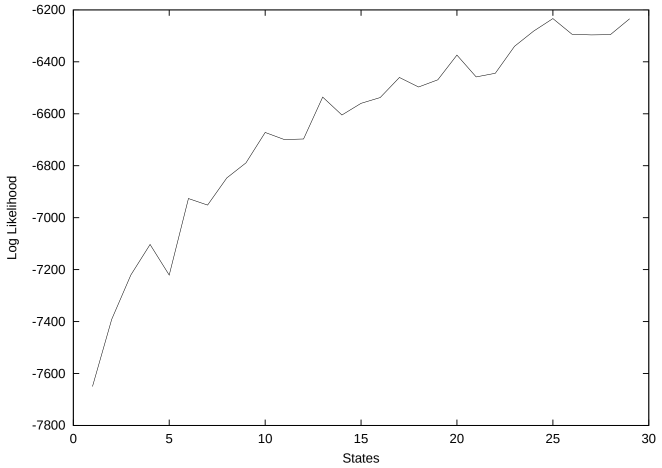


Figure 7.2: The log likelihood is improved when using HMMs with increasing number of states.

Finally figure 7.3 on the next page shows how long time it took to train the 29 HMMs on a Pentium II 300 MHz processor. As we can see the time is parabolic increasing with the number of states, which is in agreement with the stated training time of $\mathcal{O}(N^2T)$ as depicted in section 5.7.3 on page 56. The re-estimation of all the 29 HMMs took 10537 iterations, resulting in an average of 363 iterations for every HMM. As mentioned before the re-estimation is repeated until a threshold of 10^{-3} is reached or 500 iterations have been carried out, 5 out of the 29 HMMs stopped because they could not reach the threshold before 500 iterations.

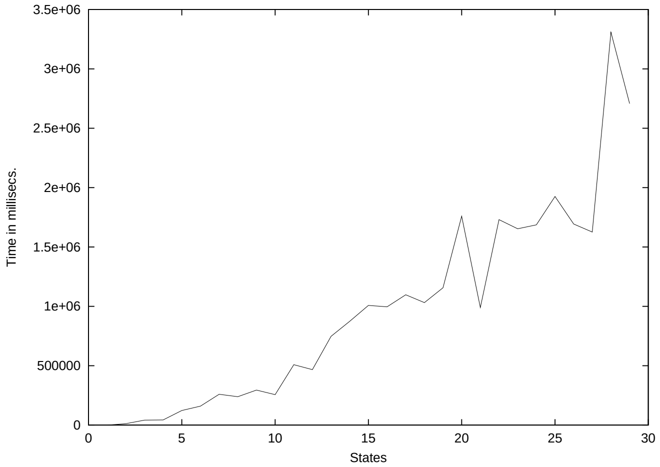


Figure 7.3: The time it took to train 29 HMMs having from 1 to 29 states on a Pentium II 300 MHz processor.

We can see from the three graphs that the log likelihoods of observing the xcopy program in the HMMs are improved whenever the number of states are increased, but we can also see that the time for training the HMMs is parabolic increasing by the number of states in the HMM. Deciding on the right number of states to use in the HMMs is therefore a compromise between the time available to re-estimate and how well the program should be observed in the HMM which it is trained for.

As mentioned in chapter 6 a normal behaviour profile for a program consist of a HMM, which is trained for the program, and the probability of observing the program in the HMM. So, in this experiment we have built 29 different normal behaviour profiles for the xcopy program. The 29 HMMs having from 1 to 29 states represent 29 different normal behaviours for the xcopy program and these together with the corresponding log likelihoods plotted in figure 7.2 on the page before represent the 29 different normal behaviour profiles. In other words, the HMM with 1 state and the log likelihood plotted in figure 7.2 on the preceding page at state number 1 having a value of -7650, represents the first normal behaviour profile for the xcopy program. The HMM with 2 states together with the log likelihood in figure 7.2 on the page before at state 2 having a value of -7391, represents the second normal behaviour profile for the xcopy program and so on.

To see how much randomly made changes to the xcopy program deviate from the normal behaviour profiles, we have randomly substituted 1, 5, 10 and 15 bytes in the xcopy program with random bytes. The four different xcopy programs with the changed bytes are then tested in the 29 HMMs by computing the probability of observing them in the HMMs. To figure out how much the four changed programs deviate from the original one we have computed the log match affinity, log10$[P_{ma}]$, which is simply the log likelihood difference between the changed programs and the original one:

$$\log10[P_{ma}] = \log10[P(\acute{O}|\lambda)] - \log10[P(O|\lambda)],$$

here $\acute{O}$ denotes the changed program and O the original program. Figure 7.4 shows the log likelihood differences between the four changed programs and the original xcopy program.

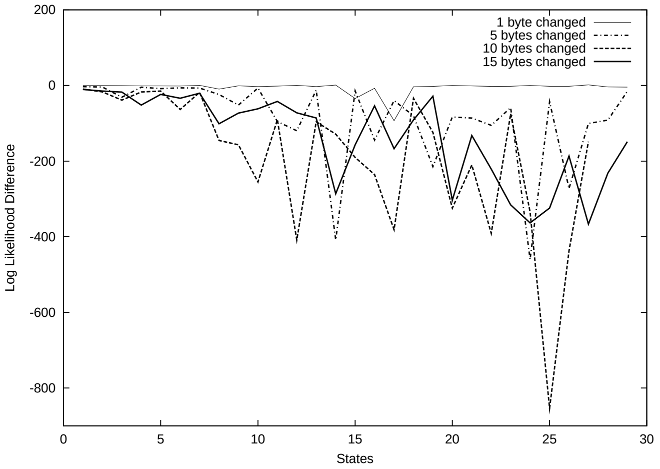


Figure 7.4: The log likelihood differences between four changed programs and the xcopy program. The changed programs were made by randomly substituting 1, 5, 10 and 15 bytes in the original xcopy program.

We can clearly see that the log likelihood difference gets bigger when the number of states in the HMMs increase, and that the programs with 5 or more substituted bytes differ more than the program with only 1 byte substituted. Furthermore, we can see that the HMMs are in fact quite good at detecting even small randomly made changes to the original programs and experiments made with programs having more than 15 randomly changed bytes deviated so much from the normal behaviour profile that we could not even compute the log likelihoods of observing them in the HMMs.

To get an impression of how long time it took to test one of the randomly changed programs in the HMMs, we plot the computation time as a function over the number of states; see figure 7.5. The test was made on a Pentium II 300 MHz machine running Redhat Linux and the plotted data is from the test with the program having 1 randomly substituted byte. As we can see from figure 7.5 the computing of the log likelihoods are parabolic increasing with the number of states. This is in agreement with the running time of observing a sequence in a HMM of $\mathcal{O}(N^2T)$ as describe in the section 5.7.3 on page 56.

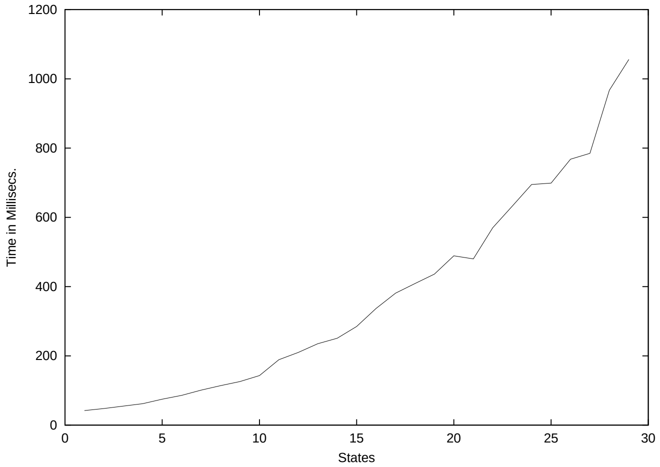


Figure 7.5: The computation time for computing the log likelihoods of observing a changed xcopy program in the 29 HMMs trained for the original xcopy program.

What we can conclude from these experiments is that HMMs are quite good at detecting even small randomly made changes in programs, but they say nothing about how well the HMMs are at detecting changes made by virus infections such as the replacement of complete code segments or adding of extra code to the program, we will look at this in the next section.

7.2 Detecting Viruses with HMMs Trained for Single Programs

To see how well the HMMs are at detecting virus infections, we have trained several HMMs on programs before infection. We start out by looking at an experiment we did on the ping program from the windows 98 distribution. The ping program is a 29KB large program for sending request packets to network hosts. The HMMs for the ping program was trained on a Pentium II 300 MHz machine with Redhat Linux running Kaffe Virtual Machine 1.0.5 for executing java 1.1 byte-code.

As in section 7.1 on page 74 we train 29 HMMs, having from 1 to 29 states.

We plot the log likelihoods computed from the training and the time it took; see figure 7.6 and 7.7. The 29 HMMs will together with the 29 computed log likelihoods plotted in figure 7.6 make up 29 different normal behaviour profiles for the ping program.

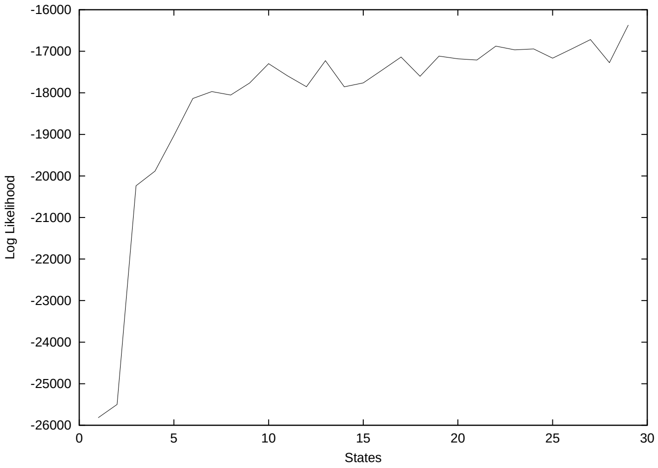


Figure 7.6: The log likelihood of training the ping program on 29 HMMs as a function over the number of states.

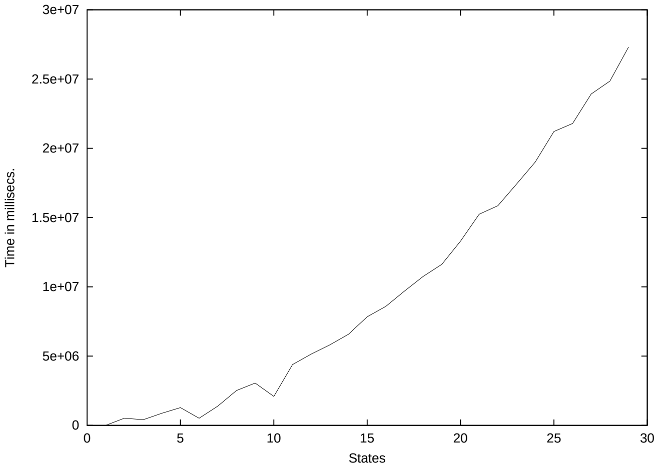


Figure 7.7: The time it took to train 29 HMMs having from 1 to 29 states with the ping program on a Pentium II 300 MHz machine.

We then try to see how much the ping program infected with the *Apathy* virus deviates from the 29 normal behaviour profiles by computing the log likelihoods of observing the infected program in the HMMs. Interesting enough the computed log likelihoods are so small that it is not even possible to represent them within the boundary of a double precision value. This means that the virus infected program deviates really much from the normal behaviour profile! Similar experiments were done with other programs like arp, write, and loadwc from the windows 98 distribution. These experiments show exactly the same result: the infected programs deviated so much from their normal behaviour profiles that the log likelihoods of observing the virus infected programs in the HMMs were simply to small to be represented within double precision values.

From these experiments we can conclude that HMMs trained on non-infected programs are able to detect when the programs have been infected with the *Apathy* virus. The virus infected programs deviated so much from their normal behaviour profile, that we could not even represent the computed log likelihood of observing them in the HMMs. Maybe this is not so surprising because the *Apathy* virus adds more than 5KB of virus code when infecting programs and this quite a big change for small programs.

7.3 Detecting Viruses with HMMs Trained for a Set of Programs

To see how good a HMM trained for a set of programs are at detecting virus in each separate program, we have trained a HMM with 5 different programs from the windows 98 distribution. The training was carried out on a Pentium III 733 MHz machine running Linux Redhat with the Kaffe Virtual Machine 1.0.5 for executing java 1.1 byte-code. We trained 24 HMMs having from 1 to 24 states resulting in 24 different normal behaviour profiles. The 5 programs used from the windows 98 distribution was: sage (12KB), mstinit (12KB), wmiexe (16KB), chlinst (16KB), and write.exe (20KB). The time it took to train the 5 programs in the 24 different HMMs is plotted in figure 7.8 on the facing page.

To see how well the 5 programs are observed in the HMMs which they have been trained for, we plot the computed log likelihoods of observing the programs in the 24 HMMs. We also plot the log likelihood of training the set of all the 5 programs which is computed during the training. This is simply the sum of observing each of the 5 program in the HMMs:

$$\begin{aligned}\log10[P(O^{(1)}, O^{(2)}, O^{(3)}, O^{(4)}, O^{(5)}|\lambda)] &= \log10[P(O^{(1)}|\lambda)] + \log10[P(O^{(2)}|\lambda)] + \\ & \log10[P(O^{(3)}|\lambda)] + \log10[P(O^{(4)}|\lambda)] + \\ & \log10[P(O^{(5)}|\lambda)]\end{aligned}$$

To see how much virus infected programs deviate from the normal behaviour profiles, we have infected the 5 programs with the *Apathy* virus. For each infected program we then compute the log likelihoods of observing it in the 24 HMMs representing the normal behaviour for the 5 programs. As it turns out,

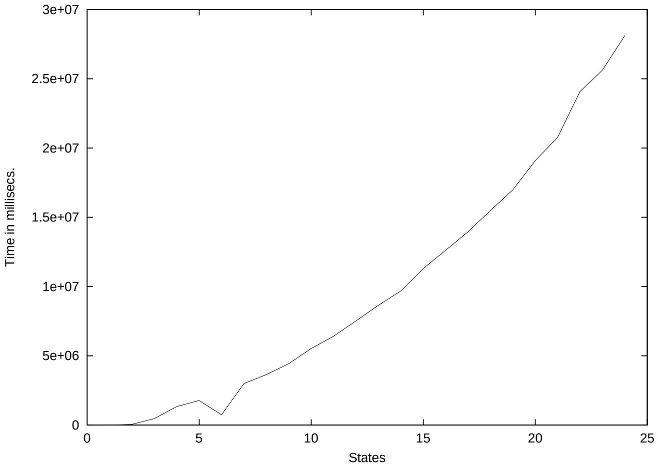


Figure 7.8: The time it took to train 24 different HMMs using a set of 5 programs. The training was carried out on a Pentium III 733 MHz machine.

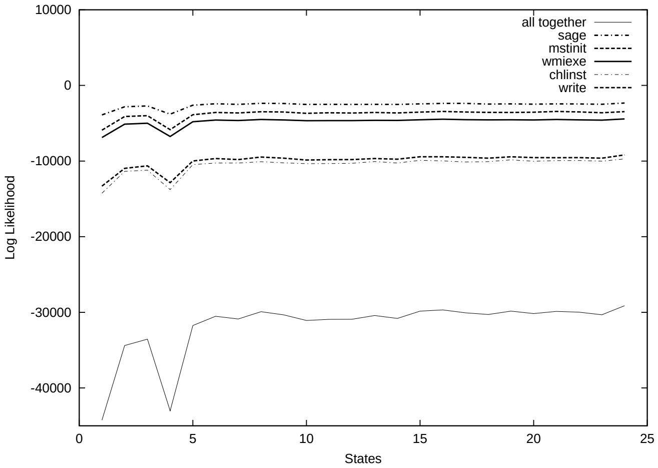


Figure 7.9: The log likelihood of observing each separate program in the HMMs trained for the set of all program.

each of the virus infected programs deviate so much from the normal behaviour that we can not represent the computed log likelihood. We can therefore conclude that a HMM trained for a set of 5 programs can surely detect if any of the programs have been infected with the *Apathy* virus.

We have now seen how the HMMs are at handling a small set of programs, but how does it cope with a larger set of programs? To experiment with this, we trained 15 HMMs having from 1 to 15 states with a set of 27 programs from the windows98 distribution. In this way we got 15 different profiles representing the 27 programs normal behaviour. Figure 7.10 show how long time it took to train the 15 HMMs.

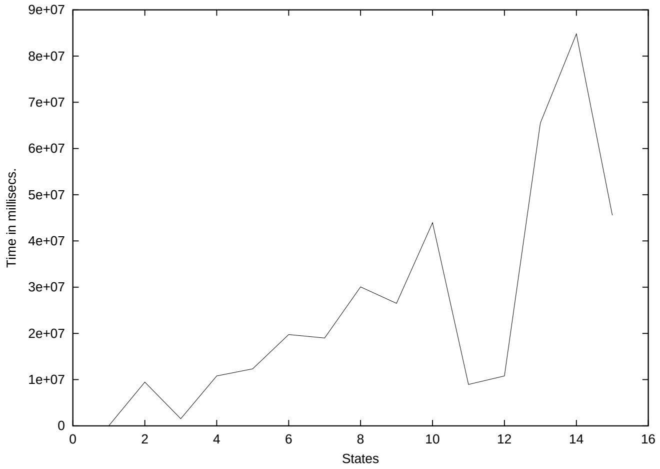


Figure 7.10: Training 15 HMMs having from 1 to 15 states on 27 programs with sizes ranging from 12KB to 29KB, the training was carried out on a Pentium III 733 MHz machine running Redhat Linux with the Kaffe Virtual Machine 1.0.5 for executing java 1.1 byte-code.

We then infected some of the 27 programs with the *Apathy* virus to see how much the infected programs would deviate from the normal behaviour profile. In contrast to the HMMs trained for the set of only 5 programs, we were now able to computed the log likelihood of observing the infected programs. To illustrate how much the virus infected programs deviated from the 15 normal behaviour profiles, we computed the log match affinity $\log10[P_{ma}]$. This value is simply the log likelihood difference between observing the infected program and the non-infected program. We plot the log likelihood difference for the 4 programs: sage, mstinit, wmiexe, and chlinst; see figure 7.11 on the facing page.

It might be difficult to see in the figure, but the log likelihood differences are in fact almost the same for all the 4 programs, the 4 curves are just plotted on top of each other. This means that all 4 programs deviate equally much from the normal behaviour profile. In fact, further experiments showed us that every *Apathy* infected program deviated with the same amount from the normal behaviour profile. This observation leads us to the following equation:

$$\text{constant} \approx \log10[P(\mathit{Apathy}(O^{(l)})|\lambda)] - \log10[P(O^{(l)}|\lambda)], \quad 1 \leq l \leq L,$$

where $O^{(l)}$ denotes the l'th program, L is the number of programs that the HMMs are trained for, and $\mathit{Apathy}(O^{(l)})$ denotes that $O^{(l)}$ is infected with the

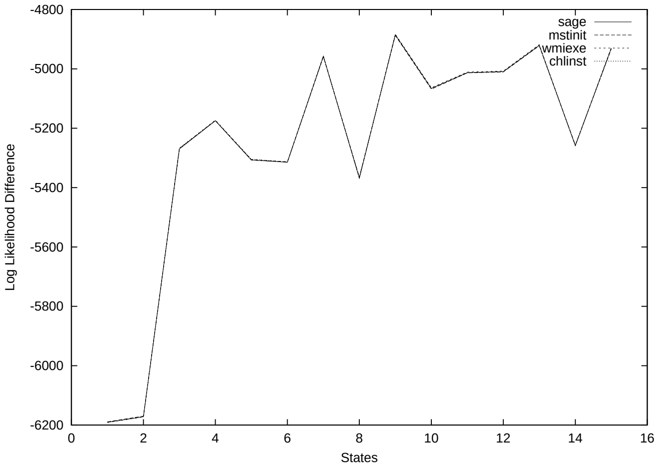


Figure 7.11: Log likelihood difference of observing programs before and after infection of *Apathy* virus in HMMs trained for a set of 27 programs.

Apathy virus. The equation illustrates that the HMMs representing the normal behaviour for the 27 programs are able to detect the same kind of behavioural change in all of the 27 infected programs, and that the constant value is an expression of the average changed behaviour caused by the virus.

What we can conclude from these experiments is that HMMs trained for several programs are able to detect if the programs later are infected with the *Apathy* virus, but it gets harder and harder for the HMM to detect the infection when using more and more programs to train the HMM after.

7.4 Detecting Viruses with HMMs Trained for Traces of System Calls

In this section we will describe some experiments made with HMMs and traces of system calls. We will especially focus on an experiment made with the ping program from the windows 98 distribution. All the experiments described in this section was made on Pentium III 450 MHz machine running Redhat Linux with the Kaffe Virtual Machine 1.0.5 for executing java 1.1 byte-code.

To track the system calls generated by the ping program we use a shareware program knowns as APISPY, the program can be freely downloaded from `http://www.wheaty.net/downloads.htm`. APISPY will trace all system calls made by a program to the system dll's in the windows system. APISPY will write the system calls, their arguments, their types, and their return values to a text file. A normal system call will look like this in the text file:

```
CharToOemA(LPSTR:008A0D38:"130.225.76",LPSTR:008A0D38:"130.225.76")
CharToOemA returns: 1
```

To compress the information, we have developed an `APIParser` class, which will substitute the system calls with integer numbers and write these together with the arguments, types and return values to a binary file. To compress the information even further it is possible to only include the numbers representing the system calls in the binary file. More information on the `APIParser` class is given in appendix D.4 on page 152.

In this experiment we trained a HMM for 41 traces of system calls. The 41 traces was generated by APISPY when running the ping program with different kinds of parameters: we tried to ping ourself, a machine on line, a machine not on line, ping with different package size, different timeouts, time to live, etc. We also tried to keep on pinging a machine until the program was interrupted. The 41 traces generated by APISPY was then put through the `APIParser` to generate 41 binary traces with numbers representing only system calls, in other words we did not save system arguments, types, and return values in the binary traces. Before been through the `APIParser` the size of the 41 traces ranged from 197 to 13771 bytes, after the `APIParser` the size of the 41 binary traces ranged from 8 to 620 bytes. To see the effect of using HMMs with increasing number of states we trained 29 HMM having from 1 to 29 states. All 41 binary traces were used to train every HMM. In figure 7.12 we have plotted the time it took to train the 29 HMMs for the 41 binary traces of system calls.

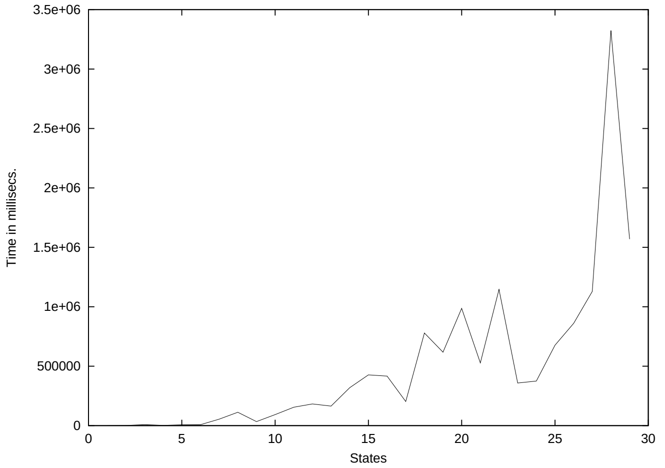


Figure 7.12: The time it took to train 29 HMMs on 41 binary traces of system calls.

After training the HMMs on the 41 binary traces we computed the average log likelihood log10[$P_{average}$] of observing the 41 binary traces in the HMMs. Generally we computed the log likelihood log10[$P(O^{(i)}|\lambda)$] for every ith binary trace and found $\log10[P_{average}]$ as $\sum_{i=1}^{41} \frac{\log10[P(O^{(i)}|\lambda)]}{41}$. In figure 7.13 we have plotted $\log10[P_{average}]$ for the 29 HMMs having from 1 to 29 states.

