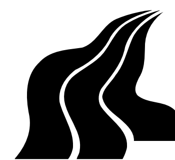
AALBORG UNIVERSITY

Error Correction of Logistical Data

- Using a Distributed Viterbi Approach

Studyboard for Electronics and Information Technology
Aalborg University
M.Sc. Thesis
Group 1033
2007-2008

Aalborg Universitet



Studienævn for Elektronik og Informations Teknologi

TITEL:
Error Correction of Logistical Data
- Using a Distributed Viterbi Approach

PROJEKTPERIODE:
3. september 2007 - 4. juni 2008

GRUPPE:
1033

MEDLEMMER:
Tom Nørgaard Jensen
Mads Philipsen

VEJLEDERE:
Jakob Stoustrup
Henrik Schiøler

OPLAG: 5

SIDER: 79

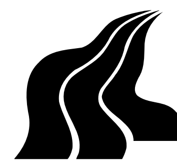
APPENDICES: 10

AFSLUTTET: 4. juni 2008

SYNOPSIS:

Rapporten omhandler et distributions system, hvor adskillige aktiver cirkulerer mellem flere forskellige parter. Aktiverne bliver registreret når de bliver modtaget eller afsendt af en af parterne i systemet. Denne registrering foregår ved hjælp af RFID teknologi, hvilket betyder at registreringerne kan være fejlbehæftede. Da parterne i systemet er forpligtet til at betale pant for aktiverne, er det vigtigt at panten er beregnet på et korrekt grundlag. Derfor er det ønskeligt at rette data, således aktivernes lokation kan bestemmes selv hvis nogle registreringer er manglende.
Til dette formål er der blevet konstrueret en estimator, der kan estimere aktivernes sekvens af lokationer baseret på de fejlbehæftede data. Denne estimator er baseret på de statistiske egenskaber ved systemet, som kan modelleres som en skjult Markov kæde.
Derudover er en simulerings model blevet konstrueret, således det er muligt at teste om estimatoren får bestemt den korrekte sekvens af lokationer.
Estimatoren er blevet testet med et setup i mindre målestok og resultaterne herfra er ekstrapoleret til et fuld skala system. Beregningstiden for systemet viser at algoritmen er brugbar, såfremt den distribueres imellem flere processorer. Software til et distribueret system, bestående af elleve lokationer er blevet designet og implementeret i C++. Test resultater indikerer at estimatoren ikke er i stand til at opfylde kravet til fejl procent i sin nuværende tilstand.

Aalborg University



Studyboard for Electronics and Information Technology

TITLE:
Error Correction of Logistical Data
- Using a Distributed Viterbi Approach

PROJECTPERIOD:
September 3 2007 - June 4 2008

GROUP:
1033

PARTICIPANTS:
Tom Nørgaard Jensen
Mads Philipsen

SUPERVISORS:
Jakob Stoustrup
Henrik Schiøler

NUMBER PRINTED:5

NUMBER OF PAGES: 79

NUMBER OF APPENDICES: 10

CONCLUDED: 4th of June 2008

ABSTRACT:

The report deals with a distribution system, where several assets circulate between multiple parties. The assets are registered when being received or returned by one of the parties in the system. This registration is done using RFID technology, which means that the registrations are prone to error. Since the parties in the system are obliged to pay deposit for the assets, it is important that the deposits are calculated on the correct basis. Therefor, it is desirable to correct the data, such that the whereabouts of the assets can be determined even if some registrations are missing.
For this purpose an estimator, which can estimate the location sequence of the assets based on the erroneous data, has been developed. The estimator is based on the statistical properties of the distribution system, which can be modelled as a hidden Markov chain.
Furthermore, a simulation model of the distribution system has been constructed, in order to test if the estimator is able to determine the correct location sequence.
The estimator has been tested in a small scale system. The results have been extrapolated to cover a full scale system. The computational time for the large scale system, show that the algorithm is only usable when distributed between multiple processors. Software for a distributed setup consisting of eleven locations has been designed and implemented in C++. Test results indicate that the estimator in its current condition is not able to fulfil the requirement to its error percentage.

Preface

This project has been carried out by group 1033, during the 9th and 10th semester at the section of 'Automation and Control' at Aalborg University, in cooperation with Lyngsoe Systems. The work has been carried out between 3rd of September 2007 and 4th of June 2008.

The project proposal has been provided by Lyngsoe Systems, where the contact have been Section Manager Jørgen Albøge.

The report uses the Harvard Reference System, where the reference in the report consists of a (Last name, Year). On page 79 a complete bibliography can be found. In references to tables, figures and equations, the first digit specifies the chapter. For instance; Table: 2.1 is the first table in chapter two. Appendices referenced throughout the report, are found at the end of the report.

CD-ROM Contents: The enclosed CD-ROM contains programs, source code and the corresponding doxygen generated documentation. Measurement files, Matlab scripts, Matlab figures and the Simulink models are also included on the CD-ROM. The CD-ROM is divided into folders corresponding the chapters of the report. The contents includes README-files containing a description of the files. The report is also included in PDF and Postscript formats.

Tom Nørgaard Jensen   Mads Philipsen

Contents

1 Introduction **11**

1.1 Description of Distribution System 11

1.2 Requirements for Estimator 12

1.3 Limitations from Requirements 13

1.4 Methods 13

1.5 Outline of Report 13

2 Modelling of System **15**

2.1 Discrete Event Systems 15

2.2 Markov Chains 23

2.3 Validation of Simulation Model 33

3 Simulink Model of System **37**

3.1 Simulink Model 37

3.2 Validation of Simulink Model 42

4 Estimator Design **43**

4.1 Viterbi Algorithm 43

4.2 Customised Viterbi Algorithm 47

4.3 Graphical User Interface 54

5 Distribution of Algorithm **55**

5.1 Distribution Paradigms 55

5.2 Load Balancing 56

6 Distributed Software **61**

6.1 Description of Events 61

6.2 Analysis of Distributed Software 62

6.3 Implementation of Distributed Algorithm 69

6.4 Test of Distributed Software 71

7 Conclusion **77**

7.1 Future Work 77

Bibliography **79**

Appendices **79**

A Rank-Sum Test **81**

B Validation of Hidden Markov Model **83**

C Validation of Simulink Model **87**

D Simulink Model **89**

E Test of Viterbi Implementation **95**

F Test of Time Consumption of Viterbi Algorithm **99**

G Test of Error Rate of Viterbi Algorithm **103**

H Test of Time Consumption of Custom Viterbi Algorithm **105**

I Test of C++ Implementation of Viterbi Algorithm **109**

J Test of Load Balancing Algorithm **115**

K Graphical User Interface for Model and Algorithms **119**

Chapter 1

Introduction

This work treats the design of an model based estimator. Prior to the design of the estimator, the system is modelled and validated. Afterwards an estimator is designed and tested in the following chapters. This chapter a describes the distribution system, after which the requirements for the estimator are listed and explained. This chapter is concluded with a section describing the various methods used in this work.

1.1 Description of Distribution System

In a closed system, where assets circulate between multiple parties, it can be desirable to make a registration, which can document the location of the assets as a function of time. This registration can be used for multiple purposes. For instance; the calculation of the deposit of the assets, or to ensure adequate stock at the participants. A practical example of such a system is the distribution system of the cc container seen in Figure 1.1 and 1.2, which is a flower and pot plant trolley. These trolleys are among others distributed between garden centres, auction houses and stores all over Europe. The system consist of approximately 4e6 assets, and 20e3 locations. Part of the distribution network is illustrated in Figure 1.3, where an example of a transportation of cc's is shown. One part of the figure indicates a number of random cc's being transported from a garden centre to an auction house. The other part indicates a transport of a new number of random cc's being transported from the auction house to a store that has bought plants from the auction house. More generally cc's can be transported between all the participants in the system.
The number of cc's that a participant possess at a given time, can then form the basis for the payment of deposit. The registration of when the containers are transported from one participant to another is based on RFID technology. The system consists of one or more reading ports at a given participant, and an RFID transmitter mounted on each of the cc's in the system. The reading ports are placed such that it is necessary to pass them when delivering or picking up cc's at the participant.The registration data consists of a port number an RFID tag number and a time stamp. These RFID data are collected decentralised at the participants, and are from here transferred to a central server, every now and then. The data can in some cases be prone to error, and these errors will result in wrong calculations of the deposits.
In the system, the following sources of errors have been identified:

- The cc's pass through the reading portal, without being registered. It is expected that more than 95 % of the cc's will be registered.
- The cc's are registered without having passed through the reading portal. For instance if they are placed close to a reading port without actually being transported.
- Some cc's have a faulty RFID tag, due to this they are not registered or only registered rarely. The RFID tag, will be equipped with a bar code, which will add the possibility for manual registration of cc's caring a faulty tag.

Figure 1.1: *The cc container*

Figure 1.2: *A loaded cc container*

- Cc's introduced by third party suppliers, which can have identification coinciding with the original cc's.