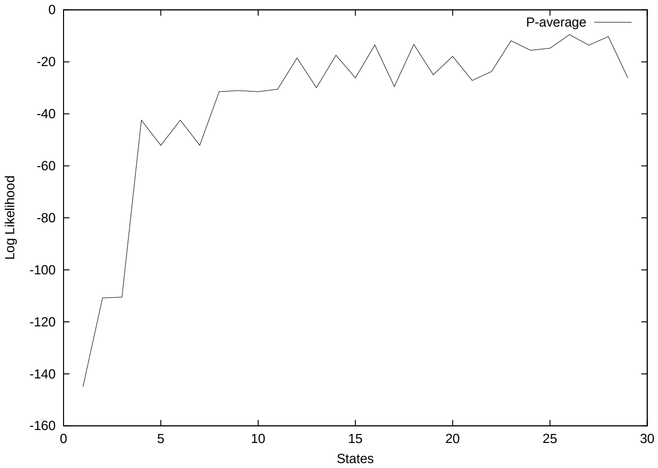


Figure 7.13: The average log likelihoods of observing the 41 binary traces in each of the 29 HMMs having from 1 to 29 states.

From figure 7.13 we can see that the average log likelihood is improved whenever we increase the number of states in the HMMs. The 29 HMMs will together with the log likelihoods plotted in figure 7.13 represent 29 different normal behaviour profiles for the ping program.

To see if the HMMs were able to recognise normal behaviour of the ping program we generated two new traces different from any of the 41 traces used to train the HMMs with. The two new traces were longer than any of the 41 other traces and were generated by giving different values of parameters to the ping program. To be correct we pinged another machine with a package size of 32 bytes with roughly 20 echo request, resulting in a binary trace of 772 bytes, and then we pinged our self with a package size of 64 bytes with roughly 50 echo request, resulting in a binary trace of 1744 bytes. In figure 7.14 on the following page we have plotted the average log likelihoods together with the log likelihoods of observing the two new binary traces in the HMMs. The figure gives an expression of how the two new traces deviate from the average log likelihood.

As we can see from figure 7.14 on the next page the two new binary traces do deviate from the average normal behaviour, but once the HMMs have over 23 states the deviations are not that big. This is quite good if we recall that the two binary traces were generated from executions of the ping program with different values of parameters and that the binary traces was much longer than the normal ones. We can therefore conclude that HMMs with some deviations

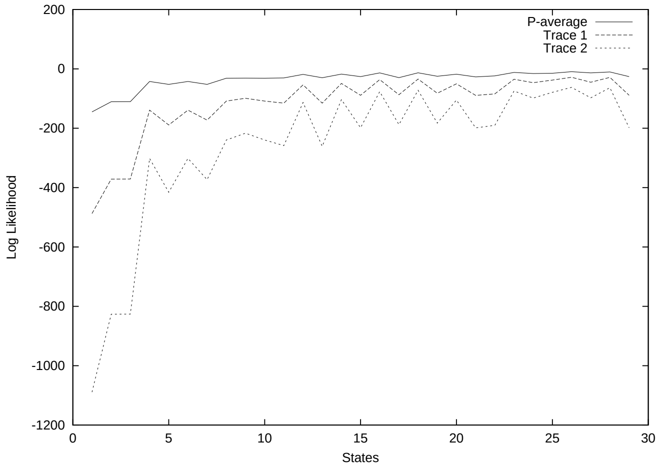


Figure 7.14: The average log likelihood of the normal behaviour together with the log likelihood of observing the two new binary traces.

can recognise the two new binary traces as having similar kinds of behaviour as the 41 original traces.

Next we made an experiment were we infected the ping program with the *Apathy* virus, to see how much it would deviate from the normal behaviour profile represented by the 29 HMMs and the computed average log likelihoods. We executed the infected ping program and tracked the systems calls with APISPY, resulting in a 14345 byte long trace. We then put the trace through the `APIParser` and got a 628 byte long binary trace of system calls. The binary trace representing the system calls of the infected program was then tested in the 29 HMMs representing the normal behaviour of the ping program. As it turned out the log likelihoods of observing the binary trace in the HMMs were so small that they could not be represented within the boundary of double precision values. In others words the behaviour of the infected ping program deviated really much from the normal behaviour.

The above experiments have convinced us, that we can use traces of system calls generated by a program to train a HMM to represent the normal behaviour of a program. We saw how normal traces not use during the training period deviated a little bit from the normal behaviour, but it was nothing compared to the deviations seen from the trace generated by a virus infected program. In this way the HMMs are clearly able to distinguish between normal and abnormal behaviour of a program.

7.5 Conclusion

What we have seen in this chapter is that HMMs can be used to train the normal behaviour of a program using either the complete binary code of a program or by using traces of system calls generated by a program.

We saw how small randomly made changes to the binary code of a program could be detected because the changed program deviated from the normal behaviour represented by the HMMs. Furthermore, when more than 15 bytes are randomly changed in a program with a size of 4KB it deviates so much from the normal behaviour, that we can not even represent the log likelihoods of observing the changed program in the HMMs.

To test how good the HMMs were at detecting changes made to programs due to virus infections, we infected some programs with the *Apathy* virus. The HMMs were trained on the complete binary code of the program to built a normal behaviour profile for it. Afterwards we infected the program with a virus and saw how much the program code now deviated from the normal behaviour. The results showed that the infected program deviated so much from the normal behaviour profile that we were not even able to represent the log likelihood of observing the infected program in the HMMs.

Another experiment showed that when training a HMM for several programs we were still able to see if any of the programs had been infected with a virus. But it also showed that it got harder and harder for the HMM to detect the deviations when more and more programs were used to train a the HMM.

Finally we saw how traces of system calls generated by a program also could be used to train a HMM representing the normal behaviour of the program. We trained a HMM with 41 traces generated by normal execution of a program. Then we generated a trace by executing the same program infected with the *Apathy* virus. The infected trace deviated so much from the normal behaviour that we could not represent the log likelihood of observing this trace in the HMM. We also made an experiment with traces generated by executing the non-infected program with different values of parameters than used during the training period. We saw how these deviated a little bit, but it was nothing compared to the trace generated by the infected program. This showed us that the HMMs could distinguish between normal and abnormal traces of system calls.

All these experiments have convinced us that the HMMs can be used to detect abnormal behaviour in programs due to the infection of the *Apathy* virus. We can off course not generalise and conclude that HMMs can be used to detect all kinds of viruses because we have only experimented with the *Apathy* virus, but it seems like HMMs are really good a detecting changed behaviour due to virus infections.

The experiments also showed that it took a long time to train the HMMs when training them on the complete binary code of programs. It might be a good idea to try to figure out how we could represent the static code of programs in a smaller and better way if this approach was going to be used in our computer immune system. The dynamic approach on the other hand seemed to be rather fast compared with the static one, here we trained a HMM with 4 states for 41 traces of system calls generated by the ping program in only 2380 milliseconds, whereas training on the complete binary code of the ping program took 401367 milliseconds in a similar HMM. Furthermore, when training on traces of system calls we only track the behaviour of the executed code and do not spend time and resources on code, which is never going to be executed under normal use of the program. This fact together with the fast training time for traces of system calls, convince us that the dynamic approach is better than the static one.

Chapter 8

Discussion

8.1 Summary

We have in this thesis examined how the most important components and techniques of the biological immune system operate. We have described and analysed already existing computer systems designed with inspiration from the biological immune system. We especially looked at IBM's computer immune system for virus detection and elimination and UNM's applications for intrusion detections. Then we discussed how we could model the most important components and mechanisms of the biological immune system in a computer. We looked at how we could represent self and nonself and how to do loose matching in our system. We proposed three different matching approaches: Hamming Distance, R-Contiguous Symbol and Hidden Markov Models and compared them. With inspiration in the computer immune systems designed and implemented by IBM and UNM together with inspiration from the discussion of how to model the immune system in a computer, we designed and discussed a computer immune system for virus detection. We reasoned why we had chosen HMMs to do the loose matching in our system and proposed several different kinds of ways to use the HMMs. We did several experiments with HMMs and virus infected programs to test the different ways. The experiments indicated that all the different ways of using the HMMs could be used to detect that a program had been infected with a virus known as *Apathy*.

8.2 Conclusion

Inspired by the components and techniques from the biological immune system and already published papers on computer immune systems, we have designed a system for virus detection. Like the biological immune system is able to distinguish between self and nonself, we have designed a system which is able to distinguish between normal behaviour and abnormal behaviour in programs.

Through our research, we found that the normal behaviour of a program could be represented by HMMs and discussed two different approaches to train the HMMs for the normal behaviour: a static approach and a dynamic approach. In the static approach we defined the normal behaviour from the binary code of a program, whereas we in the dynamic approach used traces of system calls generated by a program to define the normal behaviour of a program. Furthermore, we discussed how we could train the HMMs to represent the normal behaviour of the programs; we discussed training one HMM for every program, and training one HMM for a set of programs.

To test all the different ways of using the HMMs we implemented some software in Java. A framework for representing and training the HMMs were developed. We experimented with representing the behaviour of programs by training HMMs on binary code from the programs and on traces of system calls generated by the programs. We observed how well the HMMs detected randomly made changes in programs, and how well they detected that the programs had been infected with a virus.

The experiments indicated that the HMMs were quite good at detecting changed behaviour. HMMs trained on static code from one or several non-infected programs were able to detect if any of the programs were infected with a virus. The same positive results were found when training a HMM on traces of system calls generated by a non-infected program – again the HMMs were able to detect that the program were infected with a virus. We can therefore conclude that HMMs can be used to detect changed behaviour in programs due to virus infections.

From the experiments we also learnt that training HMMs on traces of system calls were much faster than training HMMs on binary code. This together with the fact that the dynamic approach can detect the virus while it is trying to execute itself convince us that the dynamic approach is the ideal approach to use. Furthermore, the dynamic approach enable us to train in a real environment tracking how the user normally use the programs. In this way we are able to get a more true profile of how the programs normally behave in the user's environment.

8.3 Future Work

Although the results presented in chapter 7 suggest that the HMMs could be used in detecting viruses, further testing must be completed to validate this. Especially tests with other viruses, more programs and tests made in real environments should be carried out. During our experiments we have not been presented with any problem of false negatives, that is recognising a program as not being infected with a virus although it is. Maybe this is because we used the

Apathy virus that adds over 5KB of viral code to the program when infecting it, which is quite a big change for the small programs we have been experimenting with. Clearly test with smaller and more clever viruses should be carried out to test the HMM's ability to detect these.

With the experiments we have not been concerned with the efficiency of the approaches. We have only concluded that the dynamic approach is much faster than the static one, which is also one of the reasons why we believe it is better than the static one. But when training and detecting in a real environment as the program is being executed, the user might be annoyed by the extra computation time that the computer immune system will require. The system therefore needs to be as effective as possible, and if it turns out that the training and the detecting in our system takes to long time, we might cut down on the number of different system calls tracked, or only use every second or third system call generated by the program.

In our design of the computer immune system we have not been concerned with how to eliminate the virus, but clearly a complete computer immune system should also be able to take some kind of action when detecting a virus. One of the most obvious actions to take, when using the dynamic approach, is simply to stop executing the program as soon as the system detects that the program behaves in an abnormal way. The user should then be alerted about the abnormal behaviour and could for instance either delete the program or quarantine it, disabling it to cause any further damage.

Appendix A

The Immune System

We will in this chapter give an introduction to the immune system of the human body. We will focus on some of the general properties of the immune system and the mechanisms provided by it, and not so much on how the different cells are built or interact through signals and bindings. The chapter could be seen as an introduction to the basic concepts of the immune system, explaining the working of the ***innate immune system*** and ***adaptive immune system***. The innate immune system is a defence mechanism inherited through genes from our parents, whereas the adaptive immune system is a defence mechanism built from gene rearrangements and adaptive learning in our body.

We introduce a lot foreign words, probably not to well known for computer scientists; these will be written in the font ***font***, and are explained on the way in the text and further described in the glossary at the end of this thesis; see page 159.

The chapter takes reference in the book *Immunobiology: The Immune System in Health and Disease, 5th Ed.*; see [1]. This is known to be one of the best and newest books in the area of basic immunobiology intended for medical students and scientists who want to know more about the immune system.

We start by explaining the primary goal of the immune system and how it is built from different layers of defence systems. Then we take a look at the most important players of the immune system and go into depth with the innate immune system together with the adaptive immune system. Finally we summarise some of the most important characteristics of the immune system.

A.1 Goal of the Immune System

The immune system of the human body resembles that of other species; primitive systems used for recognition and signalling, which might be thought of as being very specific for each species, have been found similar in humans, birds, fishes, reptiles and even plants. This indicates that the immune system is an evolutionary process, evolved through time to give all living organisms some kind of defence mechanism against infectious agents.

This defence mechanism has been able to protect us from infections by evolving and adjusting itself to the surrounding environment. The immune system is therefore characterised by the selection of those organisms that are best fit for the elimination of infectious agents. These organisms will gain the necessary resources for survival and reproduction; all others will be eliminated.

A.2 A Layered Defence System

The immune system is built from different layers of defence systems. The skin protects us from microorganisms by forming a seal of cells held together by tight junctions. Physical conditions such as low pH value and relative high temperature[1] just below and in the skin give microorganisms poor living conditions. The innate immune system, which is able to recognise a broad class of infectious agents, can trigger an immediate response and destroy them. And if the innate immune system is unable to handle the infection, it will present it to the adaptive immune system, which has the ability to recognise a much wider variety of infectious agents than the innate immune system. The response triggered by the adaptive immune system, to destroy the infectious agents is though, in contrast to the innate immune system, rather slow. The characteristics of the innate and adaptive immune system is summarised in table A.1.

Characteristics	Innate	Adaptive
Recognition	Able to recognise almost any infectious agents; the recognition is not particularly specific	Able to recognise any infectious agents; the recognition is very specific
Response time	Immediate and fast	Slow and long
Evolved through	Genes inherited from our parents	Gene rearranging and adaptive learning.

Table A.1: Advantages and disadvantages of the innate and adaptive immune system.

The four different layers of defence systems, which make up the immune system of the human body, are sketched in figure A.1 on the next page.

[1] Many infections grow better at lower temperature than the body's 37 °C.

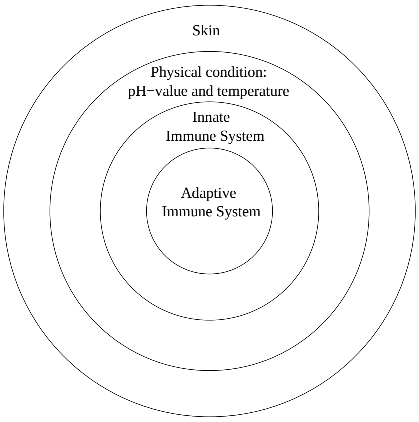


Figure A.1: The natural immune system consist of different layers.

A.3 Players of the Immune System

The major players of the immune system originate from stem cells in the bone marrow. These cells, known as pluripotent hematopoietic stem cells, can divide to produce all the cellular elements of our blood. The stem cell divides into two more specialised stem cells: common lymphoid progenitor and common myeloid progenitor, these give rise to white blood cells including the ***B-cells*** and ***T-cells***, red blood cells that can carry oxygen, and platelets that are important in blood clotting; see figure A.2.

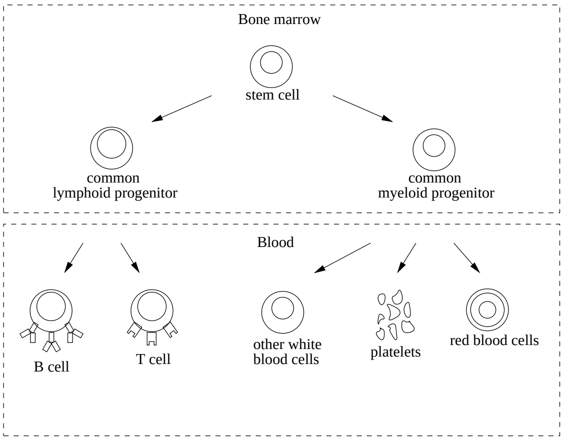


Figure A.2: All the cellular elements of our blood originate from stem cells in the bone marrow.

The white blood cells include or advance into cells like ***macrophages***, ***den-***

dritic cells, ***neutrophils***, and ***eosinophils***, which performs various important functions in both the adaptive and innate immune system, whereas the B and T-cells, also known as B and T-***lymphocytes***, are only responsible for the adaptive immune system. The macrophages and neutrophils, also known as ***phagocytes***, take up bacteria and destroy them as a part of innate immunity, and eosinophils have the ability to kill parasites coated with ***antibody***. Antibody is a protein that is induced and secreted from different plasma cells of the body against different kind of *substances*. Such *substances* are known as ***antigens*** because they can stimulate the generation of antibodies [1, p.2]. When the innate immune system is not able to destroy the infection, macrophages and dendritic cells take up the infectious agents and present them to the adaptive immune system.

A.4 Innate immunity

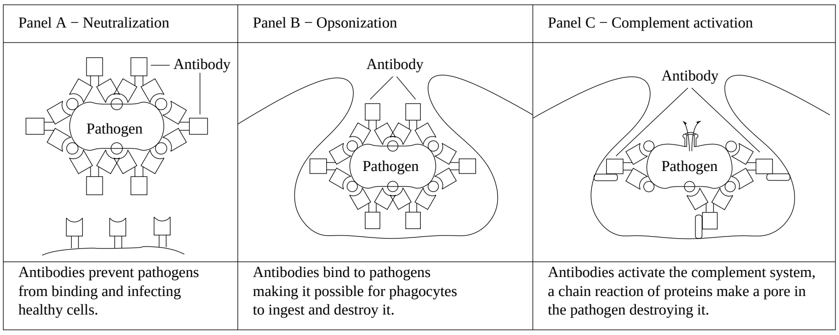
When a microorganism compromises the two first layers of defence, the skin and the physical conditions, and starts to replicate, the innate immune system will in most cases immediately start to recognise the intruders. Phagocytes like macrophages residing in tissue and neutrophils in blood, help in recognising, ingesting and destroying many of the ***pathogens***, which are the microorganisms capable of causing a disease. Some phagocytes produce toxic products like acid, toxic nitrogen, and toxic oxygen when the pathogens are ingested, others prevents the pathogens from surviving by binding essential nutrients and preventing their uptake by the pathogen. To engulf, ingest and destroy the pathogens, antibodies produced by especially B cells, participate in host defence in three main ways:

Neutralisation The antibody binds to pathogens and their toxic products thereby blocking their access to the cell that they might infect or destroy; see figure A.3 on the next page panel A.

Opsonization The antibody enables a phagocytic cell to ingest and destroy the pathogen by coating the pathogen; see figure A.3 on the facing page panel B.

Complement activation The antibodies activate a system of different proteins, known as the complement system. The activation of the complement system is done by antibodies and starts a chain reaction of proteins forming a pore in the membrane of the pathogen, this allows for free passage of water and other substances destroying the cell; see figure A.3 on the next page panel C.

The recognition of pathogens by macrophages and neutrophils results in release of cytokines and chemokines. These two products will attract other phagocytes to the site of infection, result in a production of new fresh phagocytes, contain the spread of infection to the blood stream by an activation of clotting mechanisms, and an elevation of the body temperature, causing a fever which



Panel A − Neutralization

Panel B − Opsonization

Panel C − Complement activation

Antibodies prevent pathogens from binding and infecting healthy cells.

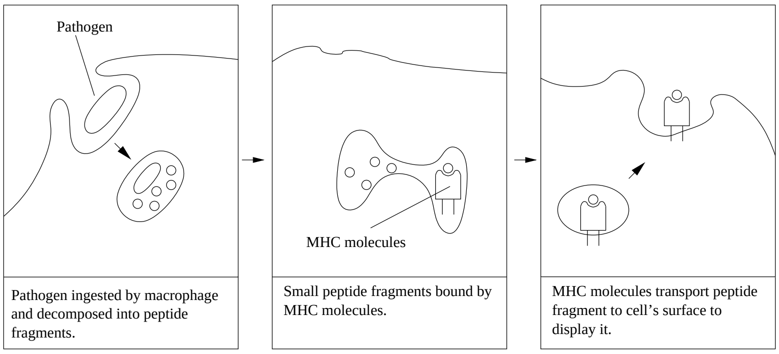
Antibodies bind to pathogens making it possible for phagocytes to ingest and destroy it.

Antibodies activate the complement system, a chain reaction of proteins make a pore in the pathogen destroying it.

Figure A.3: The main three defence to eliminate, ingest and destroy the pathogens.

is beneficial to host defence, because most pathogens grow better at lower temperature.

If the innate immune system is not able to destroy the pathogens, the phagocytes work like containers until the adaptive immune system can recognise the pathogens. The pathogens engulfed in the phagocytes are presented to the T-cells of the adaptive immune system by molecules called ***major histocompatibility complexes*** (MHC). Pathogens that have infected phagocytes, produce small peptide fragments, which are bound by the MHC molecules in the phagocyte. The MHC molecules travel to the surface of the cell and display the peptide fragment for the T-cell; see figure A.4.



Pathogen ingested by macrophage and decomposed into peptide fragments.

Small peptide fragments bound by MHC molecules.

MHC molecules transport peptide fragment to cell's surface to display it.

Figure A.4: A pathogen is ingested by a macrophage and small peptide fragments are displayed on the cell's surface by MHC molecules.

A.5 Adaptive Immunity

The B and T-cells, or B and T-lymphocytes as they are also known, make up the adaptive immune system and originate from stem cells in bone marrow. The T-cell migrates from the bone marrow through the blood and into the ***thymus*** to mature, thus the name T cell, whereas the B-cell stays in the bone marrow and develop there, thus the name B-cell. The thymus is an organ located in the upper part of the middle chest just behind the breastbone.

The success of the adaptive immune system is bound to the lymphocyte's ability to recognise harmful substances, known as ***nonself***, from harmless ones, known as ***self***. Lymphocytes are through their immature stages presented to self antigens and those who react will die by programmed cell death also known as ***apoptosis***. The recognition of substance is done through ***receptors*** on the lymphocytes. The receptors on the lymphocytes have only one specificity, but millions of different lymphocytes are developed every day.

Once matured, the lymphocytes will migrate from the bone marrow and the thymus, to the peripheral lymphoid organs such as the lymph nodes and the spleen, circulating in the blood and the lymphatic system. Here they will meet with antigens carried from site of infections by macrophages and dendritic cells. The lymphocytes will bind to the antigens, which will activate them to proliferate into effector cells with specific functions. When B-cells are activated they will proliferate into plasma cells and start to secrete antibodies with the help of some activated T-cells. When T-cells are activated they proliferate into two different effector cells: ***cytotoxic T- killer cells***, that kills the cell with the engulfed pathogen to prevent virus replication, or ***T-helper cells***, that help in activating other cells such as the B-cell to secrete antibodies. In the peripheral lymphoid tissue, lymphocytes that have not encountered foreign antigen will receive signals necessary for them to survive. In this way the body can maintain a fixed amount of lymphocytes, ensuring that the receptors of the lymphocytes are functional, and assert that only those capable of recognising foreign antigen will survive.

A.5.1 The T-Cell

The T-cell progenitor originates in the bone marrow, and migrates through the blood to the thymus to mature and differentiate. In the thymus the receptor genes of the T-cell are randomly rearranged and the immature T-cell is exposed to self antigen presented by MHC molecules. Those T-cell receptors which recognise the MHC molecules, but do not react strongly to the self antigen receive a survival signal, known as ***positive selection***. Whereas those who cannot recognise the MHC molecule or those who react strongly to the self antigen, will die from programmed cell death or receive a signal that also leads to cell death; this is known as ***negative selection***. The receptors of the T-cell must be able to recognise the body's own MHC molecules, otherwise it would not be able to respond to foreign antigen presented on cell surfaces by MHC molecules. To further mature the T-cell, it is also exposed to dendritic cells in the thymus, here it will receive a signal leading to cell death if it binds to self antigen on the dendritic cell.