An approach for solving the above mentioned problems is to construct a model of the distribution system. The model is used as a basis for the construction of an estimator. The estimator uses RFID data to estimate the state of the system, which is used for error correction of the data.
A distribution system like this is inherently of the type Discrete Event System, where the state space of the system is naturally described by a discrete set, and where state transitions are observed at discrete points in time. In the distribution system described above, the state of the system could be the number of cc's possessed by the participants at a given time or it could be the location of a single cc, depending on the level of abstraction. The events triggering the state transition in the system could be the RFID readings or the transports between participants.

1.2 Requirements for Estimator

The main purpose of the estimator is to determine the amount of cc's at each of the participants in the distribution system at any given time, in order to make the deposit claims correct. This should be done using the possibly faulty RFID readings from the RFID reader gates, which are placed at every participant. From the supplier of the cc containers it is required that the data are correct in 98.75 % of the cases.
Furthermore, the estimator should be able to detect if certain RFID tags are broken, and if possible locate these such that it is possible to remove these tags from the system. The RFID tags which have been copied and introduced by third party suppliers should likewise be detected and located with the aim of removing them from the system. The estimator should be feasible to implement in a system consisting of 4e6 assets and 20e3 locations, since this is the scale of the cc distribution system.

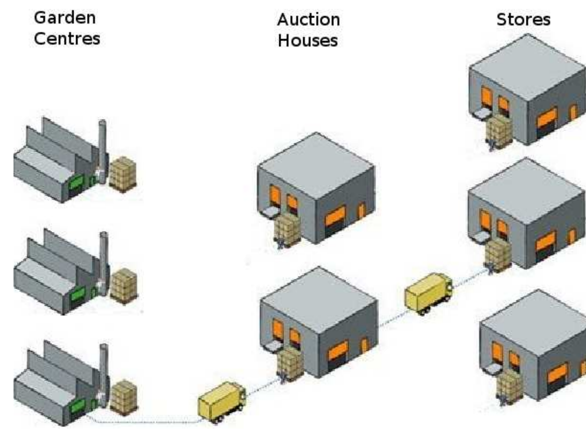


Figure 1.3: *Illustration of part of the distribution system*

1.3 Limitations from Requirements

It has been chosen to focus on the state estimation during the project work, since this issues can be handled by a discrete event system model based estimator.The other issues, concerning error handling in the system would require a fault detection and isolation (FDI) algorithm. Since it is assessed that the design of an estimator for a system in the magnitude (4e6 assets and 20e3 parties) of the distribution of cc containers throughout Europe is extensive enough to cover the entire project period, there will not be focused on designing an FDI algorithm.

1.4 Methods

This section describes the methods used in the work presented in this master thesis. A problem has been presented by Lygsoe Systems, concerning error correction of logistical data. An approach to solve this problem has been analysed, and the solution is based on model based estimation, using a customised Viterbi algorithm.

An analysis of the system in question has been carried out based on an interview of Jørgen Albøge from Lyngsoe Systems, and a description of the properties of the distribution system has been constructed. This description has been used to make a requirement description. Two simulation models of the distribution system, has been made. First a hidden Markov model has been implemented in Matlab. Secondly a simulation model has been constructed in Simulink based on the system properties. The Simulink model models interdependent assets. The simulation models has been verified using a rank sum test.

The customised Viterbi algorithm has been changed to support distribution on multiple processors. Furthermore, software for the distributed algorithm has been analysed, using an OO software analysis approach. UML diagrams has been constructed as documentation of the analysis and design of the software. The software has been implemented in C++.

1.5 Outline of Report

Chapter 2: Discrete Event Systems The chapter covers the discrete event modelling frameworks, automata, petri nets and hidden Markov models, and modelling of the distribution system as a hidden Markov model.

Chapter 3: Modeling of Distribution System The chapter covers the description and validation of the Simulink model.

Chapter 4: Estimator Design The chapter describes the analysis and implementation of the Viterbi algorithm, as well as the customised Viterbi. Various test comparing the two algorithms are included in the chapter.

Chapter 5: Distribution of Algorithm The chapter describes the analysis and choice of distribution paradigms, which is used in the distribution of the algorithm.

Chapter 6: Distributed software The chapter covers the analysis, design and implementation of the software needed for the distribution of the algorithm.

Chapter 7: Conclusion The chapter includes the conclusion of the project, and a future work section.

Chapter 2

Modelling of System

This chapter describes the modelling of the distribution system in question. The chapter starts with an introduction to different modelling formalisms associated with the modelling of discrete event systems. This leads to a choice of the appropriate modelling formalism for the distribution of cc containers. The system is then modelled using the chosen formalism, and the model is then validated using output from an actual system.

2.1 Discrete Event Systems

This section presents an analysis of two different modelling approaches for modelling discrete event systems (DES). The section includes an introduction to the automata and petri net approach. After the introduction to the different terminologies, which are used in the two approaches, different kinds of modelling examples will be introduced, in order to find the approach which will be suitable for the modelling of the cc container distribution system.

2.1.1 Automata

This section describes the automata concept, and the properties of automata, this will be used later to determine which modelling approach is the most appropriate for modelling the distribution system. The section is based on (Cassandras and Lafortune, 1999).

Definition of Automaton

The automaton is used to represent a DES both by a graph and by the use of equations. The graphical representation of the DES is the simplest way to represent the automaton, an example of a graphical representation is shown in Figure 2.1, which illustrate a state transition diagram. The nodes in the diagram represent the state set X of the DES. The arcs between the nodes, represent state transitions, where the labels on the arcs represent events from the event set E of the automaton. The arcs of the state diagram represent the transition function of the automaton, which is denoted $f : X \times E \rightarrow X$. The transition function, describes the trajectory from the current state based on the event set. The automaton shown in Figure 2.1, correspond to the transition function f:

$$\begin{array}{rr} f(x,a) = x & f(x,g) = z \\ f(y,a) = x & f(y,b) = y \\ f(z,b) = z & f(z,a) = f(z,g) = y \end{array} \tag{2.1}$$

where $f(y, a) = x$ means that if the automaton is in state y, and event a occurs, the automaton will make a transition to the state x. The initial state of the automaton is denoted by x_0, which is illustrated by an arc, coming from nowhere. The diagram also includes a set of marked states X_m, which is a subset of X. Marked states are acceptable final states. The marked states X_m are illustrated by double circles.

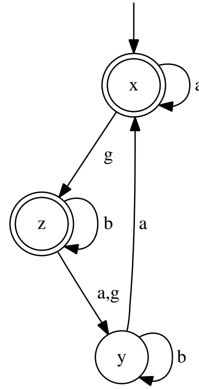


Figure 2.1: *Illustration of automaton state transition diagram*

Definition (Deterministic automaton) A deterministic automaton, G is defined as:

$$G = (X, E, f, \Gamma, x_0, X_m) \tag{2.2}$$

where:

X is the set of states
E is the finite set of events, which can cause transitions in G
$f : X \times E \to X$ is a transition function $f(x, e) = y$ which means that there is a transition labelled by event e from state x to state y
$\Gamma : X \to 2^E$ is the active event function; $\Gamma(x)$ is the set of all events e for which $f(x, e)$ is defined and it is called the active event set of G at x
x_0 is the initial state
$X_m \subseteq X$ is the set of marked states

The automaton G operates as follows. It starts in the initial state x_0 and upon an occurrence of an event $e \in \Gamma(x_0) \subseteq E$ it will make at transition to state $f(x_0, e) \in X$. This process continues based on the transitions for which f is defined. The automaton is called deterministic because f is a function over $X \times E$ to X, which means that the transition from a state is uniquely determined by the event that triggers the transition.

Definition (Non-deterministic automaton) A non-deterministic automaton, denoted by G_{nd} is defined as

$$G_{nd} = (X, E \cup \{\varepsilon\}, f_{nd}, \Gamma, x_0, X_m) \tag{2.3}$$

where these element have the same interpretation as for the deterministic version, except for f_{nd} and x_0

> f_{nd} is a function $f_{nd} : X \times E \cup \{\varepsilon\} \to 2^X$; that is: $f_{nd}(x, e) \subseteq X$ whenever it is defined. This means that an event can lead to multiple different states.
>
> The initial state x_0 may itself be a set of states, that is $x_0 \subseteq X$

The automaton is called non-deterministic because the transition from a state is not uniquely determined by the event triggering the transition, since the output of f is set valued.
This concludes the basic principles used in automata, the next section deals with the modelling of DES using petri nets.