The T-cell can mature in two ways, recognising MHC class I molecules or recognising MHC class II molecules. A T-cell that recognises MHC I molecules, mostly found on dendritic cells, will later differentiate into a cytotoxic T-cell able to kill other virus infected cells, whereas T-cell recognising MHC II, mostly found on B-cells and macrophages, will help in activating and differentiating the cell displaying the antigen.

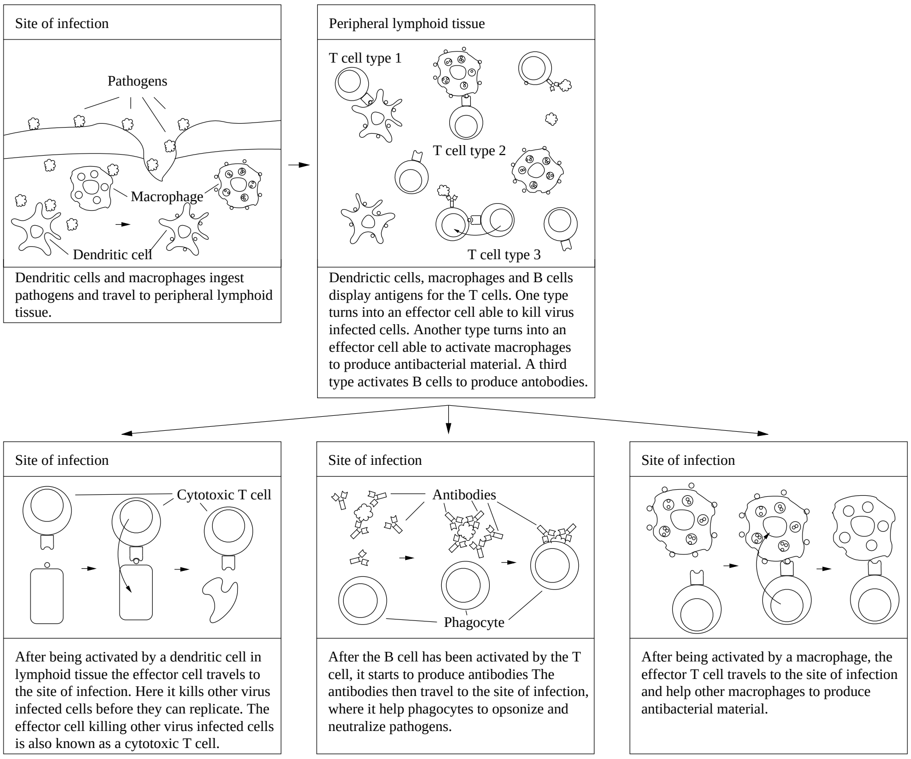
In mice 5×10^7 T-cells are generated every day, but only 2-4% of these will mature and leave the thymus [1, p.232]. The rate of T-cell's development is highest before puberty and declines after puberty. Experiments in mice have shown that if the thymus is removed after puberty, the function of the T-cells will still be maintained. This indicates that once a certain repertoire is present, the number of T-cells are maintained by cloning mature T-cells [1, p.231].

When the T-cell is matured it will turn to the peripheral lymphoid tissue where it will meet with dendritic cells and macrophages. The dendritic cells and the macrophages have ingested the pathogens at the infected site and travelled to the nearest lymph node to display the antigens for the T-cells. One type of T-cells recognise the antigens presented by dendritic cells and turns into an effector cell able to kill other virus infected cells before the virus can replicate. Other types of T-cells recognise antigens presented by macrophages and turn into effector cells by activating them to produce antibacterial material by releasing cytokines. A third type of T-cells in the peripheral lymphoid tissue will meet with B-cells, and help them in destruction of extracellular pathogens by recognising antigen displayed by MHC II molecules on the B-cell's surface and activating them to produce antibodies. The antibodies produce by the B-cell will in turn with T effector cells, migrate to the site of infection to help phagocytes in ingesting and neutralising the infection, and kill virus infected cells; see figure A.5 on the next page.

A.5.2 The B Cell

The B-cell originates from stem cells in the bone marrow, and in contrast to the T-cell, the B-cell stays in the bone marrow to mature and proliferate. In the bone marrow, rearrangement of the genes in the B-cell progenitor, causes the receptors of the B-cell to develop. The immature B-cell can now interact with antigens in its environment by a very simple type of receptor on its cell surface. Any B-cell that reacts strongly to self antigens receives a signal leading to cell death in a process of negative selection.

The almost mature B-cell now leaves the bone marrow and migrates to circulate in the peripheral lymphoid organs and blood. In this environment the



Dendritic cells and macrophages ingest pathogens and travel to peripheral lymphoid tissue.

Dendrictic cells, macrophages and B cells display antigens for the T cells. One type turns into an effector cell able to kill virus infected cells. Another type turns into an effector cell able to activate macrophages to produce antibacterial material. A third type activates B cells to produce antobodies.

After being activated by a dendritic cell in lymphoid tissue the effector cell travels to the site of infection. Here it kills other virus infected cells before they can replicate. The effector cell killing other virus infected cells is also known as a cytotoxic T cell.

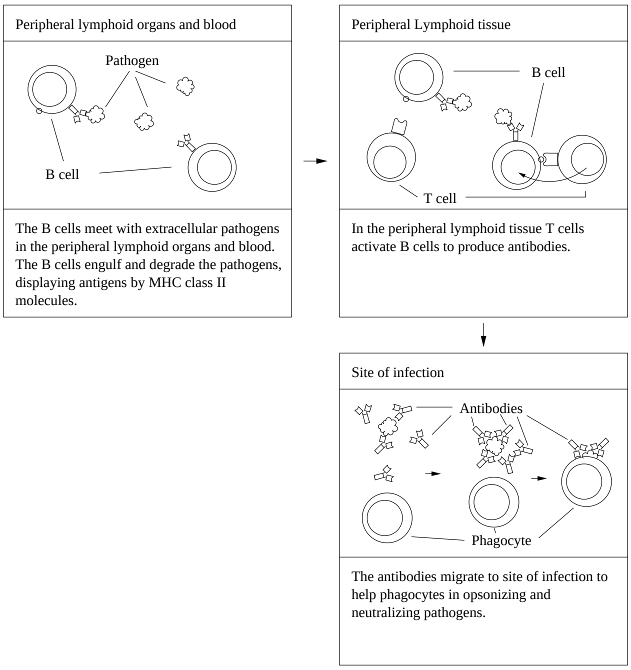
After the B cell has been activated by the T cell, it starts to produce antibodies The antibodies then travel to the site of infection, where it help phagocytes to opsonize and neutralize pathogens.

After being activated by a macrophage, the effector T cell travels to the site of infection and help other macrophages to produce antibacterial material.

Figure A.5: Pathogens are ingested at the site of infection, in lymphoid tissue the antigens are presented to T-cells. The T-cells turns into effector cells or activate B-cells to produce antibodies. The effector T-cells and the antibodies migrate to the site of infection. Here they help to kill, neutralise and opsonize the pathogens.

B-cell keeps on further developing, and evolves to express more advanced types of receptors on its surface. In the these environments the B-cell also meets with extracellular pathogens, which are engulfed, degraded and displayed at the B-cell's surface by MHC class II molecules. The T-helper cells recognise the antigens displayed by the MHC II molecules and activates the B-cell to differentiate into a plasma cell producing antibodies. The antibodies will neutralise the extracellular pathogens and help phagocytes in opsonizing and killing the pathogens; see figure A.6 on the facing page.

One could ask the question: *why does B-cells need activation from T-helper cells?* The answer to this is, that not all self reactive B-cells are killed in the bone marrow, and some of these might bind to self extracellular particles, resulting in an ***autoimmune response***. The extra verification from T-helper cells prevent the B-cells from becoming active and killing the self particles [1,


Peripheral lymphoid organs and blood
Pathogen
B cell
Peripheral Lymphoid tissue
B cell
T cell
Site of infection
Antibodies
Phagocyte

The B cells meet with extracellular pathogens in the peripheral lymphoid organs and blood. The B cells engulf and degrade the pathogens, displaying antigens by MHC class II molecules.

In the peripheral lymphoid tissue T cells activate B cells to produce antibodies.

The antibodies migrate to site of infection to help phagocytes in opsonizing and neutralizing pathogens.

Figure A.6: B-cells meet with pathogens in peripheral lymphoid organs and blood. These are ingested, degraded and displayed on the B-cell's surface by MHC class II molecules. In the peripheral lymphoid tissue the B-cells get activated by T-cells to produce antibodies, the antibodies help phagocytes to destroy pathogens.

p.259].

What is important to notice about B-cells is that they are able to recognise antigens that are present outside the cells (extracellular), where most bacteria are found, whereas T-cells, by contrast, can detect antigens generated inside infected cells (intracellular), for example those due to a virus[1, p.23]. Furthermore, B-cells produce antibodies when activated, helping phagocytes (macrophages and neutrophils) in neutralising, opsonizing and killing pathogens, whereas T-cells either kill infected cells or help in activating other cells.

A.5.3 Clonal Expansion and Immunological Memory

One major feature of the adaptive immune system is its ability to clone the lymphocytes and thereby increase the effect of adaptive immunity; this is known as ***clonal expansion***. When the lymphocytes meet with foreign antigens, circulating the peripheral lymphoid tissue and blood, they are activated and start to enlarge. The lymphocyte then start dividing itself in 3-5 days, resulting in clones of 1000 daughter cells [1, p.19]. Each of the daughter cells then proliferates into effector cells and carries out their specific functions as described above.

When the effector cells have eliminated the pathogen, most of the cells will die from programmed cell death, leaving a small amount of effector cells. These will differentiate into memory cells and form the basic of ***immunologically memory***, which will provide the body with long lasting protective immunity. The immunological memory give the body the ability to respond more rapidly and effectively when previously encountered pathogens are meet again.

The disease known as measles, affecting children, caused by a virus, and resulting in high fever and an eruption of red spots on the skin, is a good example of how immunological memory works in the human body. If a child has been exposed to the virus through vaccination or infection, the result will be long-term protection from measles.

The memory B-cell undergoes ***affinity maturation*** and is therefore more mature and has higher affinity than newly matured naive B-cells. The higher affinity for antigen recognition will increase their uptake of pathogens and they will be able to express the antigen more rapidly on their surfaces. This enables them to interact faster with T-helper cells than naive B-cells, and thereby increasing the amount and rate of specific antibody and thereby eliminating the pathogens faster. A fast response to a previously encountered pathogen could also happen from preexisting antibody, still floating around in the body as a result of the first encounter with the pathogen.

Memory T-cells are just long-lived T effector cells. They are not better than other T effector cells, whereas memory B-cells are, because they undergo further maturation and therefore have higher affinity than naive B-cells. As with the memory B-cell, the memory T cell can react more quickly than naive T-cells, this is because dendritic cells and macrophages in the first encounter with the pathogen, have already activated it.

A.6 Properties of the Immune System

The immune system of the body is a highly advanced and complex system of mechanisms for recognising pathogens, signalling between cells and evolving of best-fit cells. As explained above, the immune system is able to recognise self from nonself, and thereby being able to know which substances should be eliminated or not. Furthermore, if one of the lymphocytes fails to do its job correctly, it does not matter because there over millions of other lymphocytes doing it right - the system is fault safe because of the huge amount of lymphocytes distributed all over the body; in other words if something goes wrong it will not affect the whole body. As a consequence of recognising pathogens, the lymphocytes will be activated to clone themselves to respond more rapidly and effectively to the pathogen, thereby killing the virus and its host faster than it can replicate. To further increase the efficiency of the immune response, when recognising pathogens, phagocytes will secrete cytokines and chemokines to attract more pathogens to the site of infection and stimulate a production of new and fresh phagocytes. The immune system also has the ability to learn and remember, which gives it the ability to act more quickly to previously encountered pathogens. Finally, being a layered defence system, enables the immune system to make it much harder for pathogens to invade.

Appendix B

A Short Introduction to Viruses

A virus is often defined as a program that replicates itself by copying its own code into other environments. The virus normally copy itself to environments like executables files, document files, boot sectors and networks. A virus will often execute some sort of action separately from the replication, this can vary from annoying messages printed on the screen to disastrous actions like overwriting the Flash BIOS on the motherboard.

The most general classification of viruses is made from the environment in which they use to replicate themselves:

Boot viruses either copy themselves to a boot sector of a diskette or a master boot sector of a hard disk or change the pointer to an active boot sector such that virus code will be run instead. The virus code is executed when the computer is rebooted.

File viruses infect executable files by overwriting existing code or substituting original file with a virus file. The viruses code is executed when the infected file is executed.

Macro viruses infect document files, spreadsheets, presentation files and databases. The virus is written in a macro language supported by the application, which is normally used to handle these kind of files. The virus is executed when the file is opened, changed or saved from the application.

Network viruses use network protocols like FTP or email to spread. The virus code is executed when the user run an infected file or if some program is set up to automatically run incoming files.

The classification does not mean that a virus can not be of more than one of the above type; there is a lot of examples where viruses are of mixing types: both boot sector and file viruses, both macro and network viruses and so on.

Another term often used in relation with viruses is the term **worm**. A worm is a virus that replicates itself without the need of a host file. It replicates itself by creating a new file containing the virus code or by simply sending itself to others using network protocols. A file worm normally renames itself like INSTALL.EXE or SETUP.EXE to push the user to executed the virus. Whereas a network worm often use names from already existing files on the infected computer; this is done to make the receiving part of the virus believe that the worm is a trustworthy file or document send from the infected user.

When reading about different kinds of viruses, terms like memory resident, stealth, and polymorphic are often used to characterise the viruses. The meaning of these three terms are explained below.

Memory resident viruses are able to stay resident in memory, by leaving a copy of themselves in memory. Memory resident viruses is rather troublesome to remove because they keep on infecting files on the disk as long as they stay in memory, and the anti virus applications are not always able to find the virus code in the RAM. Even though the virus is removed from all the files on the hard disk, the virus is still in the system and able to infect new files again. The virus will only be removed from memory if the computer is rebooted or its process killed.

Stealth viruses cover their own presence in the system, such that the user is unable to detect their presence. Whenever an infected boot sector or file is read, the virus will substitute its own data with the uninfected original data. Furthermore, the virus also tries to hide itself, by changing the size of the infected file to the original size.

Polymorphic viruses change their own code by encrypting the main code of the virus with different keys or by rearranging the executable virus code. When a normal anti virus application search for viruses it uses a small signature, which describes the instructions and sequences of instructions in the virus code. If the instructions or sequences of instructions are changed every time the virus copy itself, or if the instructions of the main virus code are encrypted with different keys upon replication, the signature will not be able to match the virus code anymore.

For further information on viruses, their classification, and often used terms we refer to [20–27].

Appendix C

Hidden Markov Models

We will in this chapter give an introduction to Hidden Markov Models and some of the algorithms used in connexion with such a model. A Hidden Markov Model (HMM) is a state machine where all the state transitions have fixed probabilities. In this way we are able to express that some state transitions are more likely to happen than others. Furthermore, every state of the HMM will have a number of symbols, each with their own probability of being observed in that state. As an example we might have a state with two symbols: A and B, where the probability of choosing symbol A is 0.6 and the probability of choosing symbol B is only 0.4. In this way, we are able to express that it is more likely to observe symbol A than symbol B in the state concerned.

HMMs have existed since the 1960s, but it was first in the late 1980s the HMMs became popular. This was mainly due to better algorithms and their improved execution time, but also because HMMs where found useful in applications such as speech recognition. Later on the HMMs have been widely used in biological sequence analysis searching for patterns in e.g. DNA, and have been successfully applied to noise reduction in e.g. signal and image applications.

We start out by introducing the theory of HMMs through a well-known example from the literature [29] [28]: the Urn and Ball Example. Then we define the notation we are going to use and demonstrate it through the Urn and Ball Example. Later we describe algorithms for solving problems like finding the best path, how well does a sequence of symbols match a HMM, and how do we optimise a HMM to match a specific sequence of symbols. Finally we discuss some implementation issues when working with HMMs and solutions to these.

C.1 Introducing a Hidden Markov Model

A Hidden Markov Model is a state machine where each state transitions have fixed probabilities. The Markov Model is denoted *hidden* because we cannot be sure of the outcome of being in a certain state. Instead we use a probability distribution, to indicate the symbols most likely to be observed in the state concerned. To get a better understanding of the problem we take a look at the Urn and Ball example; see figure C.1.

C.1.1 Urn and Ball Example

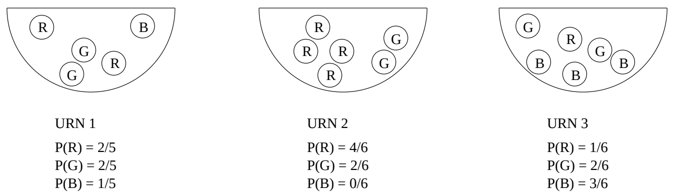


Figure C.1: The Urn and Ball example: each urn contains a number of coloured balls. A ball is picked from an urn, its colour observed and it is then put back into the same urn. A new urn is chosen, according to a probability distribution, and the procedure starts over again.

Here we have three urns all with different coloured balls in. In some urns there are more balls than in others, and the colours of the balls in each urn are also different. In urn number one we have five balls: two red, two green and one blue. The probability of choosing a red ball from urn number one is therefore 2/5, and the probability of choosing a green ball is 2/5 and so forth. The idea is to pick a ball from an urn, observe its colour and then put it back. After this a new urn is chosen. The selection of the new urn is done according to a probability distribution, defining how likely it is to move from the current urn to any of the other urns. The urns in this example represent the states in the HMM, whereas the balls represent the symbols observable in every state. The selection of a new urn indicates the state transition probability distribution, and the different amount of coloured balls in each urn represent the probability distribution of the observable symbols in each state. The Urn and Ball example presented in figure C.1 could be represented by the HMM sketched in figure C.2 on the next page.

The transition probabilities are not given directly from the Urn and Ball example in figure C.1, so here we just assume that user will not prefer any urn over another. The probability of making a state transition from one urn to another is 1/3, because we can make a state transition to all three states when being in one state and each of them is equally likely to take place. Though, one could picture an example, where some urns where placed more conveniently than others, if such a case the transition probabilities in figure C.2 on the next page should be changed accordingly.

At each clock tick, t, a new state transition is made. The clock tick is in this

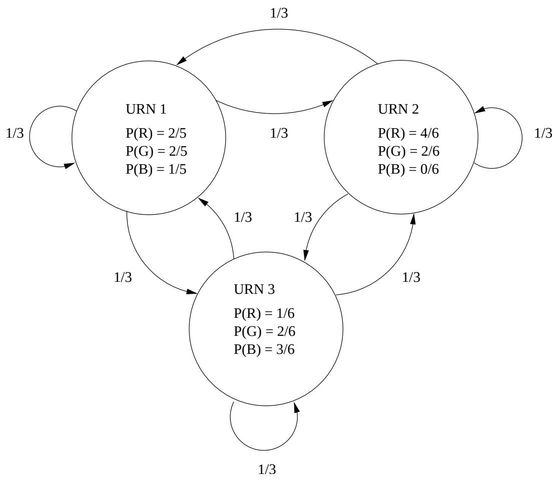


Figure C.2: The Urn and Ball example represented with a HMM.

definition represented by a positive whole number and will take place with the same frequency; thus we are not able to express that we can reside in one state longer than another. After a state transition is made a symbol is observed. The observable symbols used in this definition are discrete symbols such as numbers, letters and colours, but could in practise be anything: as for instance in speech recognition, where they use vectors of symbols.

C.1.2 General Notation

After introducing the Urn and Ball example we now turn to some more general notation, which we are going to use later on when looking at some of the algorithms for HMMs. In the notation below $q_t = S_i$ will denote that we are in state S_i at time t.

T: The length of the observed sequence or the total number of time ticks.
N: The total number of states in the HMM.
M: The number of distinct observation symbols.
O: The observation sequence of symbols $O = O_1 O_2 \cdots O_T$.
S: The set of states $S = \{S_1, S_2, \cdots, S_N\}$.
V: The set of observation symbols $V = \{v_1, v_2, \cdots, v_M\}$.
A: The state transition probability distribution: $a_{ij} = P(q_{t+1} = S_j | q_t = S_i)$, the probability of making a transition from state S_i at time t to state S_j at time $t+1$.
B: The observation probability distribution: $b_j(k) = P(v_k \text{ at } t | q_t = S_j)$, the probability of observing symbol v_k at time t given we are in state S_j at time t.
π: The initial probability distribution: $\pi_i = P(q_1 = S_i)$, the probability of being in state S_i at time 1.
λ: A HMM defined by its parameters A, B and π as $\lambda = (A, B, \pi)$.

In the following a state sequence of length T possible to take place in a HMM λ will be denote as:

$$Q = q_1 q_2 \cdots q_T \tag{C.1}$$

The probability of observing a symbol sequence $O = O_1 O_2 \cdots O_T$ given a state sequence Q and a HMM λ will be found as:

$$\begin{aligned} P(O|Q,\lambda) &= \prod_{t=1}^{T} P(O_t|q_t,\lambda) && \text{(C.2)} \\ &= b_{q_1}(O_1) b_{q_2}(O_2) \cdots b_{q_T}(O_T) && \text{(C.3)} \end{aligned}$$

The probability of a possible state sequence of length T to take place given the HMM λ is:

$$\begin{aligned} P(Q|\lambda) &= \pi_{q_1} \prod_{t=2}^{T} P(q_{t-1}|q_t,\lambda) && \text{(C.4)} \\ &= \pi_{q_1} a_{q_1 q_2} a_{q_2 q_3} \cdots a_{q_{T-1} q_T} && \text{(C.5)} \end{aligned}$$

And the joint probability of both observing the symbol sequence O and the state sequence Q at the same time is:

$$\begin{aligned} P(O,Q|\lambda) &= P(O|Q,\lambda) P(Q|\lambda) && \text{(C.6)} \\ &= \pi_{q_1} b_{q_1}(O_1) a_{q_1 q_2}\ b_{q_2}(O_2) a_{q_2 q_3} \cdots b_{q_T}(O_T) a_{q_{T-1} q_T} && \text{(C.7)} \end{aligned}$$

C.1.3 Urn and Ball Example Revisited

If we look at the HMM for the Urn and Ball example from figure C.2 on the preceding page, we can now express it more formally. The total number of states in the model:

$$N = 3$$

The set of states:

$$S = \{urn_1, urn_2, urn_3\}$$

The set of observation symbols:

$$V = \{R, G, B\}$$

The state transition probability distribution:

$$A = \begin{array}{c|ccc} j\backslash i & 1 & 2 & 3 \\ \hline 1 & 1/3 & 1/3 & 1/3 \\ 2 & 1/3 & 1/3 & 1/3 \\ 3 & 1/3 & 1/3 & 1/3 \end{array}$$

The observation probability distribution:

$$B = \begin{array}{c|ccc} j\backslash k & 1 & 2 & 3 \\ \hline 1 & 2/5 & 2/5 & 1/5 \\ 2 & 4/6 & 2/6 & 0/6 \\ 3 & 1/6 & 2/6 & 3/6 \end{array}$$

The initial state probability distribution:

$$\pi = \begin{array}{c|ccc} i & 1 & 2 & 3 \\ \hline & 1/3 & 1/3 & 1/3 \end{array}$$

For the initial state probability distribution π we assume that the user will not prefer to start in any urn different from another. The initial state probability distribution for urn 1, 2 and 3 is therefore all 1/3 because there are three urns and each of them is equally likely to be chosen to begin with.