2.1.2 Petri Nets

This sections focus is on a modelling formalism for discrete event systems known as petri nets. Petri nets can represent a larger class of systems than finite state automata. On the other hand petri nets does not have the advantage of having systematic rules or operations for combination of one or more petri nets, as is the case with automata. (Cassandras and Lafortune, 1999)

Definition of a Petri Net

A petri net consists of three basic elements; places, transitions and the relations between them. In petri nets events are related to the transitions, and places in the petri net are related to the given conditions that has to be fulfilled in order for the event/transition to occur. A place in the petri net can be both input and output to a transition. The relations between places and transitions are indicated by arcs between them, the arcs can have different weights.
In Figure 2.2 an example of a petri net graph is illustrated. In the figure the places are symbolised with a circle and the transitions are symbolised with a bar. The arcs in the figure symbolise the relations between the places and the transitions. The weight of the arcs are illustrated by writing them on top of the arc, hence the arc between place p_1 and transition t_1 in Figure 2.2 has the weight 2. If the weight is 1 no number is written as in the case of the weight of the arc between transition t_1 and place p_2 in Figure 2.2. As an alternative the petri net in Figure 2.2 can also be illustrated as in Figure 2.3, where the weights are indicated by the number of arcs between places and transitions. The two petri net graphs are thereby equivalent. A petri net structure P_N is defined as a four tuple

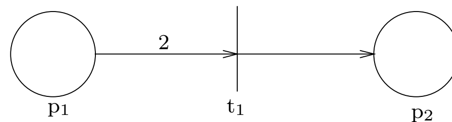


Figure 2.2: *Example petri net graph*

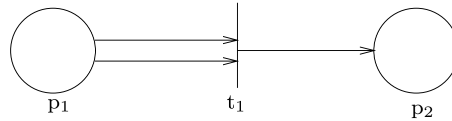


Figure 2.3: *Example petri net graph equivalent to Figure 2.2*

in the following way:

$$P_N = (P, T, A, w) \tag{2.4}$$

Where:
- P is the set of places
- T is the set of transitions
- A is the set of arcs
- w is the set of weights on the arcs

The set P of places and the set T of transitions in the petri net structure P_N is represented by:

$$P = \{p_1, p_2, \cdots, p_{n-1}, p_n\} \tag{2.5}$$
$$T = \{t_1, t_2, \cdots, t_{m-1}, t_m\} \tag{2.6}$$

The set A of arcs in the petri net structure is represented by:

$$A = \{\cdots, (p_i, t_j), \cdots, (t_j, p_i), \cdots\} \tag{2.7}$$

Where:
- (p_i, t_j) is an arc from place p_i to transition t_j
- (t_j, p_i) is an arc from transition t_j to place p_i

Finally, the set w of weights in the petri net structure will be represented by:

$$w = \{\cdots, w(p_i, t_j), \cdots, w(t_j, p_i), \cdots\} \tag{2.8}$$

It follows from the definition in (2.7), that the sets A and w contains at most $2nm$ elements.
As an example, the petri net structure P_{N1} illustrated in Figure 2.2 can be written as following:

$$P_{N1} = (P, T, A, w), \tag{2.9}$$
$$P = \{p_1, p_2\} \tag{2.10}$$
$$T = \{t_1\} \tag{2.11}$$
$$A = \{(p_1, t_1), (t_1, p_2)\} \tag{2.12}$$
$$w = \{2, 1\} \tag{2.13}$$

So far the structure of the petri net has been considered, the following section covers the dynamics of petri nets.

Petri Net State Dynamics

In order to be able to describe the dynamics of the petri net, information about the state of the net is needed. This information is obtained by introducing the concept of marking to the petri net.
The marking of a petri net is defined as an n-dimensional row vector x consisting of non-negative integers, describing the markings of each of the places p_i in the petri net. The marking contain information about when a given event/transition is enabled in the petri net. The vector x is given by:

$$x = \{x(p_1), \cdots, x(p_n)\} \tag{2.14}$$

When introducing the marking vector x to the petri net structure, the net is referred to as a marked petri net or just a petri net for simplicity. The marked petri net P_N is a five tuple defined as:

$$P_N = (P, T, A, w, x) \tag{2.15}$$

The state of the petri net is then defined as the marking vector. A transition is enabled in the petri net if the marking of all places connected as inputs to a given transition is greater than or equal to the weight on the arc connecting the place to the transition. This can be expressed as:

$$x(p_i) \geq w(p_i, t_j), \ \forall p_i \in P : (p_i, t_j) \in A \tag{2.16}$$

The marking of the petri net can be introduced to the petri net graph, by using the concept of tokens. This is illustrated in Figure 2.4, where each dot inside the places in the graph is an illustration of a token. The marking or state vector x of the petri net illustrated in Figure 2.4 can be verified to be:

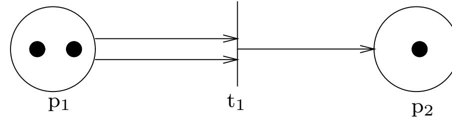


Figure 2.4: *Illustration of a petri net graph including tokens*

$$x = \{2, 1\} \tag{2.17}$$

corresponding to the number of tokens in each place p_i in the petri net.
When the requirement described in Equation (2.16) is fulfilled, the transition t_j is allowed to occur. This is also referred to as firing of the transition.
When the transition fires it consumes tokens from the places connected as inputs corresponding to the weights of the arcs connecting the places to the transition. Additionally the transition adds tokens to the places connected as outputs to the transition corresponding to the weight of the arcs connecting the transition to the place. This can be expressed for place p_i and transition t_j as:

$$x'(p_i) = x(p_i) - w(p_i, t_j) + w(t_j, p_i) \tag{2.18}$$

Where:
$x'(p_i)$ is the new marking of place p_i

If there is no connection between p_i and t_j, the two weights w in (2.18) are evaluated as being zero. From (2.18) it can be seen that tokens are generally not conserved in petri nets, since it is possible that the sum of the weights of places connected as inputs to a transition is different than the sum of the weights of places connected as outputs from a transition, that is:

$$\sum_{p_i \in P} w(p_i, t_j) > \sum_{p_i \in P} w(t_j, p_i), \text{ or } \sum_{p_i \in P} w(p_i, t_j) < \sum_{p_i \in P} w(t_j, p_i) \tag{2.19}$$

Because of this it is possible for a finite petri net graph to have an infinite number of states, which is opposite the case with finite state automata.
To illustrate the state dynamics of petri nets, an example will now be given. (Cassandras and Lafortune, 1999) In Figure 2.5 a graph of a petri net is shown in its initial state x_0. As it is apparent from the figure it is only transition t_1 that is enabled. If it is now considered that transition t_1 fires,

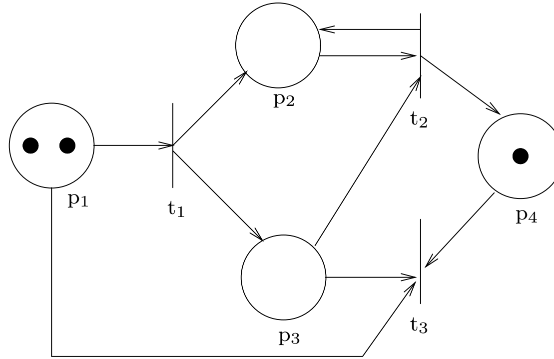


Figure 2.5: *Illustration of a petri net in initial state x_0*

the resulting state x_1 of the petri net will be as illustrated in Figure 2.6. Since p_1 is input to transition t_1, one token is removed from this place. A token is added to both p_2 and p_3, since these places are outputs from t_1.
In the new state x_1 it is possible for either of the transitions t_1, t_2 and t_3 to fire, but only one at a time. Figure 2.7 illustrates the new state x_2 of the petri net if transition t_2 is fired from the state

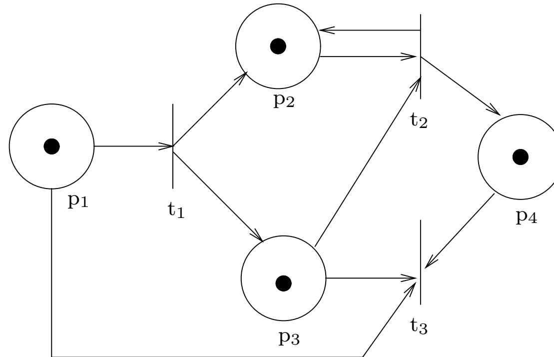


Figure 2.6: *Resulting state x_1 of the petri net upon firing of transition t_1 in state x_0*

x_1. As can be seen a token is removed from p_2 and p_3, and a token is added to p_2 and p_4. If the transition t_3 is fired from state x_1 instead, the resulting state x_2' will be as illustrated in Figure 2.8. Here a token has been removed from p_1, p_3 and p_4 and no tokens are added since t_3 has no outputs. As it is apparent from the figure no more transitions can occur in this state, and it is therefor referred to as a deadlock state of the petri net.
The above described behaviour can be captured in a simple linear equation, describing the relation between the current and the resulting state upon the firing of a transition. First a firing vector u is defined as an m-dimensional row vector in the following manner:

$$u = [0, \cdots, 0, 1, 0, \cdots, 0] \tag{2.20}$$

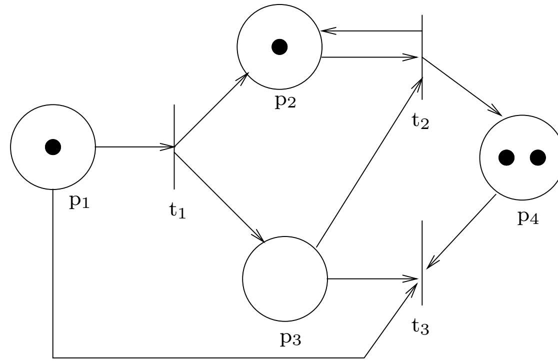


Figure 2.7: *Resulting state x_2 of the petri net upon firing of transition t_2 in state x_1*