Let us take an example to understand the notation better: what is the probability of choosing a red ball from urn number 2, moving on to urn number 1 choosing a blue ball, and ending in urn number 3 choosing a green ball. The observation sequence is given by $O = RBG$, the length of the observation sequence is $T = 3$, the state sequence is $Q = urn_2 urn_1 urn_3$, and the HMM is $\lambda = (A, B, \pi)$ where A, B and π are given as above.

$$\begin{aligned} P(O, Q|\lambda) &= P(q_1 = urn_2)P(R|q_1 = urn_2) \\ &\quad P(q_2 = urn_1|q_1 = urn_2)P(B|q_2 = urn_1) \\ &\quad P(q_3 = urn_3|q_2 = urn_1)P(G|q_3 = urn_3) \\ &= \pi_2 b_2(1)\ a_{21} b_1(3)\ a_{13} b_3(2) \\ &= 1/3 \times 4/6 \times 1/3 \times 1/5 \times 1/3 \times 2/6 \\ &= 8/4860 \end{aligned}$$

In the above calculations the notation of $b_1(3)$ indicates the probability of observing the colour B in state 1, we use the argument 3 to $b_1()$ because the colour B is at index 3 in the set of observation symbols $V = \{R, G, B\}$. As we can see, finding the probability $P(O, Q|\lambda)$ takes of the order of $2T$ calculations.

C.2 Algorithms for Hidden Markov Models

We are in this section going to take a look at some of the algorithms for HMMs. We first introduce a general procedure to generate an observation sequence $O = O_1 O_2 \cdots O_T$. After that we take a look at an algorithm called the Forward-Backward algorithm for calculating the probability of being state S_i at time t having observed the sequence $O = O_1 O_2 \cdots O_t$. Next we take a look at the backward-forward algorithm, which calculates the probability of being in state S_i at time t having observed the sequence $O = O_{t+1} O_{t+2} \cdots O_T$. Then an algorithm, called the Viterbi algorithm, for finding the best state sequence is introduced and finally we discuss the Baum-Welch algorithm for iteratively maximising the HMM parameters A, B and π.

C.2.1 Generating an Observation Sequence

To generate an observation sequence $O = O_1 O_2 \cdots O_T$, we can use the following small procedure:

step 1: Choose an initial state $q_1 = S_i$, according to the initial state distribution, π.
step 2: Set time $t = 1$.
step 3: Choose $O_t = v_k$ according to the observation probability distribution $b_i(k)$ for v_k in state S_i.
step 4: Make a transition from state $q_t = S_i$ to $q_{t+1} = S_j$ according to the state transition probability a_{ij}.
step 5: Set $t = t + 1$, go to step 3 if $t < T$ or stop algorithm.

If we take a look at the HMM of the Urn and Ball example in figure C.2 on page 109, we could generate the following observation sequence from the procedure above:

t	1	2	3	4	5	6	$\cdots$	T
q_t	urn_1	urn_3	urn_1	urn_2	urn_1	urn_1	$\cdots$	q_T
O_t	R	B	G	R	G	R	$\cdots$	O_T

Table C.1: An observation sequences generated from the HMM of the Urn and Ball example in figure C.2 on page 109.

C.2.2 The Forward-Backward Algorithm

One of the tasks of HMMs is to figure out how well an observed sequence of symbols $O = O_1 O_2 \cdots O_T$ matches a specific model λ. One method of doing this is to generate all possible state sequences Q_i of length T, finding the probability

$P(O|Q_i, \lambda)$ for all these generated state sequences, taking into account how likely each state sequence Q_i are, $P(Q_i|\lambda)$, and summing these up.

$$\sum_{\forall Q_i} P(O|Q_i, \lambda) P(Q_i|\lambda) \tag{C.8}$$

With our knowledge of basic probability, we know that if all the events of Q_i are independent and if $\sum P(Q_i|\lambda) = 1$, the probability in equation (C.8) could be written as:

$$P(O|\lambda) = \sum_{\forall Q_i} P(O, Q_i|\lambda) = \sum_{\forall Q_i} P(O|Q_i, \lambda) P(Q_i|\lambda) \tag{C.9}$$

So, to give a good estimate of how well an observation sequence O matches a specific HMM λ, we can find the joint probability of the observation sequence O and all the possible state sequences Q_i able to occur at the same time.

If the model λ has N states and the observation sequence is T long, we have to generate N^T different state sequences. For each of these state sequences Q_i, we have to find the probability $P(O, Q_i|\lambda)$, which takes in the order of $2T$ calculations, see the calculation of $P(O, Q|\lambda)$ in section C.1.3 on page 110, resulting in a total order of $2TN^T$ calculations. This is rather many calculations even for a small HMM.

So, instead of generating all possible state sequences and calculating the joint event of these and the observation sequence, we could do something smarter. We define a forward variable

$$\alpha_t(i) = P(O_1 O_2 \cdots O_t, q_t = S_i|\lambda), \qquad 1 \le i \le N \tag{C.10}$$

which denotes the probability of the joint event of observing the sequence $O = O_1 O_2 \cdots O_t$ and being in state S_i at time t.

The forward variable $\alpha_t(i)$ is found recursively, by the Forward-Backward algorithm, using the following two steps:

step 1: Initialisation.

$$\alpha_1(i) = \pi_i b_i(O_1), \qquad 1 \le i \le N \tag{C.11}$$

step 2: Induction.

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(O_{t+1}), \qquad \begin{array}{l} 1 \le j \le N \\ 1 \le t \le T-1 \end{array} \tag{C.12}$$

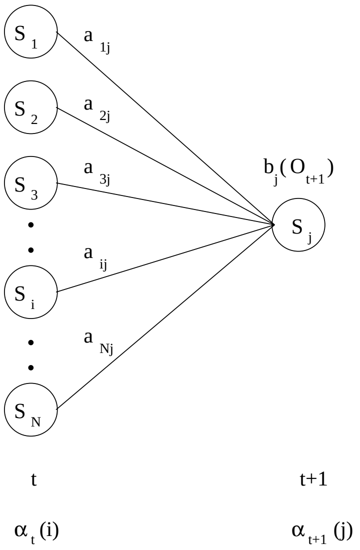


Figure C.3: Step 2 of the Forward-Backward algorithm. The algorithm reuses the forward variables $\alpha_t(i)$ when finding $\alpha_{t+1}(j)$.

Step 1 will find the joint probability of starting in state S_i and observing the symbol O_1. Step 2 will find the joint probability of having observed $O_1O_2\cdots O_t$ in all possible state sequences of length t and now observing O_{t+1} in state S_j; see figure C.3.

Given the Forward-Backward algorithm we can now calculate the probability $P(O|\lambda)$ as:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i) \tag{C.13}$$

This is so, because the forward variable $\alpha_T(i)$ is defined as

$$\alpha_T(i) = P(O_1O_2\cdots O_T, q_T = S_i|\lambda) \tag{C.14}$$

and when summing up all these probabilities for $1 \leq i \leq N$ we get:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i) \quad = \quad \sum_{i=1}^{N} P(O_1O_2\cdots O_T, q_T = S_i|\lambda), \tag{C.15}$$

because the forward variable $\alpha_T(i)$ gives us the joint probability of observing $O = O_1O_2\cdots O_T$ in all possible state sequences of length T being in state S_i at time T, and when summing up $\alpha_T(i)$ for every state S_i, we get the probability of observing O in every possible state sequence of length T for every state in the HMM.

The smart thing about the Forward-Backward algorithm for finding $P(O|\lambda)$ is that we can reuse the forward variables $\alpha_t(i)$, when finding $\alpha_{t+1}(j)$. To explain this further we take a look at an example:

Given the HMM $\lambda = (A, B, \pi)$ where the parameters A, B and π are taken from section C.1.3 on page 110, calculate how well the sequence $O = RBG$ matches the model. The model from section C.1.3 on page 110 has three states so we first need to find

$$\begin{aligned}
\alpha_1(1) &= \pi_1 b_1(1) = 1/3 \times 2/5 = 2/15 \\
\alpha_1(2) &= \pi_2 b_2(1) = 1/3 \times 4/6 = 4/18 \\
\alpha_1(3) &= \pi_3 b_3(1) = 1/3 \times 1/6 = 1/18
\end{aligned}$$

using equation (C.11).

We then find

$$\begin{aligned}
\alpha_2(1) &= [\alpha_1(1)a_{11} + \alpha_1(2)a_{21} + \alpha_1(3)a_{31}]b_1(3) \\
&= [2/15 \times 1/3 + 4/18 \times 1/3 + 1/18 \times 1/3] \times 1/5 = 37/1350 \\
\alpha_2(2) &= [\alpha_1(1)a_{12} + \alpha_1(2)a_{22} + \alpha_1(3)a_{32}]b_2(3) \\
&= [2/15 \times 1/3 + 4/18 \times 1/3 + 1/18 \times 1/3] \times 0/6 = 0 \\
\alpha_2(3) &= [\alpha_1(1)a_{13} + \alpha_1(2)a_{23} + \alpha_1(3)a_{33}]b_3(3) \\
&= [2/15 \times 1/3 + 4/18 \times 1/3 + 1/18 \times 1/3] \times 3/6 = 37/540
\end{aligned}$$

from equation (C.12), reusing $\alpha_1(1)$, $\alpha_1(2)$ and $\alpha_1(3)$.

And finally find

$$\begin{aligned}
\alpha_3(1) &= [\alpha_2(1)a_{11} + \alpha_2(2)a_{21} + \alpha_2(3)a_{31}]b_1(2) \\
&= [37/1350 \times 1/3 + 0 \times 1/3 + 37/540 \times 1/3] \times 2/5 = 259/20250 \\
\alpha_3(2) &= [\alpha_2(1)a_{12} + \alpha_2(2)a_{22} + \alpha_2(3)a_{32}]b_2(2) \\
&= [37/1350 \times 1/3 + 0 \times 1/3 + 37/540 \times 1/3] \times 2/6 = 152/14261 \\
\alpha_3(3) &= [\alpha_2(1)a_{13} + \alpha_2(2)a_{23} + \alpha_2(3)a_{33}]b_3(2) \\
&= [37/1350 \times 1/3 + 0 \times 1/3 + 37/540 \times 1/3] \times 2/6 = 152/14261
\end{aligned}$$

also from equation (C.12), reusing $\alpha_2(1)$, $\alpha_2(2)$ and $\alpha_2(3)$. The calculations of $\alpha_3(1)$ are shown graphically in figure C.4 on the following page.

The probability of the sequence $O = RBG$ given the model λ using equation (C.13) is therefore:

$$\begin{aligned}
P(O|\lambda) &= \sum_{i=1}^{N} \alpha_T(i) \\
&= \sum_{i=1}^{3} \alpha_3(i) \\
&= 259/20250 + 152/14261 + 152/14261 \\
&= 313/9177 = 0.0341
\end{aligned}$$

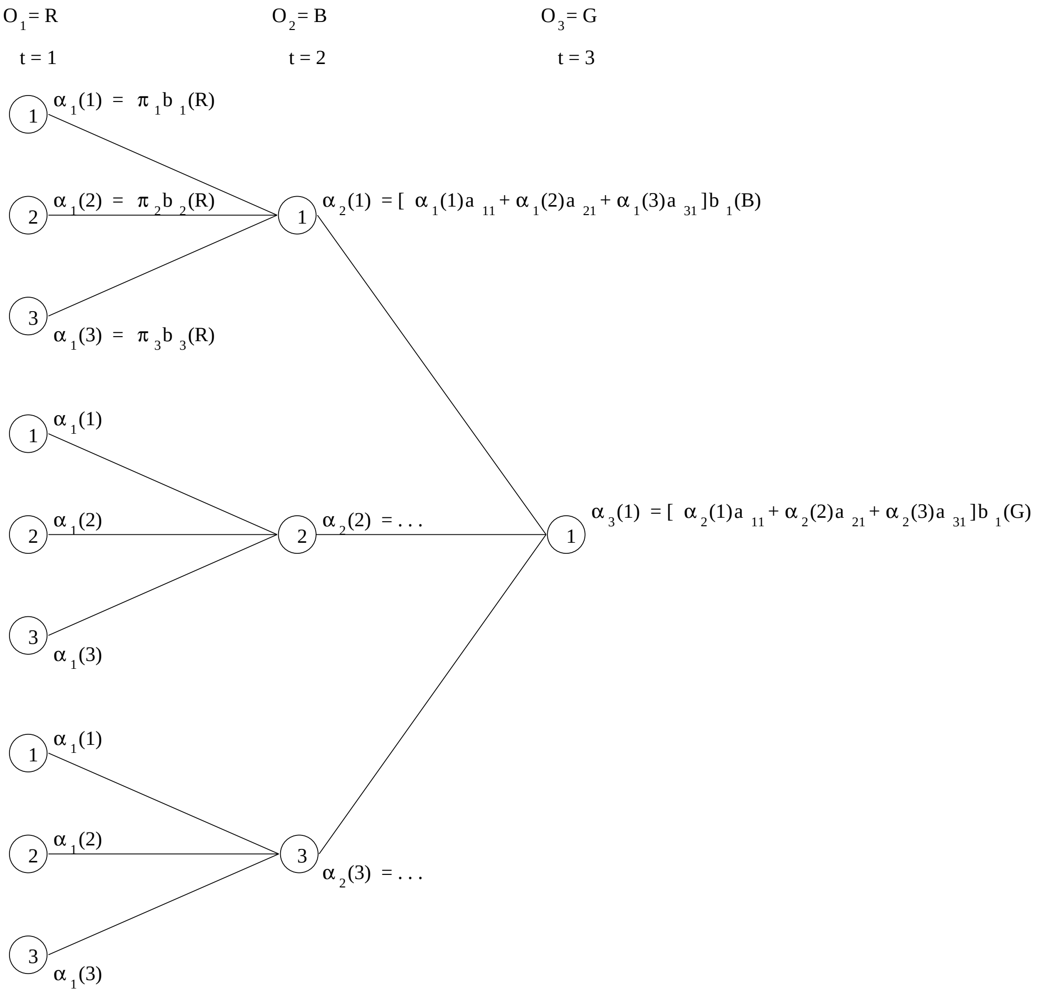


Figure C.4: A graphical representation of the calculations for $\alpha_3(1)$.

By looking at the calculations in the example above we can see that the initialisation step takes 1 multiplication for each state i, where $1 \le i \le N$, resulting in the order of N calculations. The induction step takes $N-1$ additions and $N+1$ multiplications for each state i and time tick t, where $1 \le i \le N$ and $1 \le t \le T-1$, resulting in the order of $(N-1+N+1)N(T-1) = 2NN(T-1) = 2N^2(T-1)$ calculations. The asymptotic complete running time for the Forward-Backward algorithm will thus be in the order of $\mathcal{O}(N^2T)$, which is much better than the running time for equation (C.8) of $\mathcal{O}(TN^T)$.

C.2.3 The Backward-Forward Algorithm

The Backward-Forward algorithm is quite like the Forward-Backward presented in section C.2.2 on page 112. The difference is that we start out by looking at the last symbol O_T in the observation sequence instead of the first symbol O_1. For the Backward-Forward algorithm we define the backward variable as:

$$\beta_t(i) = P(O_{t+1}O_{t+2}\cdots O_T|q_t = S_i, \lambda), \qquad 1 \le i \le N \tag{C.16}$$

The probability of having observed the sequence $O_{t+1}O_{t+2}\cdots O_T$ given we are in state S_i at time t and the model λ.

The algorithm is defined by the following two steps:

step 1: Initialisation.

$$\beta_T(i) = 1, \qquad 1 \le i \le N \tag{C.17}$$

step 2: Induction.

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij}b_j(O_{t+1})\beta_{t+1}(j), \qquad 1 \le i \le N$$
$$t = T-1, T-2, \cdots, 1 \tag{C.18}$$

In step 1 we *arbitrarily* define $\beta_T(i)$ to have the value 1. The argumentation for doing this is not clear in neither [28] nor [29], and looking at the definition for $\beta_t(i)$ in equation (C.16) for $t = T$ results in the probability of observing an empty sequence of symbols given we are in S_i at time T: $\beta_T(i) = P(\emptyset|q_T = S_i, \lambda)$, which is clearly not 1. Anyhow, with the initial value of the backward variable for every state S_i set to 1, the induction part of the algorithm will always give an useful result, whereas an initial value of 0 would always result in $\beta_t(i)$ becoming 0. Step 2 of the algorithm represents that to be in state S_i at time t and further observe the symbol sequence $O = O_{t+1}O_{t+2}\cdots O_T$, you have to observe the symbol O_{t+1} in one of the $S_1, S_2, \cdots, S_N$ states at time $t+1$, making a transition from state S_i to the state concerned and taking into account the rest of the observation sequence $O_{t+2}O_{t+3}\cdots O_T$ represented by the $\beta_{t+1}(j)$ term in equation (C.18). The induction step of the Backward-Forward algorithm is illustrated graphically in figure C.5 on the following page.

To fix the idea of the Backward-Forward algorithm we take an example of calculating the backward probabilities for the HMM presented in section C.1.3 on page 110, given the observation sequence $O = RBG$.

First we initialise the Backward-Forward algorithm using equation (C.17).

$$\begin{aligned}\beta_3(1) &= 1\\ \beta_3(2) &= 1\\ \beta_3(3) &= 1\end{aligned}$$

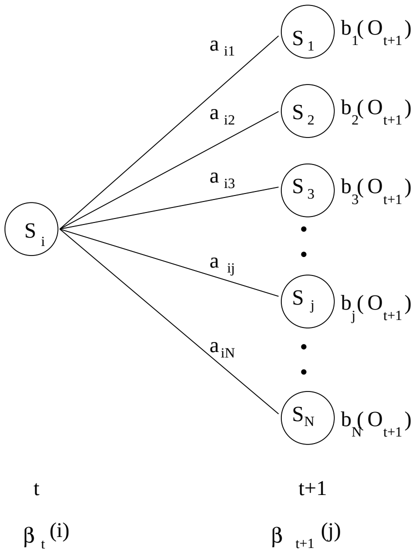


Figure C.5: Step 2 of the Backward-Forward algorithm. The algorithm reuses the backward variables $\beta_{t+1}(j)$ when finding $\beta_t(i)$.

We then use equation (C.18) for the inductive part of the Backward-Forward algorithm.

$$
\begin{aligned}
\beta_2(1) &= a_{11}b_1(2)\beta_3(1) + a_{12}b_2(2)\beta_3(2) + a_{13}b_3(2)\beta_3(3) \\
&= 1/3 \times 2/5 \times 1 + 1/3 \times 2/6 \times 1 + 1/3 \times 2/6 \times 1 \\
&= 2/15 + 2/18 + 2/18 = 16/45 \\
\beta_2(2) &= a_{21}b_1(2)\beta_3(1) + a_{22}b_2(2)\beta_3(2) + a_{23}b_3(2)\beta_3(3) \\
&= 1/3 \times 2/5 \times 1 + 1/3 \times 2/6 \times 1 + 1/3 \times 2/6 \times 1 \\
&= 2/15 + 2/18 + 2/18 = 16/45 \\
\beta_2(3) &= a_{31}b_1(2)\beta_3(1) + a_{32}b_2(2)\beta_3(2) + a_{33}b_3(2)\beta_3(3) \\
&= 1/3 \times 2/5 \times 1 + 1/3 \times 2/6 \times 1 + 1/3 \times 2/6 \times 1 \\
&= 2/15 + 2/18 + 2/18 = 16/45
\end{aligned}
$$

$$\begin{aligned}
\beta_1(1) &= a_{11}b_1(3)\beta_2(1) + a_{12}b_2(3)\beta_2(2) + a_{13}b_3(3)\beta_2(3) \\
&= 1/3 \times 1/5 \times 16/45 + 1/3 \times 0/6 \times 16/45 + 1/3 \times 3/6 \times 16/45 \\
&= 16/675 + 0 + 8/135 = 56/675 \\
\beta_1(2) &= a_{21}b_1(3)\beta_2(1) + a_{22}b_2(3)\beta_2(2) + a_{23}b_3(3)\beta_2(3) \\
&= 1/3 \times 1/5 \times 16/45 + 1/3 \times 0/6 \times 16/45 + 1/3 \times 3/6 \times 16/45 \\
&= 16/675 + 0 + 8/135 = 56/675 \\
\beta_1(3) &= a_{31}b_1(3)\beta_2(1) + a_{32}b_2(3)\beta_2(2) + a_{33}b_3(3)\beta_2(3) \\
&= 1/3 \times 1/5 \times 16/45 + 1/3 \times 0/6 \times 16/45 + 1/3 \times 3/6 \times 16/45 \\
&= 16/675 + 0 + 8/135 = 56/675
\end{aligned}$$

When calculating $\beta_1(i)$ we use $b_j(3)$, because $b_j(O_{t+1}) = b_j(O_{1+1}) = b_j(O_2)$ and the second observation symbol $O_2 = B$ is found at index 3 in the set of symbols $V = \{R, G, B\}$.

If we take a closer look at the Backward-Forward algorithm, we can see that the induction step will require in the order of $2N$ multiplications and $N-1$ additions for every state i and time tick t, where $1 \le i \le N$ and $t = T-1, T-2, \cdots, 1$, resulting in the order of $(2N+N-1)N(T-1) = (3N-1)N(T-1)$ calculations or a complete asymptotic running time of $\mathcal{O}(N^2T)$ as with the Forward-Backward algorithm presented in section C.2.2 on page 112.

We will see how the Backward-Forward algorithm will be used later on in section C.2.4 and C.2.5 on page 123 to assist in calculating the probability $P(q_t = S_i|O,\lambda)$ and optimising the parameters of the HMM.

C.2.4 The Viterbi Algorithm

The Viterbi algorithm finds the *best* state sequence Q^*, in the HMM λ given an observation sequence O. With the *best* state sequence we understand the state sequence which is most likely to occur, given the observation sequence O.

One method of doing this, is to find the the probability of being in state S_i at time t given the symbol sequence O and the model λ:

$$\gamma_t(i) = P(q_t = S_i|O,\lambda), \qquad \begin{array}{l} 1 \le i \le N \\ 1 \le t \le T \end{array} \tag{C.19}$$

Then for all the states at each time tick t, return the state which results in the highest probability of $P(q_t = S_i|O,\lambda)$:

$$q_t = \underset{1 \le i \le N}{\operatorname{argmax}}(\gamma_t(i)), \qquad 1 \le t \le T \tag{C.20}$$

This will give us the most likely state to be in, at every time tick t, for the observation sequence O. In equation (C.20) the function

$$\underset{\forall i}{\operatorname{argmax}}(\exp(i)) \tag{C.21}$$

will return the argument i, which results in the maximal value of $\exp(i)$.

Then, how do we find the probability given in equation (C.19)? Well, we use the forward and backward variables as defined in section C.2.2 on page 112 and C.2.3 on page 117.

$$\begin{aligned}\gamma_t(i) &= \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} &\text{(C.22)}\\ &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N}\alpha_t(i)\beta_t(i)} &\text{(C.23)}\end{aligned}$$

This is true because the forward variable will account for the joint probability of observing the symbols $O_1O_2\cdots O_t$ and being in state S_i at time t and the backward variable will account for the probability of observing the symbols $O_{t+1}O_{t+2}\cdots O_T$ given we are in state S_i at time t. Furthermore we divide with $P(O|\lambda) = \sum_{i=1}^{N}\alpha_t(i)\beta_t(i)$ to normalise the probability such that: $\sum_{i=1}^{N}\gamma_i(t) = 1$.

Unfortunately this method does not take into account how likely the states are to occur after each other, e.g. it could be very likely to be in state S_i at time t and very likely to be in state S_j at $t+1$, but impossible to do a transition from state S_i to S_j, because a_{ij} is zero. The method only determines the most likely state at every time instant t not taking into account the probability of the actual state sequence. In this way, the above mentioned method is able to return a state sequence which is simply not possible. So the method for finding the best state sequence given in equation (C.20) is not so good after all.

Instead we introduce the Viterbi algorithm, which goes through the observation symbols one by one, choosing the most likely state S_i at time t to observe O_t in, and taking into account how likely it is to end up in state S_i according to the state transitions or the initial probability distribution.

step 1: Initialisation

$$\begin{aligned}\delta_1(i) &= \pi_i b_i(O_1), \qquad 1 \le i \le N &\text{(C.24)}\\ \psi_1(i) &= 0 &\text{(C.25)}\end{aligned}$$

step 2: Induction

$$\delta_t(j) = \max_{1\le i\le N}[\delta_{t-1}(i)a_{ij}]b_j(O_t), \qquad 1 \le j \le N, \quad 2 \le t \le T \qquad \text{(C.26)}$$

$$\psi_t(j) = \operatorname*{argmax}_{1\le i\le N}[\delta_{t-1}(i)a_{ij}], \qquad 1 \le j \le N, \quad 2 \le t \le T \qquad \text{(C.27)}$$

In the above equations, $\delta_t(j)$ will give us the probability of the best state sequence of length t ending in state j, whereas $\psi_t(j)$ will keep track of the best states chosen, so we can extract the state sequence when the algorithm is done. To find the probability of the best state sequence for an observation sequence $O = O_1O_2 \cdots O_T$, and extract the sequence afterwards, we use the following equations:

$$\begin{aligned} P^* &= \max_{1 \leq i \leq N}[\delta_T(i)] && \text{(C.28)} \\ q_T^* &= \underset{1 \leq i \leq N}{\operatorname{argmax}}[\delta_T(i)] && \text{(C.29)} \\ q_t^* &= \psi_{t+1}(q_{t+1}^*) && \text{(C.30)} \end{aligned}$$

The best state sequence $Q^* = q_1^*q_2^* \cdots q_T^*$ is extracted by first finding q_T^* from equation (C.29) and then tracking backwards using equation (C.30).

What is important to notice, as with the Forward-Backward and Backward-Forward algorithms, is that we are able to reuse $\delta_{t-1}(i)$ when finding $\delta_t(j)$ and $\psi_t(j)$, bringing the necessary calculations to a minimum. To illustrate this we will do a small example, finding the best state sequence for the observation sequence $O = RBG$, using the HMM presented in section C.1.3 on page 110.

We start out by initialising the Viterbi algorithm using equation (C.24) and (C.25). There are three states in the model, so we need to calculate:

$$\begin{aligned} \delta_1(1) &= \pi_1 b_1(1) & \psi_1(1) &= 0 \\ &= 1/3 \times 2/5 \\ &= 2/15 \\ \delta_1(2) &= \pi_2 b_2(1) & \psi_1(2) &= 0 \\ &= 1/3 \times 4/6 \\ &= 4/18 = 2/9 \\ \delta_1(3) &= \pi_3 b_3(1) & \psi_1(3) &= 0 \\ &= 1/3 \times 1/6 \\ &= 1/18 \end{aligned}$$

Using equation (C.26) and (C.27) we find that:

$$\begin{aligned} \delta_2(1) &= \max[\delta_1(1)a_{11}, \delta_1(2)a_{21}, \delta_1(3)a_{31}]b_1(3) \\ &= \max[2/15 \times 1/3, 2/9 \times 1/3, 1/18 \times 1/3] \times 1/5 \\ &= 2/9 \times 1/3 \times 1/5 = 4/270 = 2/135 \\ \psi_2(1) &= \operatorname{argmax}[\delta_1(1)a_{11}, \delta_1(2)a_{21}, \delta_1(3)a_{31}] \\ &= \operatorname{argmax}[2/15 \times 1/3, 2/9 \times 1/3, 1/18 \times 1/3] = 2 \end{aligned}$$

$$\begin{aligned} \delta_2(2) &= \max[\delta_1(1)a_{12}, \delta_1(2)a_{22}, \delta_1(3)a_{32}]b_2(3) \\ &= \max[2/15 \times 1/3, 2/9 \times 1/3, 1/18 \times 1/3] \times 0 \\ &= 2/9 \times 1/3 \times 0 = 0 \\ \psi_2(2) &= \operatorname{argmax}[\delta_1(1)a_{12}, \delta_1(2)a_{22}, \delta_1(3)a_{32}] \\ &= \operatorname{argmax}[2/15 \times 1/3, 2/9 \times 1/3, 1/18 \times 1/3] = 2 \end{aligned}$$

$$\begin{aligned}
\delta_2(3) &= \max[\delta_1(1)a_{13}, \delta_1(2)a_{23}, \delta_1(3)a_{33}]b_3(3) \\
&= \max[2/15 \times 1/3, 2/9 \times 1/3, 1/18 \times 1/3] \times 3/6 \\
&= 2/9 \times 1/3 \times 3/6 = 6/162 = 1/27 \\
\psi_2(3) &= \text{argmax}[\delta_1(1)a_{13}, \delta_1(2)a_{23}, \delta_1(3)a_{33}] \\
&= \text{argmax}[2/15 \times 1/3, 2/9 \times 1/3, 1/18 \times 1/3] = 2
\end{aligned}$$

reusing the values of $\delta_1(1)$, $\delta_1(2)$ and $\delta_1(3)$.

And again from equation (C.26) and (C.27) we find that:

$$\begin{aligned}
\delta_3(1) &= \max[\delta_2(1)a_{11}, \delta_2(2)a_{21}, \delta_2(3)a_{31}]b_1(2) \\
&= \max[2/135 \times 1/3, 0 \times 1/3, 1/27 \times 1/3] \times 2/5 \\
&= 1/27 \times 1/3 \times 2/5 = 2/405 \\
\psi_3(1) &= \text{argmax}[\delta_2(1)a_{11}, \delta_2(2)a_{21}, \delta_2(3)a_{31}] \\
&= \max[2/135 \times 1/3, 0 \times 1/3, 1/27 \times 1/3] = 3
\end{aligned}$$

$$\begin{aligned}
\delta_3(2) &= \max[\delta_2(1)a_{12}, \delta_2(2)a_{22}, \delta_2(3)a_{32}]b_2(2) \\
&= \max[2/135 \times 1/3, 0 \times 1/3, 1/27 \times 1/3] \times 2/6 \\
&= 1/27 \times 1/3 \times 2/6 = 2/486 = 1/243 \\
\psi_3(2) &= \text{argmax}[\delta_2(1)a_{12}, \delta_2(2)a_{22}, \delta_2(3)a_{32}] \\
&= \max[2/135 \times 1/3, 0 \times 1/3, 1/27 \times 1/3] = 3
\end{aligned}$$

$$\begin{aligned}
\delta_3(3) &= \max[\delta_2(1)a_{13}, \delta_2(2)a_{23}, \delta_2(3)a_{33}]b_3(2) \\
&= \max[2/135 \times 1/3, 0 \times 1/3, 1/27 \times 1/3] \times 2/6 \\
&= 1/27 \times 1/3 \times 2/6 = 2/486 = 1/243 \\
\psi_3(3) &= \text{argmax}[\delta_2(1)a_{13}, \delta_2(2)a_{23}, \delta_2(3)a_{33}] \\
&= \max[2/135 \times 1/3, 0 \times 1/3, 1/27 \times 1/3] = 3
\end{aligned}$$

reusing the values of $\delta_2(1)$, $\delta_2(2)$ and $\delta_2(3)$.

So the best state sequence for the observation $O = RBG$ of length $T = 3$ can now be found using equations (C.29) and (C.30):

$$\begin{aligned}
q_3^* &= \underset{1 \le i \le N}{\text{argmax}}[\delta_3(i)] \\
&= \text{argmax}[\delta_3(1), \delta_3(2), \delta_3(3)] \\
&= \text{argmax}[2/405, 1/243, 1/243] = 1 \\
q_2^* &= \psi_3(q_3^*) \\
&= \psi_3(1) \\
&= 3 \\
q_1^* &= \psi_2(q_2^*) \\
&= \psi_2(3) \\
&= 2
\end{aligned}$$

Resulting in the best state sequence $Q^* = q_1^* q_2^* q_3^* = 231$, which is clearly the best state sequence for observing $O = RBG$ when looking at the HMM in figure C.2 on page 109. As we can see, the calculations of the Viterbi algorithm are quite similar to the calculations of the Forward-Backward algorithm presented in section C.2.2 on page 112. The only difference is that we maximise the terms in the Viterbi algorithm, whereas we sum up the terms in the Forward-Backward algorithm. Figure C.4 on page 116 shows a graphical representation for finding the variable $\alpha_3(1)$ given the HMM in section C.1.3 on page 110 and the observation $O = RBG$, this figure could also represent the finding of $\delta_3(1)$, if all α variables were substituted with δ's and instead of doing a summation when finding $\alpha_2(1)$ and $\alpha_3(1)$ we should do a maximisation when finding $\delta_2(1)$ and $\delta_3(1)$.

If we look at the initialisation part of the Viterbi algorithm, equation (C.24) and (C.25), we can see that it requires in the order of N multiplications. The induction part of the algorithm, equation (C.26) and (C.27), requires N multiplications for the $\delta_{t-1}(i)a_{ij}$ term, 1 multiplication by the $b_j(O_t)$ term, 1 procedure call to max and 1 procedure call to argmax. In equation (C.27) we simply reuse the $\delta_{t-1}(i)a_{ij}$ term already calculated in equation (C.26), thus resulting in only N multiplications and not $2N$ multiplications – no need to calculate the $\delta_{t-1}(i)a_{ij}$ term twice. If we estimate the procedure call of max and argmax to take in the order of N calculations when given N arguments, the induction step of the Viterbi algorithm will have a complexity of $N+1+N+N = 3N+1$. The induction step will be run for every state i and time tick t, where $1 \leq i \leq N$ and $2 \leq t \leq T$, thus resulting in a complexity of $(3N+1)(N(T-1))$. The induction step will together with the initialisation part take in the order of $(3N+1)(N(T-1)) + N$ calculations, thus resulting in a complete asymptotic running time of $\mathcal{O}(N^2T)$.

C.2.5 The Baum-Welch Algorithm

An always recurrent problem with HMMs is to adjust the parameters A, B and π to maximise the probability of observing a certain sequence of symbols. This is quite a hard problem and no analytic solutions have been given yet. Though, in contrast to an analytic solution, there are a few algorithms which are able to optimise the parameters iteratively; one of them is known as the Baum-Welch algorithm. The idea here is to keep on training the HMM, to maximise the probability of observing a certain symbol sequence O, until a certain threshold has been reached.

Before taking a look at the Baum-Welch algorithm, we introduce the joint probability of being in state S_i at time t and in state S_j at time $t+1$ given the observation sequence O and the model λ:

$$\xi_t(i,j) = P(q_t = S_i, q_{t+1} = S_j | O, \lambda), \qquad \begin{array}{l} 1 \leq i \leq N \\ 1 \leq j \leq N \end{array} \tag{C.31}$$

The event of being in state S_i at time t and in state S_j at time $t+1$ could be illustrated graphically; see figure C.6.

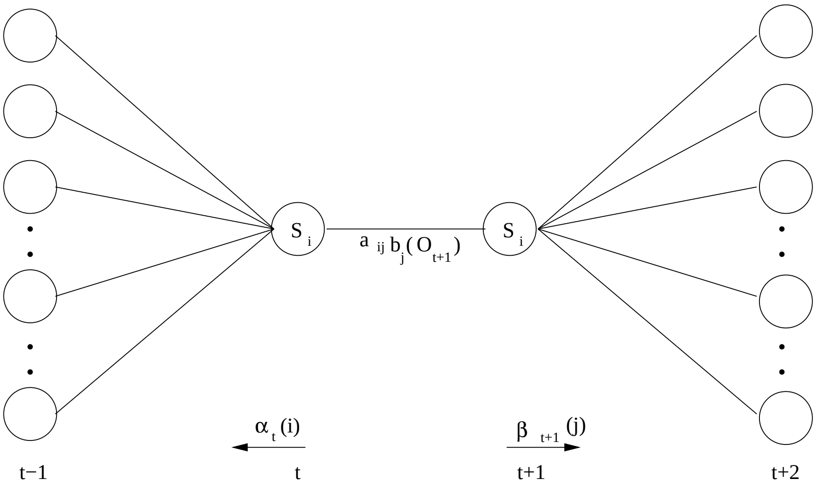


Figure C.6: Graphically representation of the joint event of being in state S_i at time t and in state S_j at time $t+1$.

From figure C.6 it is clear to see, that the probability given in equation (C.31) could be written as:

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)}, \qquad 1 \le i \le N$$
$$1 \le j \le N \qquad \text{(C.32)}$$

To be in state S_i at time t we need to have observed the symbol sequence $O = O_1 O_2 \cdots O_t$; thus the $\alpha_t(i)$ term. To be in state S_j at time $t+1$, coming from state S_i and observing the symbol O_{t+1} in S_j, we must make a transition from state S_i to S_j and account for the probability of O_{t+1} in S_j; thus the term $a_{ij}b_j(O_{t+1})$. We should also take into account observing the rest of the symbol sequence $O_{t+2}O_{t+3} \cdots O_T$, given we are in state S_j at time $t+1$; thus the term $\beta_{t+1}(j)$. Finally we divide with the probability $P(O|\lambda)$, to achieve a proper normalisation value for $\xi_t(i,j)$.

If we take a closer look at $\xi_t(i,j)$ summing it up for all j, we get the following result:

$$\sum_{j=1}^{N} \xi_t(i,j) = \sum_{j=1}^{N} \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)} \qquad \text{(C.33)}$$

$$= \left[\sum_{j=1}^{N} a_{ij}b_j(O_{t+1})\beta_{t+1}(j)\right] \frac{\alpha_t(i)}{P(O|\lambda)} \qquad \text{(C.34)}$$

$$\overset{\text{by def. of } \beta_t(i)}{=} \frac{\beta_t(i)\alpha_t(i)}{P(O|\lambda)} \qquad \text{(C.35)}$$

which in fact is equal to the variable $\gamma_t(i)$ as defined in equation (C.22) on page 120, resulting in the following equation:

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j) \tag{C.36}$$

As mentioned in section C.2.4 on page 119, $\gamma_t(i)$ is the probability of being in state S_i at time t. This means that:

$$\begin{aligned}
\gamma_1(i) &= \text{the probability of being in state } S_i \text{ at time } t = 1 \\
\gamma_2(i) &= \text{the probability of being in state } S_i \text{ at time } t = 2 \\
\cdots &= \cdots \\
\gamma_T(i) &= \text{the probability of being in state } S_i \text{ at time } t = T
\end{aligned}$$

And summing up all these probabilities over time gives us a measure, which can be interpreted as the number of times we have visited state S_i. Or by excluding $t = T$ from the sum, a measure, which can be interpreted as the number of transition made from state S_i.

$$\sum_{t=1}^{T-1} \gamma_t(i) = \text{Expected number of transitions made from state } S_i \tag{C.37}$$

As mentioned before $\xi_(i,j)$ is the probability of being in state S_i at time t and in state S_j at time $t+1$, thus making a transition from S_i to S_j. The summation of $\xi_t(i,j)$ over time from t=1 to t=T-1, can therefore be interpreted as the number of transitions made from state S_i to S_j.

$$\sum_{t=1}^{T-1} \xi_t(i,j) = \text{Expected number of transitions from state } S_i \text{ to state } S_j \tag{C.38}$$

We can now reestimate the parameters of the HMM $\lambda = (A, B, \pi)$ by the parameters $\overline{A}$, $\overline{B}$ and $\overline{\pi}$ given by:

$$\overline{\pi}_i = \text{the probability of being in } S_i \text{ at time } 1 \ = \gamma_1(i) \tag{C.39}$$

$$\begin{aligned} \overline{a}_{ij} &= \frac{\text{Expected number of transitions from state } S_i \text{ to state } S_j}{\text{Expected number of transitions out of state } S_i} \\ &= \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \end{aligned} \tag{C.40}$$

$$\begin{aligned} \bar{b}_j(k) &= \frac{\text{Expected number of times in } S_j \text{ observing symbol } v_k}{\text{Expected number of times in } S_j} \\ &= \frac{\sum_{\substack{t=1 \\ O_t=v_k}}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \end{aligned} \tag{C.41}$$

In the re-estimation of $\overline{a}_{ij}$ we need the probability of doing just *one* transition from S_i to S_j, the sum $\sum_{t=1}^{T-1} \xi_t(i,j)$ gives us the probability of *all* the expected transitions, so we need to divide by the number of transitions out of state S_i. The same applies for the re-estimation of $\bar{b}_j(k)$, where we divide by the number of times in state S_j.

If we are given a HMM $\lambda = (A, B, \pi)$ and its re-estimated model $\overline{\lambda} = (\overline{A}, \overline{B}, \overline{\pi})$, where $\overline{A}$, $\overline{B}$ and $\overline{\pi}$ are calculated as described above, it can be proven that either the two models are the same, $\lambda = \overline{\lambda}$, or the probability of observing the symbol sequence O in $\overline{\lambda}$ is greater then observing it in λ, $P(O|\overline{\lambda}) > P(O|\lambda)$. This enable us to reestimate the HMM λ iteratively until a preferred threshold μ has been reached:

step 1: Given the HMM λ, find the re-estimated model $\overline{\lambda}$ using equation (C.39), (C.40) and (C.41).

step 2: If $\mu > P(O|\overline{\lambda}) - P(O|\lambda)$ then terminate the algorithm and return $\overline{\lambda}$ otherwise set $\lambda = \overline{\lambda}$ and go to step 1.

If we look at the re-estimation of the initial probability distribution $\overline{\pi}$, we can see from equation (C.22) that it is defined by:

$$\gamma_1(i) = \frac{\alpha_1(i)\beta_1(i)}{P(O|\lambda)} \tag{C.42}$$

From equation (C.13) we know that $P(O|\lambda)$ can be expressed as $\sum_{i=1}^{N} \alpha_T(i)$ giving us:

$$\gamma_1(i) = \frac{\alpha_1(i)\beta_1(i)}{\sum_{i=1}^{N} \alpha_T(i)} \tag{C.43}$$

The $\alpha_1(i)$ term requires 1 multiplication for every i, see equation (C.11) on page 113, whereas the $\beta_1(i)$ term will require $(3N-1)(T-1)$ calculations for every i, see equation (C.18) on page 117. For every i, $1 \leq i \leq N$, we need to find a new numerator of equation (C.43), resulting in $(1+(3N-1)(T-1))N$ calculations. The denominator only needs to be calculated once, this requires 1 multiplication for the initialisation part of $\alpha_T(i)$, $N+1$ multiplications plus $N-1$ additions for the induction part of $\alpha_T(i)$, see section C.2.2 on page 116, and we need to find $\alpha_T(i)$ for all i, where $1 \leq i \leq N$, giving the denominator in the order of $(1+N+1+N-1)N = (2N+1)N$ calculations. If we add the required calculations for the numerator and denominator of equation (C.43) together, we get in the order of $(1+(3N-1)(T-1))N+(2N+1)N$ calculations or an asymptotic running time of $\mathcal{O}(N^2T)$.

If we look at the re-estimation of the state transition probability distribution in terms of the forward and backward variables, we can see from equation (C.36) and (C.32) that $\overline{a}_{ij}$ is defined by:

$$\overline{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{C.44}$$

$$= \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \xi_t(i,j)} \tag{C.45}$$

$$= \frac{\sum_{t=1}^{T-1} \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)}}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)}} \tag{C.46}$$

$$= \frac{\frac{1}{P(O|\lambda)} \sum_{t=1}^{T-1} \alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\frac{1}{P(O|\lambda)} \sum_{t=1}^{T-1} \sum_{j=1}^{N} \alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)} \tag{C.47}$$

$$= \frac{\sum_{t=1}^{T-1} \alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)} \tag{C.48}$$

When finding the re-estimations of the state transition probabilities, we first calculate all forward and backward probabilities, taking in the order of $(2N+1)N(T-1)$ and $(3N-1)N(T-1)$ calculations respectively; see section C.2.2 and C.2.3. We then need to do 3 multiplications to find the expression of $\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)$ in the numerator of equation (C.48) for every i, j and t, where $1 \leq i \leq N$, $1 \leq j \leq N$ and $1 \leq t \leq T-1$, taking in the order of $3N^2(T-1)$ calculations. When finding the denominator of equation (C.48), we just reuse the calculated value of $\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)$ in the numerator and sum up all the values for each j, resulting in i additions for every j, giving us N^2 calculations. Finally we should divide the nominator with the denominator for every i and j, resulting in N^2 divisions. The complete running time for the re-estimation of the state transition probabilities therefore results in $(2N+$

$1)N(T-1)+(3N-1)N(T-1)+3N^2(T-1)+N^2+N^2 = (T-1)(8N^2)+2N^2$ calculations or an upper asymptotic running time of $\mathcal{O}(TN^2)$.

When re-estimating the observation sequence probability distributions $b_j(k)$, we first need to find the gamma values $\gamma_t(j)$ for every t and j, where $1 \leq j \leq N$ and $1 \leq t \leq T$.

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\sum_{i=1}^{N} \alpha_T(i)}$$

This takes $(2N+1)N(T-1)$ calculations for the $\alpha_t(j)$ term, $(3N-1)N(T-1)$ calculations for the $\beta_t(j)$ term, N additions for the $\sum_{i=1}^{N} \alpha_T(i)$ term, and $2TN$ calculations when multiplying $\alpha_t(j)\beta_t(j)$ and dividing with $\sum_{i=1} \alpha_T(i)$ for every t and j; thus resulting in $(2N+1)N(T-1)+(3N-1)N(T-1)+N+2TN = 5N^2(T-1)+N(1+2T)$ calculations or an upper asymptotic running time of $\mathcal{O}(TN^2)$ for finding $\gamma_t(j)$. Secondly we need to go trough all the observation symbols O_t for every state S_j and when $v_k = O_t$ we should add up the value of $\gamma_t(j)$ to $b_j(k)$. To normalise the values of every $b_j(k)$ we divide by the $\sum_{t=1}^{T} \gamma_t(j)$ term. It takes NT additions for calculating all the $\sum_{\substack{t=1 \\ O_t=v_k}}^{T} \gamma_t(j)$ terms for every t and j, NT additions when finding $\sum_{t=1}^{T} \gamma_t(j)$ for every t and j, and NT divisions when doing the normalisation of $b_j(k)$. All this together with the computing of the gamma values result in $3NT+5N^2(T-1)+N(1+2T) = 5N^2(T-1)+N(1+5T)$ calculations or an upper asymptotic running time of $\mathcal{O}(TN^2)$.

We should notice that the computing of the alpha, beta and gamma values, only need to be calculated one time and we are in such way able to spare some calculations when re-estimating pi_i, a_{ij} and $b_j(k)$ all at once. Though, the upper asymptotic running time for re-estimating pi_i, a_{ij} and $b_j(k)$ at the same time, will still be $\mathcal{O}(TN^2)$.

C.3 Implementation Issues for Hidden Markov Models

We will in this section describe some solutions to some of the implementations issues that one might run in to when implementing the algorithms for HMMs. We will especially look at how we represent the small probabilities found by the Forward-Backward and Backward-Forward algorithms within the boundary of a double precision value, how to train a HMM for multiple observation sequences, and how to initialise the parameters of the HMM.

C.3.1 Scaling

When calculating the forward and backward probabilities for long observation sequences, we often run into problems of representing the forward and backward probabilities within the normal range of double precision values in the computer. To cope with these problems, we now introduce a way of scaling the forward and backward probabilities, such that we are able to calculate the probabilities within normal double precision values in the computer.

Scaling the Forward Probabilities

The new scaled version of the Forward-Backward algorithm uses a scaling coefficient denoted c_t for every time step t or observation symbol O_t. The first scaling coefficient is defined by:

$$c_1 = \frac{1}{\sum_{i=1}^{N} \alpha_1(i)}, \tag{C.49}$$

whereas the rest of the scaling coefficients are defined as:

$$c_t = \frac{1}{\sum_{i=1}^{N} \hat{\hat{\alpha}}_t(i)}. \tag{C.50}$$

The notation of $\hat{\hat{\alpha}}_t(i)$ denote that the forward probability is still not scaled for time t, but for every $t-1, t-2, \cdots, 1$, whereas the notation $\hat{\alpha}_t(i)$ denote that the probability is scaled for every t. Non-scaled forward probabilities will just be denote $\alpha_t(i)$ as before.