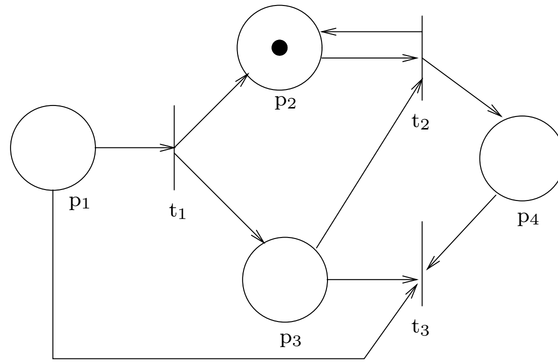


Figure 2.8: *Resulting state x_2' of the petri net upon firing of transition t_3 in state x_1*

The only nonzero entry appears in the j'th place, indicating that it is the j'th transition that fires, this entry has the value 1. The nonzero entry of the firing vector is of course limited to transitions that are enabled. Next the incidence matrix A of the petri net is defined. This is an $m \times n$ matrix, where the (j, i) entry is given by the following:

$$a_{ji} = w(t_j, p_i) - w(p_i, t_j) \tag{2.21}$$

That is for every row in the matrix, the token balance is given for each of the places in the petri net given that the transition fires.
Now it is possible to describe the relation between the current state x and the resulting state x', given the firing vector and the incidence matrix:

$$x' = x + uA \tag{2.22}$$

As an example one can consider the transition from state x_0 illustrated in Figure 2.5 to state x_1 illustrated in Figure 2.6. First the incidence matrix A is established by inspecting the petri net graph, and using the definition in (2.21):

$$A = \begin{bmatrix} -1 & 1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & -1 & -1 \end{bmatrix} \tag{2.23}$$

Then by firing transition t_1 in state x_0 it is possible to calculate the state x_1:

$$x_1 = x_0 + uA \Leftrightarrow \tag{2.24}$$

$$x_1 = [2\,0\,0\,1] + [1\,0\,0] \begin{bmatrix} -1 & 1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & -1 & -1 \end{bmatrix} \Leftrightarrow \tag{2.25}$$

$$x_1 = [1\,1\,1\,1] \tag{2.26}$$

Which can be verified in Figure 2.6. Now that both modelling formalism has been described, the choice of the formalism appropriate for the distribution system will be made.

2.1.3 Choice of Modelling Framework

This section describes some of the considerations that has been done in order to choose the proper modelling framework for the distribution system. As described in the previous sections two different modelling frameworks has been considered; automata and petri nets.

Petri Net Model of Distribution Network

An intuitive approach for modelling the distribution of assets in the system, is to have places p_i in the petri net be a representation of a warehouse or a customer. In this abstraction a token in a given place would then represent a single asset. In this framework the network structure is unknown in between samples, i.e; the weights between places and transitions are unknown, since the amount of assets moved from warehouse to warehouse in between samples can change. This is illustrated in Figure 2.9, where the question marks above the arcs indicate unknown weights.
One possible solution to this problem could be to augment the petri net with additional places s_{ij}

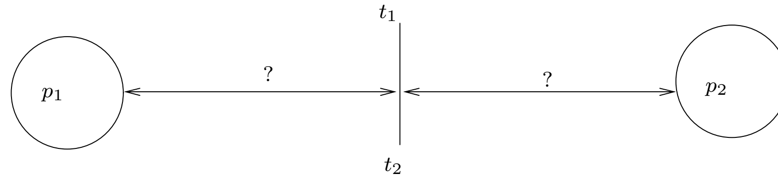


Figure 2.9: *Petri net with unknown weights. The double arrows indicate that the bar symbolises the two different transitions t_1 and t_2.*

which contain information about how many tokens to move from one place to another in between two samples. This is illustrated in Figure 2.10. In this framework the individual s_{ij} then have to be 'filled' with the number of tokens to move, by some mechanism in between samples, which violates the petri net modelling framework. Alternatively one could have transitions $t_1 \cdots t_i \cdots t_n$ between two places, where i indicates the number of tokens that have been moved and n is the maximum number of assets it is possible to possess, and the firing transition would have weights s_{ij} both in and out. The problem

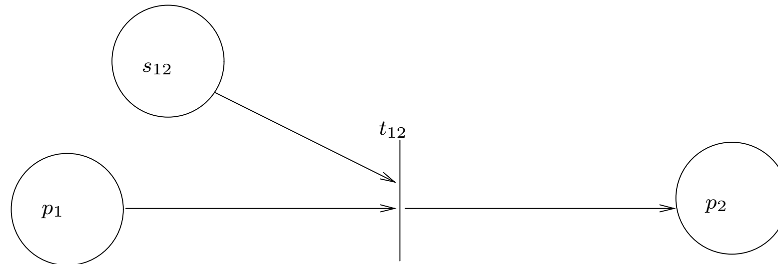


Figure 2.10: *Petri net with weight determined by s_{ij}*

is now to determine the marking of s_{ij}; that is: number of tokens in s_{ij} in between two consecutive samples. Which again indicates the weights of the arcs in between two samples.
Furthermore, it is possible for tokens (assets) to disappear from one place to turn up again a different place, which indicates that a reading has been missed. In order to implement this in the model, the set of places can be augmented by a place p_L containing all the 'Lost' tokens. This is illustrated in Figure 2.11.
It is now possible to obtain expressions for the relation between the change in the state of the petri net and the individual s_{ij}. First the number of tokens $x^+(p_1)$ gained by place p_1 between samples can be expressed as:

$$x^+(p_1) = s_{21} + s_{31} + \cdots + s_{n1} + s_{L1} \tag{2.27}$$

The number of tokens $x^-(p_1)$ lost by place p_1 between samples can likewise be expressed as:

$$x^-(p_1) = s_{12} + s_{13} + \cdots + s_{1n} + s_{1L} \tag{2.28}$$

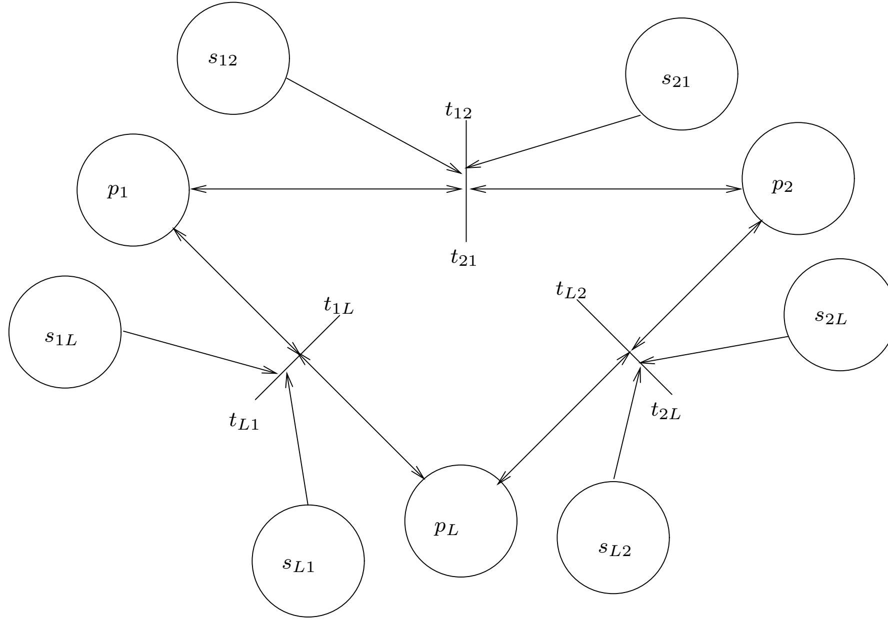


Figure 2.11: *Petri net augmented with 'Lost' place p_L with weights determined by s_{ij}*

The matrix S containing the information about network structure between two samples is constructed using entries s_{ij}:

$$S = \begin{bmatrix} 0 & s_{21} & s_{31} & \cdots & s_{n1} & s_{L1} \\ s_{12} & 0 & s_{32} & \cdots & s_{n2} & s_{L2} \\ \vdots & \vdots & \ddots & & \vdots & \vdots \\ \vdots & \vdots & & \ddots & \vdots & \vdots \\ s_{1n} & s_{2n} & s_{3n} & \cdots & 0 & s_{Ln} \\ s_{1L} & s_{2L} & s_{3L} & \cdots & s_{nL} & 0 \end{bmatrix} \tag{2.29}$$

By the use of the constructed matrix S, the number of gained tokens $x^+(p_1)$ and the number of lost tokens $x^-(p_1)$ can be expressed as:

$$x^+(p_1) = [1\,1\,\cdots\,1] \times S^T \times [1\,0\,\cdots\,0]^T \tag{2.30}$$

$$x^-(p_1) = [1\,1\,\cdots\,1] \times S \times [1\,0\,\cdots\,0]^T \tag{2.31}$$

The relation between the old marking $x(p_1)$ of p_1 and the new marking $x'(p_1)$ of p_1 can be expressed as:

$$x'(p_1) = x(p_1) + x^+(p_1) - x^-(p_1) \Leftrightarrow \tag{2.32}$$

$$x'(p_1) = x(p_1) + [1\,1\,\cdots\,1] \times (S^T - S) \times [1\,0\,\cdots\,0]^T \tag{2.33}$$

For the entire petri net, the relation between the old and the new marking can be expressed as:

$$x' = x + [1\,1\,\cdots\,1] \times (S^T - S) \tag{2.34}$$

Equation (2.34) generally results in a problem with n^2 unknown, namely the entries in matrix S, but only n equations. This indicates that the problem is not generally possible to solve. Even if it is possible to find a probable solution to the problem, the entries in the matrix S only indicates how many assets that have been moved from one location to another and not which assets that have been moved. Since this information would also be of interest, for instance in determining if a particular asset have been copied, this model approach have been discarded. Because of this, it is examined if the automata framework is more appropriate for the problem at hand.

Automata Model of Distribution System

In order to keep the information about which assets are present at a given warehouse, and not only how many, an approach where each asset is modelled separately has been considered. In this approach, an automaton is constructed for each asset in the system. This has the benefit that the location of each asset can be determined from the state of the particular automaton modelling the asset, and that a common model can be used for the assets. Figure 2.12 illustrates the approach described above. The

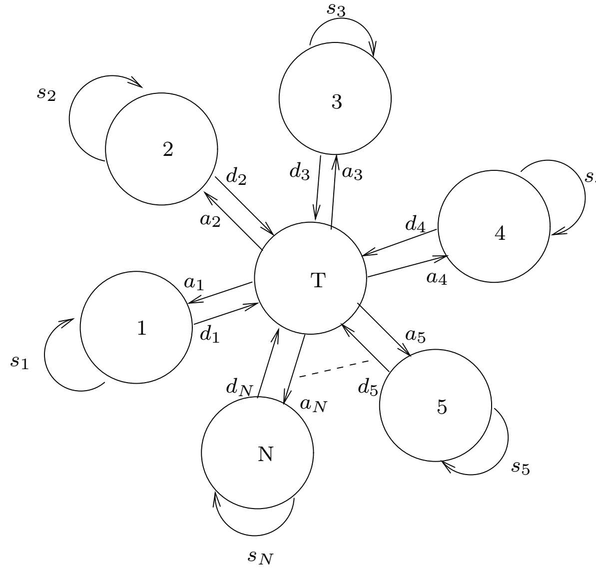


Figure 2.12: *Automaton model for a single asset*

figure shows the automaton model of a single asset. As it can be seen in the figure, the automaton contain the states $1 \cdots N$. These states represents the different warehouses present in the system. If the particular asset continues to stay in the same state i after an event has occurred, it means that the transition s_i has occurred. In order to get from one warehouse to another, a departure event d_i has to occur from the present state followed by an arrival event a_j from another warehouse. In other words, the automaton will generate event sequences of the following type:

$$\{\cdots a_i s_i^* d_i a_j s_j^* d_j \cdots\} \tag{2.35}$$

where * indicates an arbitrary sequence of the given event
As it can be seen from the figure a 'Transport' state T is introduced in the automaton, which models the time spent during the transportation between two locations.
The major problem in the system is that the above described events are only partially observable, that is; an asset can pass through a reading port without being registered. This problem can be handled by modelling the asset automaton output as a hidden Markov chain, since this is the behaviour a hidden Markov chain describes. Furthermore, the Viterbi algorithm is readily available to estimate the state sequence of a hidden Markov chain. The next sections will be used for introducing the concept of Markov chains, hidden Markov chains and how these apply for the system in question.

2.2 Markov Chains

This section covers properties of Markov chains, including the definition of a Markov chain and how they are related to automata. The section is based on (Jurafsky and Martin, 2008).
A finite state automaton (FSA) having state space $X = \{x_1, \cdots, x_n\}$ and event set $E = \{e_1, \cdots, e_t\}$ and associated transitions $f(x, e)$ for $x \in X$ and $e \in E$, is called a weighted FSA if each transition $f(x, e)$ has a probability (weight) ϕ_{ij} associated with it. The weights on the transitions indicates how likely the transition in question is, or alternately how likely the transition from state x_i to state x_j is.

The laws of probabilities require the sum of the weights on the transitions from a given state being equal to 1:

$$\sum_{j=1}^{n} \phi_{ij} = 1, \qquad \forall i \tag{2.36}$$

An example of a state sequence generated by a weighted FSA can be seen in Figure 2.13, where the state is shown at times $k-1$, k and $k+1$. The state sequence generated by a weighted FSA is called

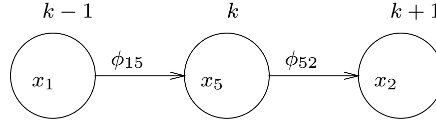


Figure 2.13: *Sample trajectory for a weighted FSA*

a first order Markov chain if the probability for the next state in the sequence is only dependent on the current state and not on the entire state history (Cassandras and Lafortune, 1999), that is, for a given state sequence $Q = \{q_1, \cdots, q_k\}$:

$$P[q_{k+1} = x_{k+1} | q_k = x_k, \cdots, q_1 = x_1] = P[q_{k+1} = x_{k+1} | q_k = x_k] \tag{2.37}$$

Furthermore, to be able to model a weighted FSA by means of Markov chains, the state sequence of the FSA has to be uniquely determined by the input sequence, that is; the FSA has to be deterministic (Jurafsky and Martin, 2008).
A Markov chain M_c can be specified by:

$$M_c = (X, \Phi, x_0, x_F) \tag{2.38}$$

Where:
X is the set of n states $\{x_1, \cdots, x_n\}$
Φ is a transition probability matrix with ϕ_{ij} as entries
x_0 is the initial state
x_F is the final state

Alternatively, for Markov chains which does not rely on specific start and end states, they can be expressed with a probability distribution over the initial states and a set of legal accepting states:

$$M_c = (X, \Phi, \boldsymbol{\pi}, X_a) \tag{2.39}$$

Where:
$\boldsymbol{\pi}=\{\pi_1, \cdots, \pi_n\}$ is the initial probability distribution of states, π_i is the probability that the chain starts in state x_i; $\sum_{i=1}^{n} \pi_i = 1$
$X_a \subseteq X$ is the set of legal accepting states

2.2.1 Hidden Markov Chains

Markov chains can be used to calculate the probability for a state sequence for a given process. In many cases however, it is not possible to observe the state sequence of a Markovian process directly. Instead a sequence of observations are made, where each observation is stochastically related to the process state at the time of the observation. In this case, the Markov chain is called hidden, since it cannot be observed directly. Instead each of the possible observations v_i in the observation set V is assigned a probability ω_{ij} of being emitted from state x_j. This is illustrated in Figure 2.14, where the Markov chain itself is illustrated above the dotted line, while the observations are illustrated below the dotted line. A hidden Markov chain M_{ch} is thus given by:

$$M_{ch} = (X, \Phi, \boldsymbol{\pi}, X_a, V, \Omega) \tag{2.40}$$

Where:
$V = \{v_1, \cdots, v_m\}$ is the set of possible observations
Ω is the observation probability matrix with ω_{ij} as entries

Furthermore, it is a requirement that the probability of the current observation is only dependent of the state emitting that observation, and not any other states or observations (Jurafsky and Martin, 2008). This means that for a state sequence $Q = \{q_1, \cdots, q_k\}$ and observation sequence $O = \{o_1, \cdots, o_k\}$:

$$P[o_i | q_1, \cdots, q_i, \cdots, q_k, o_1, \cdots, o_i, \cdots, o_k] = P[o_i | q_i] \tag{2.41}$$

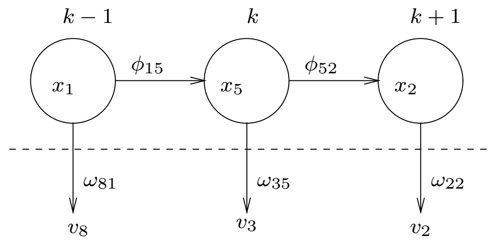


Figure 2.14: *Sample trajectory for a weighted FSA with hidden states*

2.2.2 Markov Model of Distribution System

As described in section 2.1.3 it has been chosen to model the distribution of the assets as an individual automaton for each of the assets. This is done since it is considered to be the best way to keep all the information available in the system. This section describes how the automaton for the asset relate to hidden Markov chains. First the automaton will be fitted to work as a hidden Markov process. Furthermore, new parameters needed for the hidden Markov process will be described.