With the new notation in our hands, we are now able to define the scaled Forward-Backward algorithm:

Step 1: Initialisation.

$$c_1 = \frac{1}{\sum_{j=1}^{N} \alpha_1(j)} \tag{C.51}$$

$$\hat{\alpha}_1(i) = c_1 \alpha_1(i), \qquad 1 \le i \le N \tag{C.52}$$

Step 2: Induction.

$$\hat{\hat{\alpha}}_{t+1}(j) = \left[\sum_{i=1}^{N} \hat{\alpha}_t(i) a_{ij} \right] b_j(O_{t+1}), \qquad 1 \le j \le N, \quad 1 \le t \le T-1 \tag{C.53}$$

$$c_{t+1} = \frac{1}{\sum_{i=1}^{N} \hat{\hat{\alpha}}_{t+1}(i)} \tag{C.54}$$

$$\hat{\alpha}_{t+1}(j) = c_{t+1} \hat{\hat{\alpha}}_{t+1}(j), \qquad 1 \le j \le N, \quad 1 \le t \le T-1 \tag{C.55}$$

To illustrate how the algorithm works, we take an example using the HMM presented in section C.1.3 on page 110. First we calculate $\alpha_1(i)$ as normal by the initialisation step given in equation (C.11) on page 113:

$$\begin{aligned}
\alpha_1(1) &= \pi_1 b_1(1) = 1/3 \times 2/5 = 2/15 \\
\alpha_1(2) &= \pi_2 b_2(1) = 1/3 \times 4/6 = 4/18 \\
\alpha_1(3) &= \pi_3 b_3(1) = 1/3 \times 1/6 = 1/18,
\end{aligned}$$

then we calculate the scaling coefficient as the sum of all $\alpha_1(i)$ for every state i:

$$\begin{aligned}
c_1 &= \frac{1}{\sum_{i=1}^{N} \alpha_1(i)} \\
&= \frac{1}{2/15 + 4/18 + 1/18} = \frac{1}{37/90},
\end{aligned}$$

and finally we scale all the forward values using (C.52):

$$\begin{aligned}
\hat{\alpha}_1(1) &= \frac{2/15}{37/90} = 12/37 \\
\hat{\alpha}_1(2) &= \frac{4/18}{37/90} = 20/37 \\
\hat{\alpha}_1(3) &= \frac{1/18}{37/90} = 5/37.
\end{aligned}$$

To calculate the $\hat{\alpha}_2(i)$ values we first use the definition of $\hat{\hat{\alpha}}_{t+1}(i)$ as given in (C.53):

$$\begin{aligned}
\hat{\hat{\alpha}}_2(1) &= [\hat{\alpha}_1(1)a_{11} + \hat{\alpha}_1(2)a_{21} + \hat{\alpha}_1(3)a_{31}]b_1(3) \\
&= [12/37 \times 1/3 + 20/37 \times 1/3 + 5/37 \times 1/3] \times 1/5 = 1/15 \\
\hat{\hat{\alpha}}_2(2) &= [\hat{\alpha}_1(1)a_{12} + \hat{\alpha}_1(2)a_{22} + \hat{\alpha}_1(3)a_{32}]b_2(3) \\
&= [12/37 \times 1/3 + 20/37 \times 1/3 + 5/37 \times 1/3] \times 0/6 = 0 \\
\hat{\hat{\alpha}}_2(3) &= [\hat{\alpha}_1(1)a_{13} + \hat{\alpha}_1(2)a_{23} + \hat{\alpha}_1(3)a_{33}]b_3(3) \\
&= [12/37 \times 1/3 + 20/37 \times 1/3 + 5/37 \times 1/3] \times 3/6 = 1/6,
\end{aligned}$$

then we calculate the scaling value c_2:

$$c_2 = \frac{1}{\hat{\hat{\alpha}}_2(1) + \hat{\hat{\alpha}}_2(2) + \hat{\hat{\alpha}}_2(3)} = \frac{1}{1/15 + 0 + 1/6} = \frac{1}{7/30},$$

and finally use equation (C.55) to find the scaled forward values:

$$\begin{aligned}
\hat{\alpha}_2(1) &= \frac{1/15}{7/30} = 2/7 \\
\hat{\alpha}_2(2) &= \frac{0}{7/30} = 0 \\
\hat{\alpha}_2(3) &= \frac{1/6}{7/30} = 5/7.
\end{aligned}$$

To find the values of $\hat{\alpha}_3(i)$ we do the following calculations:

$$\begin{aligned}
\hat{\hat{\alpha}}_3(1) &= [\hat{\alpha}_2(1)a_{11} + \hat{\alpha}_2(2)a_{21} + \hat{\alpha}_2(3)a_{31}]b_1(2) \\
&= [2/7 \times 1/3 + 0 \times 1/3 + 5/7 \times 1/3] \times 2/5 = 2/15 \\
\hat{\hat{\alpha}}_3(2) &= [\hat{\alpha}_2(1)a_{12} + \hat{\alpha}_2(2)a_{22} + \hat{\alpha}_2(3)a_{32}]b_2(2) \\
&= [2/7 \times 1/3 + 0 \times 1/3 + 5/7 \times 1/3] \times 2/6 = 1/9 \\
\hat{\hat{\alpha}}_3(3) &= [\hat{\alpha}_2(1)a_{13} + \hat{\alpha}_2(2)a_{23} + \hat{\alpha}_2(3)a_{33}]b_3(2) \\
&= [2/7 \times 1/3 + 0 \times 1/3 + 5/7 \times 1/3] \times 2/6 = 1/9
\end{aligned}$$

$$\begin{aligned}
c_3 &= \frac{1}{\hat{\hat{\alpha}}_3(1) + \hat{\hat{\alpha}}_3(2) + \hat{\hat{\alpha}}_3(3)} = \frac{1}{2/15 + 1/9 + 1/9} = \frac{1}{16/45}
\end{aligned}$$

$$\begin{aligned}
\hat{\alpha}_3(1) &= \frac{2/15}{16/45} = 6/16 \\
\hat{\alpha}_3(2) &= \frac{1/9}{16/45} = 5/16 \\
\hat{\alpha}_3(3) &= \frac{1/9}{16/45} = 5/16.
\end{aligned}$$

If we compare the above calculations to the calculations done for the normal Forward-Backward algorithm without scaling in section C.2.2 on page 115, we can see that the scaled probabilities now are normalised such that $\sum_{i=0}^{N} \hat{\alpha}_t(i) = 1$, enabling us to represent the forward probabilities within reasonable bounds for even large ts.

If we look at the scaled version of calculating the forward probabilities $\hat{\alpha}_t(i)$, we can express it in terms of the non-scaled probabilities $\alpha_t(i)$ as given in equation (C.11), (C.12) on page 113 and the scaling coefficients.

$$\begin{aligned}
\hat{\alpha}_1(j) &= c_1 \alpha_1(j) \\
\hat{\alpha}_2(j) &= c_2 \hat{\hat{\alpha}}_2(j) \\
&= \frac{\sum_{i=1}^{N} \hat{\alpha}_1(i) a_{ij} b_j(O_2)}{\sum_{j=1}^{N} \hat{\hat{\alpha}}_2(j)} \\
&= \frac{\sum_{i=1}^{N} \hat{\alpha}_1(i) a_{ij} b_j(O_2)}{\sum_{j=1}^{N} \sum_{i=1}^{N} \hat{\alpha}_1(i) a_{ij} b_j(O_2)} \\
&= \frac{\sum_{i=1}^{N} c_1 \alpha_1(i) a_{ij} b_j(O_2)}{\sum_{j=1}^{N} \sum_{i=1}^{N} c_1 \alpha_1(i) a_{ij} b_j(O_2)} \qquad \text{(C.56)} \\
\hat{\alpha}_{\dots}(j) &= \cdots
\end{aligned}$$

Furthermore it can be proven by induction that:

$$\hat{\alpha}_t(j) = \left(\prod_{\tau=1}^{t} c_\tau \right) \alpha_t(j) = C_t \alpha_t(j) \qquad \text{(C.57)}$$

To see why this is so, we take a look at the calculations for $\hat{\alpha}_2(j)$ in a HMM with three states, and observe how the scaling coefficients can be extracted from the summations:

$$\begin{aligned}
\hat{\alpha}_2(j) &= c_2\hat{\hat{\alpha}}_2(j) \\
&= c_2\left[\sum_{i=1}^{N}\hat{\alpha}_1(i)a_{ij}\right]b_j(O_2) \\
&= c_2[c_1\alpha_1(1)a_{1j} + c_1\alpha_1(2)a_{2j} + c_1\alpha_1(3)a_{3j}]b_j(O_2) \\
&= c_2c_1[\alpha_1(1)a_{1j} + \alpha_1(2)a_{2j} + \alpha_1(3)a_{3j}]b_j(O_2) \\
&= c_2c_1\alpha_2(j)
\end{aligned}$$

With the definition of $\hat{\alpha}_t(j)$ given in equation(C.57) and the observation made in equation (C.56) we can now see that:

$$\hat{\alpha}_t(j) = \frac{\sum_{i=1}^{N}\left(\prod_{\tau=1}^{t-1}c_\tau\right)\alpha_{t-1}(i)a_{ij}b_j(O_t)}{\sum_{j=1}^{N}\sum_{i=1}^{N}\left(\prod_{\tau=1}^{t-1}c_\tau\right)\alpha_{t-1}(i)a_{ij}b_j(O_t)}$$

$$\hat{\alpha}_t(j) = \frac{\left(\prod_{\tau=1}^{t-1}c_\tau\right)\sum_{i=1}^{N}\alpha_{t-1}(i)a_{ij}b_j(O_t)}{\left(\prod_{\tau=1}^{t-1}c_\tau\right)\sum_{j=1}^{N}\sum_{i=1}^{N}\alpha_{t-1}(i)a_{ij}b_j(O_t)}$$

$$\hat{\alpha}_t(j) = \frac{\alpha_t(j)}{\sum_{j=1}^{N}\alpha_t(j)} \tag{C.58}$$

So, with the scaling applied to the Forward-Backward algorithm, we are now guaranteed that all forward values are normalised, within $[0,1]$, and that $\sum_{i=0}^{N}\hat{\alpha}_t(i) = 1$.

Now that we have scaled the forward values, we are no longer able to compute the value $P(O|\lambda)$ by equation (C.13) given on page 114. But if we look at the property $\sum_{i=0}^{N}\hat{\alpha}_t(i) = 1$ for the scaled forward values for $t = T$, we can instead calculate it by:

$$
\begin{aligned}
1 &= \sum_{i=0}^{N} \hat{\alpha}_T(i) \Leftrightarrow && \text{(C.59)}\\
1 &= \prod_{\tau}^{T} c_\tau \sum_{i=1}^{N} \alpha_T(i) \Leftrightarrow && \text{(C.60)}\\
1 &= \prod_{\tau}^{T} c_\tau P(O|\lambda) \Leftrightarrow && \text{(C.61)}\\
P(O|\lambda) &= \frac{1}{\prod_{\tau}^{T} c_\tau} \Leftrightarrow && \text{(C.62)}\\
log[P(O|\lambda)] &= -\sum_{\tau}^{T} c_\tau. && \text{(C.63)}
\end{aligned}
$$

Here we compute the log likelihood $log[P(O|\lambda)]$ instead of $P(O|\lambda)$, because $P(O|\lambda)$ will properly be very small, and not within the boundary of double precision values in the computer.

Scaling the Backward Probabilities

We now turn to the scaling of the backward probabilities. This is done in quite the same way as with the forward probabilities, but instead of calculating new scale coefficients for all the backward probabilities, we just re-use the same scaling coefficients as found by the scaled version of the Forward-Backward algorithm, keeping the calculations to a minimum.

step 1: Initialisation.

$$\hat{\beta}_T(i) = c_T, \qquad 1 \le i \le N \tag{C.64}$$

step 2: Induction.

$$\hat{\beta}_t(i) = c_t \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \hat{\beta}_{t+1}(j), \qquad 1 \le i \le N$$
$$t = T-1, T-2, \cdots, 1 \tag{C.65}$$

To illustrate how the algorithm works, we take the example HMM presented in section C.1.3 on page 110 and calculate the $\hat{\beta}$ values for the observation sequence $O = RBG$ using the scaling coefficients found by the scaled Forward-Backward algorithm.

$$
\begin{aligned}
\hat{\beta}_3(1) &= c_3 = \frac{1}{16/45} = 45/16\\
\hat{\beta}_3(2) &= c_3 = 45/16\\
\hat{\beta}_3(3) &= c_3 = 45/16
\end{aligned}
$$

$$\begin{aligned}
\hat{\beta}_2(1) &= c_2 \times [a_{11}b_1(2)\hat{\beta}_3(1) + a_{12}b_2(2)\hat{\beta}_3(2) + a_{13}b_3(2)\hat{\beta}_3(3)] \\
&= \frac{1/3 \times 2/5 \times 45/16 + 1/3 \times 2/6 \times 45/16 + 1/3 \times 2/6 \times 45/16}{7/30} \\
&= \frac{3/8 + 5/16 + 5/16}{7/30} = 30/7 \\
\hat{\beta}_2(2) &= c_2 \times [a_{21}b_1(2)\hat{\beta}_3(1) + a_{22}b_2(2)\hat{\beta}_3(2) + a_{23}b_3(2)\hat{\beta}_3(3)] \\
&= 30/7 \\
\hat{\beta}_2(3) &= c_2 \times [a_{31}b_1(2)\hat{\beta}_3(1) + a_{32}b_2(2)\hat{\beta}_3(2) + a_{33}b_3(2)\hat{\beta}_3(3)] \\
&= 30/7
\end{aligned}$$

$$\begin{aligned}
\hat{\beta}_1(1) &= c_1[a_{11}b_1(3)\hat{\beta}_2(1) + a_{12}b_2(3)\hat{\beta}_2(2) + a_{13}b_3(3)\hat{\beta}_2(3)] \\
&= \frac{1/3 \times 1/5 \times 30/7 + 1/3 \times 0/6 \times 30/7 + 1/3 \times 3/6 \times 30/7}{37/90} \\
&= \frac{2/7 + 0 + 5/7}{37/90} = 90/37 \\
\hat{\beta}_1(2) &= c_1[a_{21}b_1(3)\hat{\beta}_2(1) + a_{22}b_2(3)\hat{\beta}_2(2) + a_{23}b_3(3)\hat{\beta}_2(3)] \\
&= 90/37 \\
\hat{\beta}_1(3) &= c_1[a_{31}b_1(3)\hat{\beta}_2(1) + a_{32}b_2(3)\hat{\beta}_2(2) + a_{33}b_3(3)\hat{\beta}_2(3)] \\
&= 90/37
\end{aligned}$$

When using the scaling coefficients found from forward values to also scale the backward values with, we are forced to always calculate the scaled forward values before we can calculate the scaled backward values. This should be kept in mind when implementing the scaled versions of the Forward-Backward and Backward-Forward algorithms.

As with the scaled forward variables, we can express the scaled backward variables in terms of the non-scaled backwards variables and the product of the scaling coefficients.

$$\hat{\beta}_t(i) = \left(\prod_{\tau=t}^{T} c_\tau\right)\beta_t(i) = D_t\beta_t(i) \tag{C.66}$$

Re-Estimating Using Scaled αs and βs

We now turn to the re-estimating of the Hidden Markov Model parameters using the scaled forward and backward probabilities.

When re-estimating the initial state probabilities π, we need to calculate $\gamma_1(i)$, but how do we find the $\gamma_t(i)$ values from the scaled αs and βs. As seen in section C.2.4 from equation (C.22) on page 120 we defined $\gamma_t(i)$ as:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} \tag{C.67}$$

Re-writing equation (C.57) and (C.66) we express $\alpha_t(i)$ and $\beta_t(i)$ as:

$$\alpha_t(i) = \frac{\hat{\alpha}_t(i)}{C_t}, \tag{C.68}$$

$$\beta_t(i) = \frac{\hat{\beta}_t(i)}{D_t}. \tag{C.69}$$

Furthermore, the terms C_t and D_t can be expressed as:

$$C_t D_t \quad = \quad \prod_{s=1}^{t} c_s \prod_{s=t}^{T} c_s = c_t \prod_{s=1}^{T} c_s = c_t C_T, \tag{C.70}$$

enabling us to express $\gamma_t(i)$ in terms of the scaled forward and backward probabilities, and the scale coefficient as:

$$\begin{aligned} \gamma_t(i) &= \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{C_t D_t P(O|\lambda)} && \text{(C.71)} \\ &= \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{C_t D_t \frac{1}{\prod_{\tau=1}^{T} c_\tau}} && \text{(C.72)} \\ &= \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{c_t C_T \frac{1}{C_T}} && \text{(C.73)} \\ &= \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{c_t} && \text{(C.74)} \end{aligned}$$

The re-estimation of the initial state probability distribution π is therefore given by:

$$\begin{aligned} \overline{\pi}_i &= \gamma_1(i) && \text{(C.75)} \\ &= \frac{\hat{\alpha}_1(i)\hat{\beta}_1(i)}{c_1} && \text{(C.76)} \end{aligned}$$

When re-estimating the state transition probabilities a_{ij} using the scaled forward and backward values, we simply use the following equation:

$$\overline{a}_{ij} \quad = \quad \frac{\sum_{t=1}^{T-1} \hat{\alpha}_t(i) a_{ij} b_j(O_{t+1}) \hat{\beta}_{t+1}(j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \hat{\alpha}_t(i) a_{ij} b_j(O_{t+1}) \hat{\beta}_{t+1}(j)}. \tag{C.77}$$

To see why this gives the correct re-estimation, we re-write the equation in terms of αs and βs as defined by equation (C.57) and (C.66):

$$\overline{a}_{ij} \quad = \quad \frac{\sum_{t=1}^{T-1} C_t \alpha_t(i) a_{ij} b_j(O_{t+1}) D_{t+1} \beta_{t+1}(j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} C_t \alpha_t(i) a_{ij} b_j(O_{t+1}) D_{t+1} \beta_{t+1}(j)}. \tag{C.78}$$

The product of the two terms C_t and D_{t+1} can by expressed by:

$$C_t D_{t+1} = \prod_{s=1}^{t} c_s \prod_{s=t+1}^{T} c_s = \prod_{s=1}^{T} c_s = C_T, \tag{C.79}$$

which is independent of t and can therefore be cancel out in both the numerator and the denominator:

$$\begin{aligned} \overline{a}_{ij} &= \frac{\sum_{t=1}^{T-1} C_t D_{t+1} \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} C_t D_{t+1} \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)} && \text{(C.80)} \\ &= \frac{\sum_{t=1}^{T-1} C_T \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} C_T \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)} && \text{(C.81)} \\ &= \frac{\sum_{t=1}^{T-1} \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}, && \text{(C.82)} \end{aligned}$$

resulting in the original re-estimation of a_{ij} as presented in equation (C.48).

The re-estimation of the observation probabilities $b_j(k)$ is quite simple, because it is expressed in terms of $\gamma_t(i)$ values and we have already shown how to compute the $\gamma_t(i)$ values using the scaled forward and backward values.

C.3.2 Finding Best State Sequence

As with the Forward-Backward and Backward-Forward algorithm, we also run into problems of representing the small probabilities within the boundary of the double precision values in the computer when finding the maximum likelihood state sequence using the Viterbi algorithm. To cope with this problems, we simply use log additions instead of multiplications as before. The Viterbi algorithm will then be defined as:

step 1: Initialisation

$$\begin{aligned} \delta_1(i) &= \log(\pi_i) + \log[b_i(O_1)], \qquad 1 \leq i \leq N && \text{(C.83)} \\ \psi_1(i) &= 0 && \text{(C.84)} \end{aligned}$$

step 2: Induction

$$\delta_t(j) = \max_{1 \leq i \leq N} [\delta_{t-1}(i) + \log(a_{ij})] + \log[b_j(O_t)], \qquad 1 \leq j \leq N, \quad 2 \leq t \leq T \tag{C.85}$$

$$\psi_t(j) = \operatorname*{argmax}_{1 \leq i \leq N} [\delta_{t-1}(i) + \log(a_{ij})], \qquad 1 \leq j \leq N, \quad 2 \leq t \leq T \tag{C.86}$$

The log probability of finding the best state sequence is then given as:

$$\log P^* = \max_{1 \leq i \leq N} [\delta_T(i)]. \tag{C.87}$$

And the extraction of the best sequence is done as before with equation (C.29) and (C.30) on page 121.

C.3.3 Multiple Observation Sequences

When training the HMM with multiple observation sequences there are two approaches: concatenate all observation sequence into one single and train as usual, or train every sequence and average parameters.

We define the set of observation sequences as:

$$O = \{O^{(1)}, O^{(2)}, \cdots, O^{(L)}\}, \tag{C.88}$$

where each observation sequence $O^{(l)}$ will be denoted $O^{(l)} = O_1^{(l)} O_2^{(l)} \cdots O_{T^{(l)}}^{(l)}$.

When using the first approach, the observation sequence to train the HMM will be:

$$O = O_1^{(1)} O_2^{(1)} \cdots O_{T^{(1)}}^{(1)} O_1^{(2)} O_2^{(2)} \cdots O_{T^{(2)}}^{(2)} \cdots O_1^{(L)} O_2^{(L)} \cdots O_{T^{(L)}}^{(L)} \tag{C.89}$$

When using the second approach, every sequence $O^{(l)}$ will be used to re-estimate the l'th parameters, and the final parameters for the HMM will be the average of these.

$$\bar{a}_{ij} = \frac{\sum_{l=1}^{L} \bar{a}_{ij}^{(l)}}{\sum_{j=1}^{N} \sum_{l=1}^{L} \bar{a}_{ij}^{(l)}} \tag{C.90}$$

$$\bar{b}_j(k) = \frac{\sum_{l=1}^{L} \bar{b}_j(k)^{(l)}}{\sum_{j=1}^{N} \sum_{l=1}^{L} \bar{b}_j(k)^{(l)}} \tag{C.91}$$

$$\overline{\pi}_i = \frac{\sum_{l=1}^{L} \overline{\pi}_i^{(l)}}{\sum_{j=1}^{N} \sum_{l=1}^{L} \overline{\pi}_i^{(l)}} \tag{C.92}$$

C.3.4 Initialising the Parameters of the HMM

Before training the HMM for one or more observation sequences we need to initialise the parameters of the HMM: A, B and π. There is no precise and correct solution for this, but experience has shown that a random or uniform initial estimate for π and A give good results. Whereas to assure a prober and correct convergence of the re-estimation formulae, a good initial estimate of the B parameter is often needed. A good initial estimate of the B parameter can for example be made by segmenting the observation sequences into states and averaging the observations in each state concerned. We could for instance used the

$$O_{i+qN}, \qquad 0 \leq q \leq \frac{T-i}{N} \tag{C.93}$$

observation symbols to estimate the observation probability distribution for state S_i in a HMM with N states and an observation sequence of length T. To illustrate this, lets take the observation sequence $AABBCDBBAAAADC$ with length 14 and a HMM with 3 states. The observation symbols used to estimate the observation probability distribution for the first state would then be O_1, O_4, O_7, O_{10}, O_{13}:

$$\begin{array}{cccccccccccccc} A & A & B & B & C & D & B & B & A & A & A & A & D & C \\ \uparrow & & & \uparrow & & & \uparrow & & & \uparrow & & & \uparrow & \end{array}$$

The observation probability distribution for the first state will then be estimated as:

$$O_1 O_4 O_7 O_{10} O_{13} = ABBAD \rightarrow \begin{cases} P(A) = 2/5 \\ P(B) = 2/5 \\ P(C) = 0 \\ P(D) = 1/5 \end{cases}$$

Another quite similar way of estimating B, is to simply divide the observation sequence into N segments and use the i'th segment to estimate the observation probability distribution for state S_i:

$$\underbrace{AABBC}_{\text{seg. 1}}\underbrace{DBBAA}_{\text{seg. 2}}\underbrace{AADC}_{\text{seg. 3}}$$

The observation probability distribution for the first state would now be estimated as:

$$O_1 O_2 O_3 O_4 O_5 = AABBC \rightarrow \begin{cases} P(A) = 2/5 \\ P(B) = 2/5 \\ P(C) = 1/5 \\ P(D) = 0 \end{cases}$$

No matter what kind of initial estimate is chosen for A, B and π, we should always assure that the stochastic constant of the HMM's parameters is satisfied:

$$\sum_{j=1}^{N} a_{ij} = 1, \qquad 1 \leq i \leq N$$

$$\sum_{k=1}^{M} b_j(k) = 1, \qquad 1 \leq j \leq N$$

$$\sum_{i=1}^{N} \pi_i = 1$$

Appendix D

Implementation, Test and Usage

All the software implemented to construct Hidden Markov Models and experimenting with them is included in a package named CIS. The CIS is short for Computer Immune System and named in this way because the classes in the package are thought of as being a preliminary version of a computer immune system. The source code for the CIS package is written in Java and can be found at:

`http://www.student.dtu.dk/~c971851/thesis/source/CIS`

and the documentation for the CIS package generated by the javadoc program can found at:

`http://www.student.dtu.dk/~c971851/thesis/source/docs/`

The software is implemented in java version 1.1 and will require a java runtime environment to work. The package CIS consist of the following five classes:

HMM: Constructs a HMM and implements the algorithms use in connexion with a HMM.
Util: Supply small utility methods for the HMM class.
Timer: Supply methods for measuring execution time.
APIParser: Supply methods for parsing and compressing APISPY output files into binary byte sequences usable for HMM training.
ByteBuffer: Supply a byte buffer and methods for writing bytes, hexadecimal values, integers, and strings to it.