Model of Asset as Hidden Markov Process

The preliminary approach for modelling the assets as individual automata is described in section 2.1.3, the graph of the automaton is repeated here in Figure 2.15. As it can be seen, the assets has a

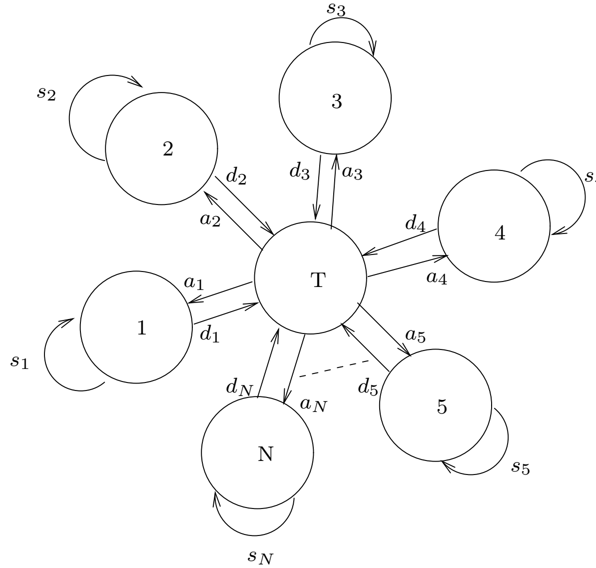


Figure 2.15: *Preliminary automaton model for a single asset*

transport state in the centre and all the other states, which corresponds to locations, are connected to this in a star like configuration.
If the automaton of an asset is to describe a first order Markov process, it is a necessity that the transition to the state at time $k+1$ is only dependant on the state at time k, and not the entire state history, that is, for a given state sequence $Q = \{q_1, \cdots, q_k\}$:

$$P[q_{k+1} = x_{k+1} | q_k = x_k, \cdots, q_1 = x_1] = P[q_{k+1} = x_{k+1} | q_k = x_k] \tag{2.42}$$

When examining the automaton graph in Figure 2.15 one can realise, that the transition away from the transport state is dependant on which state has lead into the transport state, since the transportation of an asset from one location back to the same location does not occur. Therefor, the probability related to the transition to the state at time $k+1$ is actually given by:

$$P[q_{k+1} = x_{k+1} | q_k = x_k, \cdots, q_1 = x_1] = P[q_{k+1} = x_{k+1} | q_k = x_k, q_{k-1} = x_{k-1}] \tag{2.43}$$

The expression in Equation (2.43) describes a second order Markov process, for which the mentioned Viterbi algorithm associated with hidden Markov processes does not directly apply. One solution to this problem could be to change the algorithms for use with the second order Markov process. The other approach, which will be used here, is to alter the automaton model of the system, such that it models a first order Markov process.
Figure 2.16 shows the altered automaton model for a single asset, with three locations. As it can be seen from the figure, transport states associated with each of the locations has been introduced into the model. This way it is implicit in the model, which state has led to the transport state, and the transportation of an asset away from any given location, can no longer lead back to the state representing the same location. This way the state sequences in the automaton can be considered a first order Markov chain.

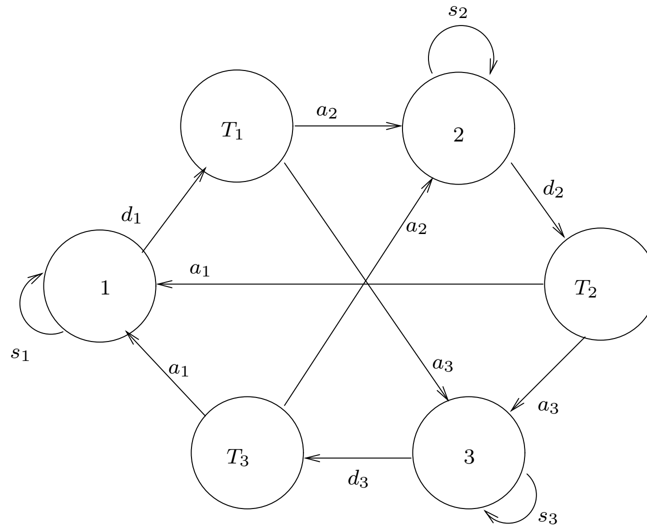


Figure 2.16: *Final automaton model for a single asset*

Synchronisation of Asset Automata: In order to be able to determine if an automaton for a specific asset has made a transition to a different location, but missed to produce an output, the automatons will be synchronised, such that every event going on in the entire system will make the time index of every single asset automaton increase. By doing this, the time index of an automaton will increase even if the transition of the particular asset has failed to produce an output. It is assumed that assets are never transported on their own, but always accompanied by at least one other asset. Thus, if one of the assets are failed to be read by the gate at a certain location, it is highly probable that another asset will be detected. This means that even if the asset is not detected at the location, the automaton representing that particular asset will still make a transition based on other events in the system. The transition will then either be a self loop to the same state which is highly likely to produce no output, or else it will be a transition to another state which will be less likely to produce no output.
It is assumed that the loading or unloading of a load of assets takes about 30 minutes, therefor the update frequency of the asset automata is limited to the reciprocal of this.

Output of Asset Automata: In the automaton model of each asset, the probability of a particular output is not related to certain states, but is instead related to the transitions between different states in the automaton. This can be realised by examining the probability of emitting a given observation from one of the states representing a location in the system.
If the automaton has ended up in the state by taking the self loop to the state, the probability of emitting a symbol from that state is very low, since it probably indicates that another asset in the system has passed a gate, thus increasing the time index of the system. In very few cases it will be an unintended reading of the particular asset, which has caused the asset to take the self loop. If

the asset has entered the location state from the transport state instead, the probability of emitting a reading will be very high, since it means that the asset has passed a gate at the location, which will have a high probability of emitting a reading and increase the time index of the system. Thus different probabilities of emitting a symbol is associated with the same state. This situation can be avoided by making the observation probabilities rely on the transitions in the automaton instead of the particular states. This means that the parameters for the hidden Markov model of the system has to be redefined compared to what was presented previously. In particular, the observation probability matrix Ω has to be redefined in order to capture this.

Hidden Markov Model of Asset

As described previously a hidden Markov chain M_{ch} can be described by the following six tuple:

$$M_{ch} = (X, \Phi, \boldsymbol{\pi}, X_a, V, \Omega) \tag{2.44}$$

Where:
Ω is the transition probability matrix where ω_{ij} is the probability that the symbol v_i is emitted from state x_j

The transition probability matrix Ω has to be changed in order to capture the nature of the system, since observations in the system are associated with transitions and not states as in the original definition. This means that instead of being a two dimensional matrix, Ω is changed to be a three dimensional matrix with entries ω_{hij} which are defined as the probability of the symbol v_h being emitted when the automaton makes a transition from state x_i to x_j, or in a more formal definition:

$$\omega_{hij} = P[v_h | x_i \rightarrow x_j] \tag{2.45}$$

Where:
The notation $x_i \rightarrow x_j$ indicates a transition from state x_i to x_j

Furthermore, a start observation probability matrix Θ is defined with entries θ_{hj} defined as the probability of the symbol v_h being emitted when x_j is the initial state, or formally:

$$\theta_{hj} = P[v_h | \text{start} \rightarrow x_j] \tag{2.46}$$

The final hidden Markov model of the asset automaton becomes the seven tuple:

$$M_{ch} = (X, \Phi, \boldsymbol{\pi}, X_a, V, \Omega, \Theta) \tag{2.47}$$

With the structure of the asset automaton established, it is necessary to determine the parameters to use in the model. For hidden Markov models the forward-backward algorithm can be used in parameter estimation of the model if the structure of the model is known, therefor this algorithm will be described in the next section. Furthermore, the forward algorithm will be described since this is used implicit in the forward-backward algorithm. These are two of the characteristic algorithms in the use of hidden Markov chains as a modelling framework (Jurafsky and Martin, 2008):

- *Computing Likelihood (The Forward Algorithm)*: Given a hidden Markov chain $M_{ch} = (X, \Phi, \boldsymbol{\pi}, X_a, V, \Omega, \Theta)$ and an observation sequence $O = \{o_1, \cdots, o_k\}$, determine the likelihood $P[O|M_{ch}]$.

- *Learning (The Forward-Backward Algorithm)*: Given an observation sequence O and the set of states $X = \{x_1, \cdots, x_n\}$, learn the Φ and Ω parameters of the hidden Markov chain M_{ch}.

The following sections will be used to describe the above mentioned forward and forward-backward algorithms in detail. The algorithms are adaptations of those described by Jurafsky and Martin (2008), changed so they fit to a hidden Markov model where the observations are related to the transitions.

2.2.3 Forward Algorithm

As mentioned, the forward algorithm is used to determine the likelihood of a given observation sequence in a hidden Markov chain. For any particular state sequence $Q = \{q_1, \cdots, q_k\}$ and observation sequence $O = \{o_1, \cdots, o_k\}$, the likelihood of the observation sequence is given by (Jurafsky and Martin, 2008):

$$P[O|Q] = \prod_{i=1}^{k} P[o_i|q_{i-1} \to q_i] \tag{2.48}$$

Since the underlying state sequence Q is unknown it is necessary to compute the probability for the observation sequence O by summing over all possible state sequences (Jurafsky and Martin, 2008). For any particular state sequence Q, the joint probability of generating an observation sequence O is given by:

$$P[O,Q] = P[O|Q] \cdot P[Q] = \prod_{i=1}^{k} P[o_i|q_{i-1} \to q_i] \cdot \prod_{i=1}^{k} P[q_i|q_{i-1}] \tag{2.49}$$

The probability for the given observation sequence is now given by summing over all possible hidden state sequences:

$$P[O] = \sum_{Q} P[O,Q] = \sum_{Q} P[O|Q] \cdot P[Q] \tag{2.50}$$

The number of state sequences in a system composed of n states, given k observations is in general n^k. Thus the algorithm for computing the probability of a certain observation sequence will be of exponential complexity ($\mathcal{O}(n^k)$) if each of the observation probabilities are to be summed up. Instead of using this approach, the forward algorithm which has $\mathcal{O}(n^2k)$ complexity, can be used.
The forward algorithm calculates the probability of a given observation sequence, by using a graphical approach called a forward trellis. An example of the trellis is illustrated in Figure 2.17. In Figure 2.17

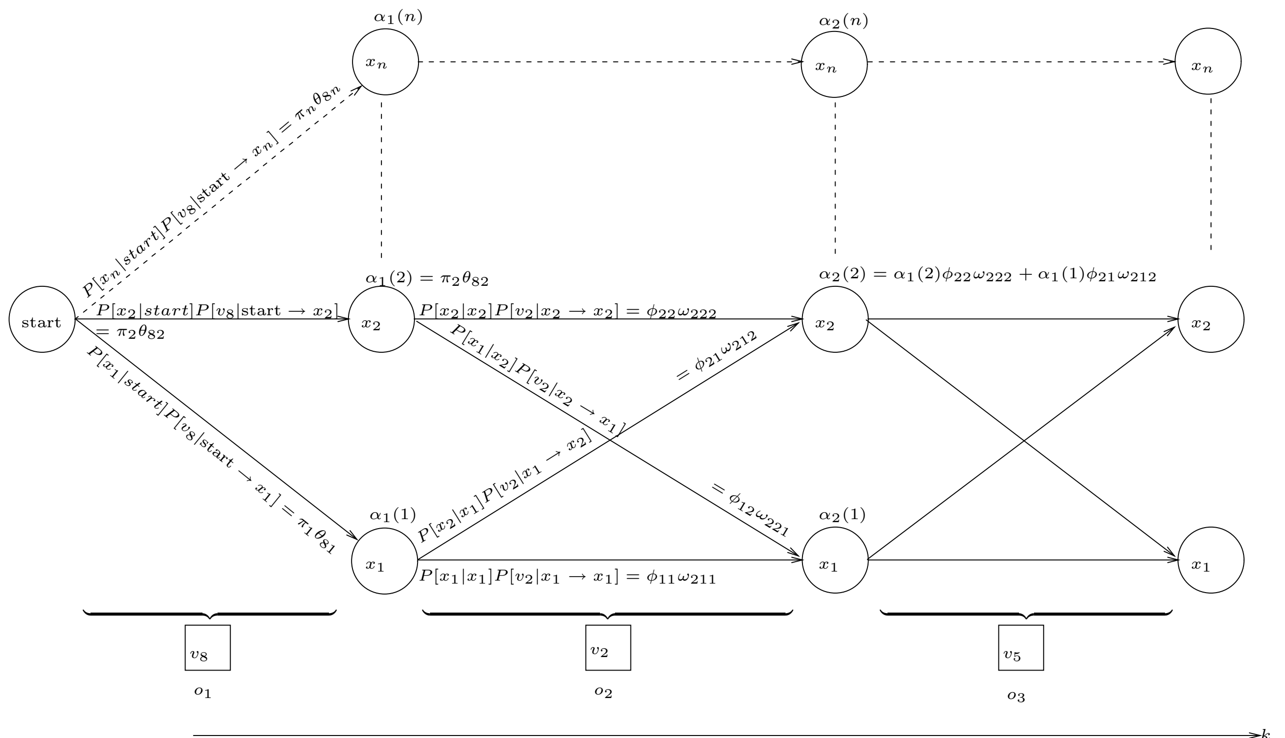


Figure 2.17: *Sample trellis for forward algorithm (Jurafsky and Martin, 2008)*

an observation sequence $O = \{o_1 = v_8, o_2 = v_2, o_3 = v_5\}$ is illustrated at the bottom. Each of the cells in the trellis is associated with a number $\alpha_k(j)$, where k is the observation number and j is the j'th state in the state set X. The number $\alpha_k(j)$ represents the probability of being in state j after seeing the first k observations, given the hidden Markov chain in question:

$$\alpha_k(j) = P[o_1, \cdots, o_k, q_k = x_j | M_{ch}] \tag{2.51}$$

Where $q_k = x_j$ corresponds to the k'th state in the sequence being x_j. For a given state x_j at time k, the number $\alpha_k(j)$ is calculated as:

$$\alpha_k(j) = \sum_{i=1}^{n} \alpha_{k-1}(i) \cdot \phi_{ij} \cdot P[o_k | x_i \to x_j] \tag{2.52}$$

If $o_k = v_h$ then $P[o_k | x_i \to x_j] = \omega_{hij}$ or θ_{hj} if x_i is the start state, as in the definition of the observation probability matrices Ω and Θ. By the definition of the elements in the trellis it is now possible to derive the forward algorithm, for K observations:

1. Initialisation:

$$\alpha_1(j) = \pi_j \cdot P[o_1 | \text{start} \to x_j], \qquad 1 \leq j \leq n \tag{2.53}$$

2. Recursion:

$$\alpha_k(j) = \sum_{i=1}^{n} \alpha_{k-1}(i) \cdot \phi_{ij} \cdot P[o_k | x_i \to x_j], \qquad 1 \leq j \leq n,\ 1 < k \leq K \tag{2.54}$$

3. Termination:

$$P[O | M_{ch}] = \alpha_K = \sum_{j=1}^{n} \alpha_K(j) \tag{2.55}$$

The forward algorithm can be used to evaluate how well a given model match a given observation sequence. This can be used to select between different models (Rabiner, 1989).

2.2.4 Forward-Backward Algorithm

The forward-backward algorithm can be used to learn the parameters Φ and Ω of a hidden Markov chain M_{ch}. For this purpose it is necessary to have a known observation sequence O produced by the process and a known state space X. The Φ and Ω parameters are estimated iteratively from an initial estimate of them, thus using this estimate, better and better estimates are obtained.
The forward-backward algorithm uses a probability related to the forward probability, which is called the backward probability $\beta_k(j)$. The backward probability $\beta_k(j)$ is the probability of seeing the observations from time $k+1$ to time K given that the state at time k is x_j:

$$\beta_k(j) = P[o_{k+1}, \cdots, o_K | q_k = x_j, M_{ch}] \tag{2.56}$$

It can be computed in a manner similar to the forward algorithm:

1. Initialisation:

$$\beta_K(i) = 1, \qquad 1 \leq i \leq n \tag{2.57}$$

2. Recursion:

$$\beta_k(i) = \sum_{j=1}^{n} \phi_{ij} \cdot P[o_{k+1} | x_i \to x_j] \cdot \beta_{k+1}(j), \qquad 1 \leq i \leq n,\ 1 \leq k < K \tag{2.58}$$

3. Termination:

$$P[O|M_{ch}] = \beta_1 = \sum_{j=1}^{n} \pi_j \cdot P[o_1|\text{start} \rightarrow x_j] \cdot \beta_1(j) \tag{2.59}$$

The probability ϕ_{ij} of taking the transition from state x_i to x_j, can be estimated by the following proposition:

$$\hat{\phi}_{ij} = \frac{\text{expected number of transitions from state } x_i \text{ to state } x_j}{\text{expected number of transitions from state } x_i} \tag{2.60}$$