We will in the following five sections describe the classes, test them and explain how they are used.

D.1 The HMM Class

The HMM class provides a framework for making Hidden Markov Models and training them using one or multiple byte sequences. Furthermore the class also provides methods for computing the likelihood of observing a byte sequence and for finding the best state sequence.

When initialising the parameters of the HMM several approaches could be used. Experience from prior work has shown that the training of the HMMs will converge faster if the initial symbol observation probability distribution, denoted B, is estimated from the symbol occurrences in the training sequence. The HMM class therefore provides a method for initialising B in a numbers of ways.

An always recurrent problem when using HMMs for large observation sequences is representing the small values of the probabilities within the boundary of double precision values in the computer. To cope with this problem the HMM class provides methods for using either scaled probabilities or non-scaled probabilities. In this way the HMM class is divided into two parts: methods for using scaled probabilities and methods for using non-scaled probabilities.

We now list the most important methods and describe their functionality:

- `HMM(int N, int M)`
  Constructor, creates a HMM with `N` states and `M` observations symbols in each state.
- `setEstB(int estB)`
  Set the type of initial estimate of B to `estB`. The type `estB` could be one of the following values: `RANDOM_B_EST`, `SEG1_B_EST`, `SEG2_B_EST`.

Methods in the HMM class using non-scaled probabilities:

- `forward(int[] O)`
  Calculates all the forward probabilities for the observation sequence `O` using the Forward-Backward algorithm as given in equation (C.11) and (C.12) on page 113.
- `backward(int[] O)`
  Calculates all the backward probabilities for the observation sequence `O` using the Backward-Forward algorithm as given in equation (C.17) and (C.18) on page 117.
- `double likelihood(byte[] O)`
  Return the likelihood of observing the sequence `O` in the HMM using non-scaled probabilities as given in equation (C.13) on page 114.
- `viterbi(int[] O)`
  Calculates the viterbi probabilities and finds the best state sequence for the observation sequence `O` using the Viterbi algorithm as given in equation (C.24) to (C.27) on page 120.
- `train(int maxIterations, double threshold, byte[] trainingSet)`
  Trains the HMM without scaling and the single observation sequence given in `trainingSet`. The method will keep on training until `threshold` or

`maxIterations` have been reached. The train method implements the Baum-Welch algorithm, which will iteratively adjust the HMM model parameters (A, B, π) as given in equation (C.48), (C.41) and (C.39) to maximise the probability of the given observation sequence in `trainingSet`.

Methods in the HMM class using scaled probabilities:

- `forwardScaling(int[] O)`
  Calculates all the forward probabilities for the observation sequence `O` using the scaled version of the Forward-Backward algorithm as given in equation (C.52) to (C.55) on page 129.
- `backwardScaling(int[] O)`
  Calculates all the backward probabilities for the observation sequence `O` using the scaled version of the Backward-Forward algorithm as given in equation (C.64) and (C.65) on page 133.
- `double logLikelihood(byte[] O)`
  Return the log likelihood of observing the sequence `O` in the HMM as given in equation (C.63) on page 133.
- `logViterbi(int[] O)`
  Calculates the log viterbi probabilities and finds the best state sequence for the observation sequence `O` using the Viterbi algorithm with log values of the HMM parameters as given in equation (C.83) to (C.86) on page 136.
- `train(int maxIterations, double threshold,`
  `byte[][] trainingSet)`
  Train the HMM with scaling and multiple observations from byte sequences in `trainingSet`. The method will keep on training until `threshold` or `maxIterations` have been reached. The train method implements the Baum-Welch algorithm using scaled forward and backward values, which will iteratively adjust the HMM model parameters (A, B, π) as given in equation (C.77), (C.41) using (C.74), and (C.76).

Methods for writing and reading HMMs to and from files.

- `writeObject(java.io.ObjectOutputStream out)`
  Writes the state of the HMM to an output stream `out` so we can restore it later by using the `readObject` method.
- `readObject(java.io.ObjectInputStream in)`
  Reads the state of a HMM from an input stream `in` so we can restore it. The method reads information from the stream and assign the fields saved in the stream to the correspondingly named fields in the current HMM object.

D.1.1 Testing the HMM Class

To test the methods in the HMM class, we use the “Urn and Ball” example given in section C.1.3 on page 110. The results found in appendix C will be compared with the results found by running the implemented algorithms on the same example, in this way we only test the functionality of the methods of the HMM class and not every possible branch or case in the methods.

To test all the algorithms we first initialise a HMM having the same parameters as the "Urn and Ball" example in section C.1.3 on page 110:

```
// Number of states.
int N=3;
// Number of different symbols in each state.
int M = 3;
// Initialise the HMM
HMM hmm = new HMM(N,M);
// Initialise the transition probability distribution.
for(int i=0;i<N;i++)
    for(int j=0;j<N;j++)
        hmm.A[i][j] = 1.0/N;
// Initialise the symbol probability distribution.
hmm.B[0][0]=2.0/5.0;
hmm.B[1][0]=4.0/6.0;
hmm.B[2][0]=1.0/6.0;
hmm.B[0][1]=2.0/5.0;
hmm.B[1][1]=2.0/6.0;
hmm.B[2][1]=2.0/6.0;
hmm.B[0][2]=1.0/5.0;
hmm.B[1][2]=0.0/6.0;
hmm.B[2][2]=3.0/6.0;
// Initialise the initial state probability distribution.
for(int i=0;i<N;i++)
    hmm.pi[i]=1.0/N;
```

The initialised HMM is then used to test the algorithms on the sequence `byte seq[] = {0,2,1}` corresponding to the sequence $O = RBG$[1], which is used in all the examples in appendix C.

Testing the Forward-Backward algorithm

The α values returned from running the `forward` algorithm on the example HMM:

0.13333333333333333	0.2222222222222222	0.05555555555555555
0.027407407407407405	0.0	0.0685185185185185
0.01279012345679012	0.010658436213991766	0.010658436213991766

The α values from the appendix C; see calculations beginning on page 115:

[1] The sequence 0,2,1 corresponds to $O = RBG$ because R have index 0, B have index 2 and G have index 1 in the set of symbols $V = \{R, G, B\}$.

$$\begin{array}{ccc} 2/15 & 4/18 & 1/18 \\ 37/1350 & 0 & 37/540 \\ 259/20250 & 152/14261 & 152/14261 \end{array} = \begin{array}{ccc} 0.1333 & 0.2222 & 0.0556 \\ 0.0274 & 0.0 & 0.0685 \\ 0.0128 & 0.0107 & 0.0107 \end{array}$$

As we can see, the two results correspond and the `forward` algorithm works as expected.

Testing the Backward-Forward algorithm

The β values returned from running the `backward` algorithm on the example HMM:

$$\begin{array}{ccc} 0.08296296296296295 & 0.08296296296296295 & 0.08296296296296295 \\ 0.3555555555555555 & 0.3555555555555555 & 0.3555555555555555 \\ 1.0 & 1.0 & 1.0 \end{array}$$

The β values from appendix C; see calculations beginning one page 117:

$$\begin{array}{ccc} 56/675 & 56/675 & 56/675 \\ 16/45 & 16/45 & 16/45 \\ 1 & 1 & 1 \end{array} = \begin{array}{ccc} 0.0830 & 0.0830 & 0.0830 \\ 0.3556 & 0.3556 & 0.3556 \\ 1.0 & 1.0 & 1.0 \end{array}$$

As we can see the two results correspond and the `backward` algorithm works as expected.

Testing the Likelihood Method

The result of running the `likelihood` method on the example HMM:

$$0.03410699588477365$$

The result found in appendix C; see calculations for $P(O|\lambda)$ on page 115:

$$313/9177 = 0.0341$$

As we can see the two values correspond and the `likelihood` method works as expected.

Testing the Viterbi algorithm

The δ and ψ values computed by the `viterbi` algorithm on the example HMM:

$$\begin{array}{ccc} 0.13333333333333333 & 0.2222222222222222 & 0.05555555555555555 \\ 0.014814814814814815 & 0.0 & 0.037037037037037035 \\ 0.0049382716049382715 & 0.004115226337448559 & 0.004115226337448559 \end{array}$$

$$\begin{array}{ccc} 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 2.0 \end{array}$$

The δ and ψ values found in appendix C; see calculations beginning on page 121:

$$\begin{array}{ccc} 2/15 & 2/9 & 1/18 \\ 2/135 & 0 & 1/27 \\ 2/405 & 1/243 & 1/243 \end{array} = \begin{array}{ccc} 0.1333 & 0.2222 & 0.0556 \\ 0.0148 & 0.0 & 0.0370 \\ 0.0049 & 0.0041 & 0.0041 \end{array}$$

$$\begin{array}{ccc} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{array}$$

As we can see the δ values correspond to each other, but the ψ values does not, this is because the indexes in the implemented Viterbi algorithm start at 0 whereas the indexes in appendix C start at 1. We should therefore subtract 1 from all the ψ values found in appendix C, resulting in identical values computed by the Viterbi algorithm and the results found in appendix C. The `viterbi` algorithm works as expected.

Testing the Baum-Welch Algorithm

The Baum-Welch algorithm uses the alpha and beta values computed by the already tested `forward` and `backward` algorithms. Running the Baum-Welch algorithm, which is implemented by the `train` method with the sequence `seq={0,2,1}`, `maxIterations=5`, and `threshold=0.001` results in the following output:

```
Iteration=0, Likelihood=0.03410699588477365, Log
             likelihood=-1.4671565311971544 Iteration=1,
             Likelihood=0.07303222775356699, Log
             likelihood=-1.1364854515655216 Iteration=2,
             Likelihood=0.17462671523930984, Log
             likelihood=-0.7578893150778534 Iteration=3,
             Likelihood=0.22502062385544125, Log
             likelihood=-0.6477776755946804 Iteration=4,
             Likelihood=0.24249959563061474, Log
             likelihood=-0.6152889812495076
```

As we can see the likelihood is improved at every iteration as we expected. Furthermore, we can see that probability of observing the sequence `seq={0,2,1}` was only 0.034 to begin with, it has now been improved to 0.242 after only 5 iterations. We can clearly see that the implemented version of the Baum-Welch algorithm works as expected.

Testing the Scaled Forward-Backward Algorithm

When running the `forwardScaling` method on the example HMM we get the following $\hat{\alpha}$ values:

```
0.32432432432432434   0.5405405405405406   0.13513513513513514
 0.2857142857142857          0.0            0.7142857142857142
       0.375                0.3125               0.3125
```

The $\hat{\alpha}$ values calculated in appendix C; see calculations beginning on page 130:

$$\begin{array}{ccc} 12/37 & 20/37 & 5/37 \\ 2/7 & 0 & 5/7 \\ 6/16 & 5/16 & 5/16 \end{array} = \begin{array}{ccc} 0.3243 & 0.5405 & 0.1351 \\ 0.2857 & 0.0 & 0.7143 \\ 0.3750 & 0.3125 & 0.3125 \end{array}$$

As we can see the two results correspond and the `forwardScaling` method works as expected.

Testing the Scaled Backward-Forward Algorithm

The $\hat{\beta}$ values computed by the `backwardScaling` method:

```
 2.4324324324324433   2.4324324324324433   2.4324324324324433
4.2857142857142865   4.2857142857142865   4.2857142857142865
 2.812500000000001    2.812500000000001    2.812500000000001
```

The $\hat{\beta}$ values calculated in appendix C; see calculations beginning on page 133:

$$\begin{array}{ccc} 90/37 & 90/37 & 90/37 \\ 30/7 & 30/7 & 30/7 \\ 45/16 & 45/16 & 45/16 \end{array} = \begin{array}{ccc} 2.4324 & 2.4324 & 2.4324 \\ 4.2857 & 4.2857 & 4.2857 \\ 2.8125 & 2.8125 & 2.8125 \end{array}$$

As we can see the two results correspond and the `backwardScaling` method works as expected.

Testing the Log Likelihood Method

The `logLikelihood` method uses the scaling coefficients calculated by the `forwardScaling` method to find the log likelihood, see equation (C.63). When running the `logLikelihood` method on the example HMM we get the following result:

```
-1.4671565311971544
```

If we calculate the base 10 logarithm of the result found for $P(O|\lambda)$ in appendix C on page 115 we get:

$$\text{log10(313/9177)=-1.4672}$$

As we can see the two results correspond and the `logLikelihood` method works as expected.

Testing the Log Viterbi Algorithm

To test that the `logViterbi` method works, we simply run it on the example HMM and observe if we get the same final result as with the `viterbi` method tested above. The result from the test run:

```
0.0  0.0  0.0
1.0  1.0  1.0
2.0  2.0  2.0
```

As we can see, the result of running the `logViterbi` method is the same as the final result returned by the `viterbi` method, see section D.1.1 on page 143, the `logViterbi` works as expected.

Testing the Scaled Baum-Welch Algorithm

As explained in section C.3.1 on page 134 re-estimating the parameters of the HMM using scaled alphas and betas result in the same parameters as the re-estimation with non-scaled alphas and betas. So to verify that the scaled Baum-Welch algorithm works correctly, we simply assure that the re-estimated parameters and the log likelihoods returned by the already tested Baum-Welch algorithm are the same as the parameters and log likelihoods returned by the scaled version of the Baum-Welch algorithm.

The log likelihoods computed during normal Baum-Welch re-estimation:

```
Iteration=0, Likelihood=0.03410699588477365,
             Log likelihood=-1.4671565311971544
Iteration=1, Likelihood=0.07303222775356699,
             Log likelihood=-1.1364854515655216
Iteration=2, Likelihood=0.17462671523930984,
             Log likelihood=-0.7578893150778534
Iteration=3, Likelihood=0.22502062385544125,
             Log likelihood=-0.6477776755946804
Iteration=4, Likelihood=0.24249959563061474,
             Log likelihood=-0.6152889812495076
```

The log likelihoods computed during the scaled Baum-Welch re-estimation:

```
Iteration=0, Log likelihood=-1.4671565311971544
Iteration=1, Log likelihood=-1.1364854515655214
Iteration=2, Log likelihood=-0.7578893150778534
Iteration=3, Log likelihood=-0.6477776755946802
Iteration=4, Log likelihood=-0.6152889812495075
```

As we can see the log likelihoods returned by the scaled and the non-scaled Baum-Welch algorithm are the same.

The re-estimated parameter A after normal Baum-Welch re-estimation:

$$\begin{array}{ccc}
0.011075356570339118 & 0.0 & 0.988924643429661 \\
0.011075356570339121 & 0.0 & 0.9889246434296608 \\
0.01107535657033912 & 0.0 & 0.9889246434296608
\end{array}$$

The re-estimated parameter A after scaled Baum-Welch re-estimation:

$$\begin{array}{ccc}
0.0110753565703391 & 0.0 & 0.988924643429661 \\
0.0110753565703391 & 0.0 & 0.9889246434296609 \\
0.0110753565703391 & 0.0 & 0.9889246434296608
\end{array}$$

As we can see the re-estimated parameter A is the same in both the scaled and non-scaled version of the Baum-Welch algorithm.

The re-estimated parameter B returned after normal Baum-Welch re-estimation:

$$\begin{array}{ccc}
0.09928602571839916 & 0.6718151843531609 & 0.22889878992843998 \\
1.0 & 0.0 & 0.0 \\
7.771879152217893\text{E-}14 & 0.4945229816460044 & 0.5054770183539179
\end{array}$$

The re-estimated parameter B computed by the scaled version of the Baum-Welch re-estimation:

$$\begin{array}{ccc}
0.09928602571839924 & 0.6718151843531611 & 0.22889878992843973 \\
1.0 & 0.0 & 0.0 \\
7.771879152217905\text{E-}14 & 0.4945229816460044 & 0.5054770183539178
\end{array}$$

As we can see the re-estimated parameter B returned by the scaled and the non-scaled version of the algorithm is the same.

The re-estimated parameter π computed after normal Baum-Welch re-estimation:

$$\begin{array}{ccc}
0.00480399279361374 & 0.9951960072062341 & 1.520504897416356\text{E-}13
\end{array}$$

The π parameter re-estimated by the scaled version of the Baum-Welch algorithm:

$$\begin{array}{ccc}
0.00480399279361374 & 0.9951960072062341 & 1.5205048974163585\text{E-}13
\end{array}$$

As we can see the scaled version of the Baum-Welch algorithm computes the same log likelihoods and parameters as the non-scaled version, we can therefore conclude that the scaled Baum-Welch algorithm works as expected.

D.1.2 Using the HMM Class

A HMM object is instantiated by using the constructor `HMM(N,M)`, which initialises a HMM with `N` states and `M` symbols in every state. To select which type of initial symbol observation probability distribution to use, the method `setEstB(type)` should be called, the value `type` could be any of `RANDOM_B_EST`, `SEG1_B_EST` or `SEG2_B_EST`. The random type indicates that symbol observation probability distribution should be randomly initialised, this is the default, whereas the two other types indicates two different ways of segmenting the training sequence(s) to find a good initial estimate. When training the HMM using scaled probabilities the `train(byte[][])` method should be used, this method also support multiple training sequences. If training the HMM using non-scaled probabilities the `train(byte[])` method should be used, this method does *not* support multiple training sequences. The log likelihood of observing a certain symbol sequence in the HMM can now be computed using the method `logLikelihood(byte[])`, if small observation sequences are used the normal likelihood can also be computed using `likelihood(byte[])`. When training is done the HMM can be saved in a file using the `writeObject()` method, to restore it later the `readObject()` method should be used.

The following small example code illustrates how the HMM class could be used. The example code will train a HMM for some small observation sequences, find the log likelihood of observing another sequence in the trained HMM, write the HMM to the disk and open it again.

```
// Make HMM with 3 states and 256 symbols in each state.
HMM hmm = new HMM(3,256);
// Train hmm on byte sequences b until threshold=0.001 or
// maxIterations=50 is reached.
byte[][] b ={{-4,56,38},{5,120,-3}};
hmm.train(50,0.001,b);
// Find the log likelihood of observing b1 in hmm.
byte[] b1={-4,120,-3};
hmm.logLikelihood(b1);

try{
    // Write hmm to file.
    FileOutputStream ostream = new FileOutputStream("test.hmm");
    ObjectOutputStream out = new ObjectOutputStream(ostream);
    out.writeObject(hmm);
    out.flush();
    ostream.close();
    // Open hmm from file
    FileInputStream istream = new FileInputStream("test.hmm");
    ObjectInputStream in = new ObjectInputStream(istream);
    hmm = (HMM)in.readObject();
    istream.close();
}
catch(Exception e){
    e.printStackTrace();
}
```

D.2 The Util Class

The Util class provides small utility methods for all the other classes in the CIS package. Most of the methods are used by the HMM class to make its code more intuitive and clean. The methods convert byte arrays into integer arrays, compute the sum of vectors, calculate the base 10 logarithm of values and stuff like that.

We now list the most important methods in the Util class and describe their functionality shortly:

- `int argmax(double[] a)`
  Returns the index in `a` that holds the maximum value. If array is empty the method will return -1.
- `double log10(double x)`
  Returns the base 10 logarithm of a double value.
- `double[] log10(double[] vec)`
  Returns the base 10 logarithm of the values in a double array.
- `normalize(double[] a)`
  Normalises a double array such that the sum of all the elements are 1.0.
- `double sum(double[] vec)`
  Returns the sum of a double array.
- `int toInt(byte b)`
  Converts a byte to an unsigned int.
- `int[] toInt(byte[] b)`
  Converts a byte array to an unsigned int array.

To support the HMM class in saving and reading its state to and from files the following four methods are supplied:

- `writeDouble(java.io.ObjectOutputStream out, double[] a)`
  Writes a double array to an ObjectOutputStream.
- `writeDouble(java.io.ObjectOutputStream out, double[][] a)`
  Writes a two dimensional double array to an ObjectOutputStream.
- `readDouble(java.io.ObjectInputStream in, double[] a)`
  Reads a double array from an ObjectInputStream.
- `readDouble(java.io.ObjectInputStream in, double[][] a)`
  Reads a two dimensional double array from an ObjectInputStream.

D.2.1 Testing the Util Class

We will in this section test the small utility methods of the Util class. In every test we submit the arguments to the method, the result of calling the method and indicate whether the result is okay or not. Table D.1 on the next page lists the tests of the `argmax`, `log10`, `normalize`, `sum`, and `toInt` methods.

As we can see from table D.1 on the following page the `normalize` method does not quite work if some of the elements in the array are negative and the

Method	Arguments	Results	OK
argmax	`[]`	`-1`	yes
	`[0.0,1.0,3.0,5.0]`	`3`	yes
	`[-10.0,-4.0,0.0]`	`2`	yes
	`[-10.0,-4.0,0.0,5.0]`	`3`	yes
log10	`1000000`	`5.999999999999999`	yes
	`1`	`0.0`	yes
	`0.000000001`	`-8.999999999999998`	yes
	`0`	`-Infinity`	yes
	`-0.00000001`	`NaN`	yes
	`-100000`	`NaN`	yes
log10	`[]`	`[]`	yes
	`[-10,-0.01,0, 0.01,1,10]`	`[NaN,NaN, -Infinity, -1.9999999999999996, 0.0,1.0]`	yes
normalize	`[]`	`[]`	yes
	`[0.0,0.0]`	`[NaN,NaN]`	yes
	`[-2.0,2.0]`	`[-Infinity,Infinity]`	no
	`[1.0,2.0]`	`[0.3333333333333333, 0.6666666666666666]`	yes
	`[-1.0,-2.0]`	`[0.3333333333333333, 0.6666666666666666]`	yes
	`[1.0,-2.0]`	`[1.0,-2.0]`	yes
sum	`[]`	`0`	yes
	`[0.0,0.0]`	`0`	yes
	`[-3.0,1.0,5.0]`	`3.0`	yes
toInt	`0`	`0`	yes
	`127`	`127`	yes
	`-1`	`128`	yes
	`-128`	`255`	yes
toInt	`[]`	`[]`	yes
	`[0,127,-1,-128]`	`[0,127,128,255]`	yes

Table D.1: Testing results for some of the methods in the Util class.

sum of the array is 0. This is not a problem for the HMM class, which uses the method to normalise the initial HMM parameters, because the initial parameter elements will never be negative resulting in a sum of zero. They are either randomly initialises to a value between 0.0 and 1.0 or initialised from the symbol occurrences in the observation sequence(s) resulting in non-negative values.

To test that the `writeDouble` and `readDouble` methods work as expected, we write three different arrays to a temporary file and read them again. The arguments and results for the two methods are listed in table D.2

Arrays written by `writeDouble`	Arrays read by `readDouble`	OK
`[]`	`[]`	yes
`[-1.0,0.0,1.0]`	`[-1.0,0.0,1.0]`	yes

`[[-3.0,-2.0,-1.0],`
`[0.0 1.0 2.0]]`
	`[[-3.0,-2.0,-1.0],`
	`[0.0 1.0 2.0]]`
		yes

Table D.2: Testing results for the reading and writing methods of the Util class.

D.2.2 Using the Util Class

All the methods in the Util class are declared `static`, so it is not necessary to make an instance of the class before using the methods. The Methods are simply called using the following syntax:

```
Util.method(..);
```

as in the example of calculating the sum of an array:

```
double a[]={-3.0,1.0,5.0};
Util.sum(a);
```

D.3 The Timer Class

The Timer class provides an easy way for measuring the execution time of code. The time will be given in milliseconds. The class provides the following methods:

- `Timer()`
  Constructor, creates a new timer and resets it.
- `start()`
  Starts the timer.
- `long stop()`
  Stops the timer, and returns the time from start to stop.
- `reset()`
  Reset the timer.
- `toString()`
  Returns the time from start to stop as a string.

D.3.1 Testing the Timer Class

To test the Timer class we simply measure the time it takes to execute the `Thread.sleep(t)` method, the method causes the currently executing thread to sleep for `t` milliseconds. Table D.3 on the next page shows the test of the timer when executing the method `Thread.sleep(t)` for `t` equal to 10, 100, 1000, and 10000.

t	Result	OK
10	18	yes
100	107	yes
1000	1002	yes
10000	10002	yes

Table D.3: Testing the Timer class on code taking 10, 100, 1000, and 10000 milliseconds.

D.3.2 Using the Timer Class

To illustrate how the timer class could be used, we list a small example code:

```
Timer t=new Timer();
t.start();
// Execute some code.
t.stop();
System.out.println("Execution took "+t);
```

D.4 The APIParser Class

The APIParser class provides methods for parsing output files from the APISPY program. All the system calls made during execution of a program executed by the APISPY program will be logged and written to an output text file. The output will contain the system calls, the parameters given to the system calls and the return values of the system calls. The APIParser class will read the output file generated by APISPY and compress the information into a byte sequence usable for HMM training.

The log files written by the APISPY program have the following syntax:

```
APIFILE   = APICALL EOF
APICALL   = NULL | IDENT APINAME ‘‘(‘‘ PARAMS ‘‘)\n’’
                   APICALL APIRETURN ‘‘\n’’ |
            APICALL
APIRETURN = NULL | IDENT APINAME ‘‘ returns: ‘‘ RETURNVAL ‘‘\n’’
PARAMS    = NULL | PARAMTYPE ‘‘:’’ PARAMVALUE |
            PARAMTYPE ‘‘:’’ PARAMVALUE ‘‘:’’ STRING |
            ‘‘,’’ PARAMS
IDENT     = NULL | ‘‘ ‘‘ IDENT
PARAMTYPE = ‘‘DWORD’’ | ‘‘WORD’’ | ‘‘BYTE’’ | ‘‘LPSTR’’ |
            ‘‘LPWSTR’’ | ‘‘LPDATA’’ | ‘‘HANDLE’’ | ‘‘HWND’’ |
            ‘‘BOOL’’ | ‘‘LPCODE’’ | ‘‘<unknown>’’
APINAME   = ‘‘AddAtomA’’ | ‘‘AddAtomW’’ | ‘‘AllocConsole’’ |
            ........
            ‘‘wsprintfW’’ | ‘‘wvsprintfA’’ | ‘‘wvsprintfW’’
```

The NULL token represents an empty string, whereas the terms RETURNVAL and PARAMVALUE are hexadecimal values. Letters enclosed with '''' are strings found in the output file. The term APINAME represents all the system calls available from the dynamic link libraries: KERNEL32.dll, ADVAPI32.dll, COMDLG32.dll, GDI32.dll, and USER32.dll. We have not included all the system calls in the specification above because there are quite many.

Executing the idle32 program from the windows 98 distribution with the APISPY program results in the following partial output:

```
...
RegisterClassA(LPDATA:0063FD4C)
RegisterClassA returns: C251
CreateWindowExA(DWORD:00000000, LPSTR:00404034:"MACR_Slavi",...)
  DefWindowProcA(HWND:00000930,DWORD:00000024,...)
  DefWindowProcA returns: 0
  ...
  DefWindowProcA(HWND:00000930,DWORD:00000001,...)
  DefWindowProcA returns: 0
CreateWindowExA returns: 930
PeekMessageA(LPDATA:0063FD8C,HWND:00000000,...)
PeekMessageA returns: 0
...
```

Every system call is mapped to an integer. System calls made to KERNEL32.dll will be mapped to numbers starting at 1000, system calls to ADVAPI32.dll start at 2000, COMDLG32.dll start at 3000, GDI32.dll at 4000, and USER32.dll at 5000. Every parameter type is mapped to a byte value starting from 10 to 20. The DWORD type is substituted with the byte value 10, the WORD type with the byte value 11 and so on. The parameter values, parameter strings and return values are converted into their corresponding byte values. Table D.4 indicates how a system call with n arguments is converted into a byte sequence. The parameter string is only included if the parameter type is of LPSTR or LPWSTR.

INT	syscall
BYTE	argtype$_1$
BYTES	argvalue$_1$
BYTES	argstring$_1$; only included if argtype$_1$ = LPSTR or LPWSTR
...	...
BYTE	argtype$_n$
BYTES	argvalue$_n$
BYTES	argstring$_n$; only included if argtype$_n$ = LPSTR or LPWSTR
BYTES	returnval

Table D.4: Illustrates how every system call is converted into a byte sequence.

We will now list the most important methods and describe their functionality:

- `APIParser()`
  Constructor for the APIParser class.
- `byte[] parse(java.lang.String file)`
  Parses the output text file from the APISPY program and returns the result as a byte array.
- `parseAPICall(java.lang.String s, ByteBuffer bb)`
  Parses an API call and writes it to a byte buffer.
- `parseParams(java.lang.String s, ByteBuffer bb)`
  Parses parameters and write them to a byte buffer.
- `parseReturnVal(java.lang.String s, ByteBuffer bb)`
  Parses return value and writes it to a byte buffer.
- `readAPISpecs(java.lang.String file)`
  Reads the API specification file defining the names of possible system calls.
- `setIncludeParamTypes(boolean b)`
  Sets whether or not to include parameter types when parsing.
- `setIncludeParams(boolean b)`
  Sets whether or not to include parameters when parsing.
- `setIncludeReturnVal(boolean b)`
  Sets whether or not to include return values when parsing.

D.4.1 Testing the APIParser class

We will in the following document the test of the APIParser class. We only test the `readAPISpecs` method and the `parse` method. This is because these are most important ones and because the `parse` method will use the three methods `parseAPICall`, `parseParams`, and `parseReturnVal` when parsing an APISPY output file, so these will be indirectly tested when testing the `parse` method.

Testing the readAPISpecs method

Here we simply read the API specification given in the file `apispec`. Every line in API specification file have the following syntax:

```
API dll_file function
```

All the function names are read from the file and saved in a hash table mapping the function names into integer values. Integer values for the function names are given according to which dll file they can be found in. Function names found in KERNEL32.dll will be mapped to numbers starting from 1000, function names in ADVAPI32.dll will be mapped to number starting from 2000, function names in COMDLG32.dll start at 3000, function names in GDI32.dll start at 4000, and function names in USER32.dll start at 5000.

Here we simply test to see that all the function names in the `apispec` file are saved in the hash table, and that they have a unique number according to the dll they can be found in. We will not list of the complete test because it would fill several pages, but table D.5 list some test examples indicating how the function names are mapped to integers.

dll	name	result	OK
KERNEL32.dll	`AddAtomA`	`AddAtomA=1000`	yes
KERNEL32.dll	`lstrlenW`	`lstrlenW=1470`	yes
ADVAPI32.dll	`RegCloseKey`	`RegCloseKey=2000`	yes
ADVAPI32.dll	`RegUnLoadKeyW`	`RegUnLoadKeyW=2046`	yes
COMDLG32.dll	`ChooseColorA`	`ChooseColorA=3000`	yes
COMDLG32.dll	`ReplaceTextW`	`ReplaceTextW=3016`	yes
GDI32.dll	`AbortDoc`	`AbortDoc=4000`	yes
GDI32.dll	`WidenPath`	`WidenPath=4280`	yes
USER32.dll	`ActivateKeyboardLayout`	`ActivateKeyboardLayout=5000`	yes
USER32.dll	`wvsprintfW`	`wvsprintfW=5466`	yes

Table D.5: Test examples of mapping API function names into integers.

As we can see from table D.5 the `readAPISpecs` method is working correctly.

Testing the parse method

When testing the `parse` method, we verify that the method works by running it on some example APISPY output files. Furthermore we verify that we can exclude parameter types, parameter values, and return values from the byte array returned by the method.

Table D.6 on the next page shows the results on running the `parse` method on the following small output from the APISPY program:

```
RegisterClassA(LPDATA:0063FD4C)
RegisterClassA returns: C251
```

The `RegisterClassA` name is mapped to the integer value 5338, which corresponds to the byte sequence `0 0 20 -38` in Java. In Java the byte values are represented by values ranging from -2^7 to 2^7-1, so if we want to represent bytes ranging from 0 to 255 we have to add 256 to the negative values: `-38` is the same as `218` (-38+256). In this way we can realise that 5338 corresponds to the byte sequence `0 0 20 -38` because:

$$0 \times 2^{24} + 0 \times 2^{16} + 20 \times 2^{8} + 218 \times 2^{0} = 5338.$$

The type `LPDATA` is mapped to the byte value 15. The hexadecimal value `0063FD4C` corresponds to the byte sequence `0 99 -3 76` because:

$\texttt{00}_{16}$	$\texttt{0}_{10}$
$\texttt{63}_{16}$	$\texttt{99}_{10}$
$\texttt{FD}_{16}$	$\texttt{253}_{10}$
$\texttt{4C}_{16}$	$\texttt{76}_{10}$

Here the byte value 253 is represented by `-3` in Java (253-256). Finally the value `C251` corresponds to `0 0 -62 81`. All parameter values and return values are aligned to four bytes.

When testing the `parse` method on the small output given above, we first parse all available information, then we exclude the parameter type, the parameter value, the return value, and finally we only include the system call.

option	result	OK
all info	`0 0 20 -38 15 0 99 -3 76 0 0 -62 81`	yes
no type	`0 0 20 -38 0 99 -3 76 0 0 -62 81`	yes
no value	`0 0 20 -38 15 0 0 -62 81`	yes
no return value	`0 0 20 -38 15 0 99 -3 76`	yes
only system call	`0 0 20 -38`	yes

Table D.6: Results on testing the parse method with different kinds of options.

We have also tested the `parse` method on several other kinds of output files from the APISPY program, the results and input files are rather big so we will not include them here, just conclude that the `parse` method works as expected.

D.4.2 Using the APIParser Class

To illustrate how the APIParser class could be used, we have included a small example:

```
// Create a new APIParser object.
APIParser p = new APIParser();

// Read the API specification from file.
p.readAPISpecs("CIS/apispec");

// The output file generated by APISPY program.
String file ="data/syscall/PING.out";

// Do not parse parameter types, values, and return values.
p.setIncludeParamTypes(false);
p.setIncludeParams(false);
p.setIncludeReturnVal(false);

// Parse APISPY output file.
```

```
byte b[] = p.parse(file);

// Write byte array to file.
try{
    FileOutputStream out=new FileOutputStream(file+".bin");
    out.write(b);
    out.flush();
    out.close();
}
catch(Exception e){e.printStackTrace();}
```

D.5 The ByteBuffer Class

The ByteBuffer class provides a byte buffer and methods for writing bytes, hexadecimal values, integers, and strings to it. The ByteBuffer class is used by the APIParser class to write information into a byte array. The ByteBuffer class provides the following methods:

- `ByteBuffer(int capacity)`
  Creates a new byte buffer with the specified capacity.
- `writeByte(byte b)`
  Writes a byte value to the byte buffer.
- `writeHexString(java.lang.String s)`
  Writes a hexadecimal number represented by a string object to the byte buffer.
- `writeInt(int i)`
  Writes an integer value to the byte buffer.
- `writeString(java.lang.String s)`
  Writes a string object to the byte buffer.
- `byte[] getBytes()`
  Returns the part of the byte buffer which has been used until now.

D.5.1 Testing the ByteBuffer Class

Table D.7 on the following page shows the test of the ByteBuffer class. As we can see the methods of the ByteBuffer class works as expected.

D.5.2 Using the ByteBuffer Class

To show how the ByteBuffer class could be used, we list a small example code:

```
ByteBuffer bb = new ByteBuffer(100);
bb.writeByte((byte) -128);
```

Method	Arguments	Results	OK
writeByte	`-128`	`-128`	yes
	`0`	`0`	yes
	`127`	`127`	yes
writeInt	`100`	`0 0 0 100`	yes
	`1000`	`0 0 3 -24`	yes
	`1000000000`	`59 -102 -54 0`	yes
	`Integer.MAX_VALUE`	`127 -1 -1 -1`	yes
writeHexString	`''''`	`0 0 0 0`	yes
	`''320''`	`0 0 3 32`	yes
	`''0102''`	`0 0 1 2`	yes
	`''FFFFFFFF''`	`-1 -1 -1 -1`	yes
	`''01020304''`	`1 2 3 4`	yes
writeString	`''''`		yes
	`''abc''`	`97 98 99`	yes
	`''ABC''`	`65 66 67`	yes
getBytes	`writeString("abcABC")`	`97 98 99 65 66 67`	yes

Table D.7: Testing results for some of the methods in the ByteBuffer class.

```
bb.writeInt(100);
bb.writeHexString("FFFFFFFF");
bb.writeString("abc");
byte[] b=bb.getBytes();
```

Glossary

adaptive immune system: the body's inner most defence system. It can recognise a much wider variety of pathogens than the innate immune system, but also have a much slower response.

affinity maturation: is when the affinity for one specific antigen increases. Developing more specific receptors for a certain antigen enables the cell to trigger an immune response more effectively and quickly.

antibody: a protein in blood that reacts to particular toxic substances by neutralising or destroying them, and thus provides immunity against them.

antigens: any of various substances that, when introduced into a living body, causes the production of antibodies.

apoptosis: or programmed cell death. Occurs in all tissues, at a relatively constant rate, and is a mean of regulating the number of cells in the body[1, p.18]].

autoimmune response: a responses where the immune system attacks itself.

B-cells: help the immune system in eliminating an infection. The B-cells secrete antibodies when activate, which help other cells in ingesting and neutralising the infection. B-cells are also known as B-lymphocytes.

clonal expansion: the adaptive immune system's ability to clone the lymphocytes and thereby increase the effect of adaptive immunity.

cytotoxic T- killer cells: T-cells, which are activated by mostly dendritic cells displaying antigen on their surfaces. The cytotoxic T-cell is able to kill other virus infected cells and thereby able to stop the replication of the virus.

dendritic cells: white blood cells that present antigens to lymphocytes.

eosinophils: white blood cells able to kill parasites coated with antibodies.

epitopes: are small regions on the pathogens. Receptors on the lymphocytes bind to these epitopes.

immunologically memory: the immune system's ability to provide the body with long lasting protective immunity.

innate immune system: the body's second inner most defence system. It can recognise a broad class of pathogens and trigger an immediate response.

lymphocytes: a certain kind of white blood cells. Responsible for detection or recognition and destruction of pathogens.

macrophages: cells advanced from white blood cells to ingest and engulf pathogens.

major histocompatibility complexes: molecules which bind small peptide fragments in a cell and display them at the cell's surface.

negative selection: is carried out by the body, in the bone marrow with respect to B-cells and in the thymus with respect to T-cells, to select those lymphocytes, which does not respond strongly to self-antigen.

neutrophils: white blood cells able to ingest and engulf pathogens

nonself: pathogens which are harmful to the body.

pathogens: disease-causing microorganisms. Four broad categories are defined: viruses, bacteria, pathogenic fungi and parasites[1].

phagocytes: cells that are able to ingest and engulf pathogens, and in most cases able to destroy them (macrophages and neutrophils).

positive selection: is carried out by the body, in the bone marrow with respect to B-cells and in the thymus with respect to T-cells, to select those lymphocytes, which does not respond strongly to self-antigen.

receptors: cover the surface of the lymphocytes and bind to antigens or ***epitopes***.

self: harmless substances, including normal functioning cells

T-cells: help the immune system in eliminating an infection, the receptors of the T-cell bind to pathogens and the T-cell gets activated. Once activated the T-cell can kill virus infected cells or help other cells in eliminating an infection. T-cells are also known as T-lymphocytes.

T-helper cells: T cells, which help other cells by activating them. The helper T-cell activates B-cells to secrete antibodies and macrophages to increase their production of antibacterial material.

thymus: an organ located in the upper part of the middle chest just behind the breastbone.

Bibliography

[1] Charles A Janeway, Paul Travers, Mark Walport, Mark Shlomchik
Immunobiology: The Immune System in Health and Disease, 5th Ed.,
Garland Publishing, 2001

[2] Jeffrey O. Kephart, Gregory B. Sorkin, David M. Chess and Steve R. White
Fighting Computer Viruses,
`http://www.sciam.com/1197issue/1197kephart.html`

[3] Jeffrey O. Kephart and William C. Arnold
Automatic Extraction of Computer Virus Signatures,
`http://www.research.ibm.com/antivirus/SciPapers/Kephart/VB94/vb94.html`,
In Proceedings of the 4th Virus Bulletin International Conference, R. Ford, ed., Virus Bulletin Ltd., Abingdon, England, 1994, pp. 178-184

[4] Jeffrey O. Kephart
A Biological Inspired Immune System for Computers,
`http://www.research.ibm.com/antivirus/SciPapers/Kephart/ALIFE4/alife4.distrib.html`,
Published in Artificial Life IV, Proceedings of the Fourth International Workshop on Synthesis and Simulation of Living Systems, Rodney A. Brooks and Pattie Maes, eds., MIT Press, Cambridge, Massachusetts, 1994, pp. 130-139

[5] Gerald Tesauro, Jeffrey O. Kephart, Gregory B. Sorkin
Neural Networks for Computer Virus recognition,
`http://www.research.ibm.com/antivirus/SciPapers/Tesauro/NeuralNets.html`,
Published in IEEE Expert, vol. 11, no. 4, Aug. 1996, pp. 5-6

[6] Jeffrey O. Kephart, Gregory B. Sorkin, Morton Swimmer, and Steve R. White
Blueprint for a Computer Immune System,
`http://www.research.ibm.com/antivirus/SciPapers/Kephart/VB97/index.html`,
Presented at the Virus Bulletin International Conference in San Francisco, California, October 1-3, 1997.

[7] Steven A. Hofmeyr, Stephanie Forrest, Anil Somayaji
Intrusion Detection using Sequences of System Calls,

`http://www.cs.unm.edu/~immsec/publications/ids.ps`,
Note: this is the pre-submission version. A somewhat later version of this paper was published in the Journal of Computer Security Vol. 6 (1998) pg 151-180

[8] Steven A. Hofmeyr, Stephanie Forrest, Anil Somayaji
A Sense of Self for Unix Processes,
`http://www.cs.unm.edu/~immsec/publications/ieee-sp-96-unix.ps`,
In Proceedings of the 1996 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, pp. 120-128 (1996)

[9] Stephanie Forrest, Anil Somayaji
Automated Response Using System-Call Delays,
`ftp://ftp.cs.unm.edu/pub/forrest/uss-2000.ps`,
Published in the Proceedings of the 9th USENIX Security Symposium (August 14-17,2000)

[10] Christina Warrender, Stephanie Forrest, Barak Pearlmutter
Detecting Intrusions Using System Calls: Alternative Data Models,
`http://www.cs.unm.edu/~immsec/publications/oakland-with-cite.ps`,
Published in the 1999 IEEE Symposium on Security and Privacy, IEEE Computer Society pp. 133-145 (1999)

[11] Steven A. Hofmeyr, Stephanie Forrest
Architecture for an Artificial Immune System
`ftp://ftp.cs.unm.edu/pub/forrest/hofmeyr_forrest.ps`,
Evolutionary Computation Journal 7 pp. 45-68 (2000)

[12] Steven A. Hofmeyr, Stephanie Forrest
Immunity by Design: An Artificial Immune System
`http://www.cs.unm.edu/~immsec/publications/gecco-steve.ps`,
Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), Morgan-Kaufmann, San Francisco, CA, pp. 1289-1296 (1999).

[13] Stephanie Forrest and Steven A. Hofmeyr
Immunology as Information Processing
`ftp://ftp.cs.unm.edu/pub/forrest/iaip.ps`
Design Principles for the Immune System and Other Distributed Autonomous Systems, L.A. Segel and I. Cohen, eds. Oxford Univ. Press 2000.

[14] Anil Somayaji, Steven Hofmeyr, and Stephanie Forrest
Principles of a Computer Immune System
`http://www.cs.unm.edu/~immsec/publications/nspw-97.ps`
1997 New Security Paradigms Workshop Langdale, ACM pp75-82, ACM (1998).

[15] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji
Computer Immunology
Communications of the ACM Vol. 40, No. 10, pp. 88-96 (1997).

[16] Stephanie Forrest, Anil Somayaji, and David H. Ackley
Building Diverse Computer Systems
`http://www.cs.unm.edu/~immsec/publications/hotos-97.ps`

In Proceedings of the Sixth Workshop on Hot Topics in Operating Systems, Computer Society Press,Los Alamitos, CA, pp. 67-72 (1997).

[17] Stephanie Forrest, Allan S. Perelson, Lawrence Allen, and Rajesh Cherukuri
Self-Nonself Discrimination in a Computer
`http://www.cs.unm.edu/~immsec/publications/virus.ps`
In Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy, Los Alamitos, CA: IEEE Computer Society Press (1994).

[18] Patrick D'haeseleer, Stephanie Forrest, and Paul Helman
An Immunological Approach to Change Detection: Algorithms, Analysis, and Implications
`http://www.cs.unm.edu/~immsec/publications/`
`ieee-sp-96-neg-selec.ps`
In Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy (1996).

[19] Patrick D'haeseleer
An Immunological Approach to Change Detection: Theoretical Results
`http://www.cs.unm.edu/~immsec/publications/CSFW96.ps`
In 9th IEEE Computer Security Foundations Workshop (1996).

[20] Computer Associates Virus Information Center
A primer for understanding computer virus basics,
`http://www3.ca.com/Solutions/Collateral.asp?ID=897\&PID=`

[21] Computer Associates Virus Information Center
An in-depth look at Macro Viruses,
`http://www3.ca.com/Solutions/Collateral.asp?ID=913&PID=`

[22] Sophos Virus Info
Computer virus prevention: a primer,
`http://www.sophos.com/sophos/docs/eng/papers/artd_wen.pdf`

[23] Sophos Virus Info
An introduction to computer viruses,
`http://www.sophos.com/sophos/docs/eng/papers/int_wen.pdf`

[24] Sophos Virus Info
Computer viruses demystified,
`http://www.sophos.com/sophos/docs/eng/refguide/viru_ben.pdf`

[25] Sophos Virus Info
Glossary of terms,
`http://www.sophos.com/virusinfo/articles/glossary.html`

[26] AVP Virus Encyclopedia
Computer Viruses - what are they and how to fight them?,
`http://www.avp.ch/avpve/entry/entry1.htm`

[27] AVP Virus Encyclopedia
The Classification of Computer Viruses,
`http://www.avp.ch/avpve/classes/classes.stm`

[28] L. R. Rabiner, B. H. Juang
An Introduction to Hidden Markov Models,
IEEE ASSP Magazine, volume 3, number 1, p. 4-16, January 1986

[29] Lawrence R. Rabiner
A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition,
Proceedings of the IEEE, vol. 77, no. 2, February 1989

[30] Anders Krogh
Chapter 4
An Introduction to Hidden Markov Models for Biological Sequences,
Computational Methods in Molecular Biology, Elsevier, 1998