In order to calculate $\hat{\phi}_{ij}$, a variable $\xi_k(i,j)$ is defined as the probability of being in state x_i at time k and in state x_j at time $k+1$, given the observation sequence and the hidden Markov chain in question:

$$\xi_k(i,j) = P[q_k = x_i, q_{k+1} = x_j|O, M_{ch}] \tag{2.61}$$

The variable $\xi_k(i,j)$ can be calculated using the following property of probabilities:

$$P[q_k = x_i, q_{k+1} = x_j|O, M_{ch}] = \frac{P[q_k = x_i, q_{k+1} = x_j, O|M_{ch}]}{P[O|M_{ch}]} \tag{2.62}$$

Where $P[q_k = x_i, q_{k+1} = x_j, O|M_{ch}]$ can be calculated from (Jurafsky and Martin, 2008):

$$P[q_k = x_i, q_{k+1} = x_j, O|M_{ch}] = \alpha_k(i) \cdot \phi_{ij} \cdot P[o_{k+1}|x_i \rightarrow x_j] \cdot \beta_{k+1}(j) \tag{2.63}$$

The probability $P[O|M_{ch}]$ can for instance be calculated as α_K, thus $\xi_k(i,j)$ is evaluated as:

$$\xi_k(i,j) = \frac{\alpha_k(i) \cdot \phi_{ij} \cdot P[o_{k+1}|x_i \rightarrow x_j] \cdot \beta_{k+1}(j)}{\alpha_K} \tag{2.64}$$

The expected number of transitions from state x_i to state x_j is then given as the sum over all k of $\xi_k(i,j)$. In order to obtain $\hat{\phi}_{ij}$, the total number of transitions out of state x_i is found by summing over all transitions out of state x_i. The final expression for $\hat{\phi}_{ij}$ is given by:

$$\hat{\phi}_{ij} = \frac{\sum_{k=1}^{K-1} \xi_k(i,j)}{\sum_{k=1}^{K-1} \sum_{j=1}^{n} \xi_k(i,j)} \tag{2.65}$$

Just like the state transition probability matrix Φ, the observation probability matrix Ω can be estimated by making an estimate $\hat{\omega}_{hij}$ of the entry ω_{hij} the following way:

$$\hat{\omega}_{hij} = \frac{\text{expected number of transitions from state } x_i \text{ to state } x_j \text{ and observing symbol } v_h}{\text{expected number of transitions from state } x_i \text{ to state } x_j} \tag{2.66}$$

The denominator of Equation (2.66) has already been calculated in the estimate of the ϕ_{ij} parameter, and the numerator can be found by simply summing $\xi_k(i,j)$ over the time indices where the observation was the symbol v_h in question:

$$\sum_{k=1, o_{k+1}=v_h}^{K-1} \xi_k(i,j) \tag{2.67}$$

The expression in Equation (2.67) means sum of $\xi_k(i,j)$ over all k for which the observation was the symbol v_h. The expression for the estimate of ω_{hij} thus become:

$$\hat{\omega}_{hij} = \frac{\sum_{k=1, o_{k+1}=v_h}^{K-1} \xi_k(i,j)}{\sum_{k=1}^{K-1} \xi_k(i,j)} \tag{2.68}$$

The forward-backward algorithm is meant to be run iteratively until the estimates of the Φ and Ω matrices converges. In practice the initial estimates of the matrices are very important for the algorithm to work properly. In order to obtain good initial estimates of the system parameters to use either directly in the model or together with the forward-backward algorithm, the next section will describe an analysis of the derivation of these.

2.2.5 Configuration of Initial System Parameters

This section deals with the initial configuration of the parameters used in the system. These parameters are, as described in section 2.2.2; the transition probability matrix Φ, the observation probability matrix Ω, the initial observation probability matrix Θ, the vocabulary V and the initial state probability distribution $\boldsymbol{\pi}$. The parameters chosen in this section can be considered as the best guess on the actual system parameters, and will be used together with the forward-backward algorithm described in section 2.2.4 to derive a better estimate of the actual parameters of the system.

Transition Probability Matrix

As it can be seen in the illustration of the asset automaton in Figure 2.16 on page 26, there exists four different types of transitions in the system. These are; self loop in a location state, self loop in a transport state, a transition from a location state to its corresponding transport state and finally a transition from a transport state to a location state different from its corresponding location state.
The automaton in question models the distribution of a cc container in the system. Since the locations in this system are considered equally likely for the assets to end up in, all of the transitions of the same type are considered to have the same probability. This means that if an asset is in location or state 'a' it is equally likely to take the self loop as if it had been in location or state 'b'. These considerations also apply to the other three types of transitions.
Correlating the above mentioned considerations with the illustration of the asset automaton in Figure 2.16 it can be realised that only transition probabilities for two types of transitions will have to be estimated. The rest of the transition probabilities can be calculated from these. The two probabilities to be estimated are the probabilities of taking self loops when in either a location state or in a transport state.
For the location state, there are only two possible transitions and since the total probability is 1, the remaining probability can be calculated by subtracting the probability for a self loop from 1. The transport state has more transitions associated with it, but since all the transitions to location states are considered equally likely, these can be calculated by subtracting the probability for the self loop from 1 and then dividing by the number of possible transitions to location states:

$$P[T_i \rightarrow x_j] = \frac{1 - B}{n - 1} \tag{2.69}$$

Where:
T_i is the transport state in question
x_j is a location state not associated with the transport state
B is the probability of a self loop in the transport state
n is the number of locations in the system

The number of possible transitions to location states are one less than the total number of location states since it is not possible to take a transition from a transport state back to its associated location state.
The following paragraph describes how the two transition probabilities are derived.

Derivation of Transition Probabilities In order to derive an estimate of the self loop transition probabilities some intermediate variables are defined, these are; N which is the number of assets in the system, M which is the average number of assets in each transport, F which is the average frequency at which the time index of the system is updated, which is done when measurements are separated at least 30 minutes, and f which is the transport frequency for each asset or the average frequency at which each individual asset is transported.
The variable which is of interest when seeking the transitional probabilities, is the transport frequency for the individual assets, since the sum of time periods each asset stays at a given location state and its associated transport state is given as the reciprocal of the transport frequency. If the average time an asset spends at a location state (T_l) and the average time an asset spends at the locations associated

transport state (T_t) is known, the transitional probability A for the self loop in the location state can be estimated from the following equation:

$$T_l^F = \sum_{j=0}^{\infty} j \cdot A^{j-1} \cdot (1-A) \Rightarrow \tag{2.70}$$

$$T_l^F = \frac{A}{1-A} \Rightarrow \tag{2.71}$$

$$A = \frac{T_l^F}{T_l^F + 1} \tag{2.72}$$

Where T_l^F is T_l normed regarding the average transport frequency F, since the asset automatons time base is synchronised to the transport frequency. Likewise, the transition probability B for the self loop in the transport state can be calculated using the following:

$$T_t^F = \sum_{j=0}^{\infty} j \cdot B^{j-1} \cdot (1-B) \Rightarrow \tag{2.73}$$

$$T_t^F = \frac{B}{1-B} \Rightarrow \tag{2.74}$$

$$B = \frac{T_t^F}{T_t^F + 1} \tag{2.75}$$

In order to express the time periods T_l and T_t in terms of the transport frequency F an estimate of the transport frequency is needed. The transport frequency is calculated using the following equations:

$$F = \frac{N \cdot f}{M} \Rightarrow \tag{2.76}$$

$$F = \frac{N \cdot \frac{1}{T_l + T_t}}{M} \Rightarrow \tag{2.77}$$

$$F = \frac{N}{M \cdot (T_l + T_t)} \tag{2.78}$$

The time periods T_l and T_t can then be expressed in terms of the frequency F, by the following expression:

$$T_l^F = F \cdot T_l \tag{2.79}$$

$$T_t^F = F \cdot T_t \tag{2.80}$$

This means that in order to derive the transitional probabilities A and B, an estimate of the time periods T_l and T_t, the total number N of assets in the system and the average number M of assets per transport is needed. Given these estimates the transport frequency F is calculated using Equation (2.78), then the time periods T_l and T_t is expressed in terms of F using Equations (2.79) and (2.80) and lastly, A and B are calculated using Equations (2.72) and (2.75).

Observation Probability Matrix

The observational probabilities are tied to the probability of an asset being read when passing through an RFID reader gate. When the asset passes the gate, one of two things can happen; either the asset is read in which case the output of the asset automaton will be the id of the gate in question. The other possibility is that the reading of the asset is missed and the corresponding output of the asset automaton will be the empty symbol. Out of the four possible transitions described previously, two of them involves the asset passing through a gate, and thus for these cases the possibility of emitting the symbol associated with the location is equal to the possibility of the asset being read by the gate. The two transitions in question is the transition from a transport state to a location state, and the transition from a location state to its corresponding transport state.
When an asset is in a transport state and takes a self loop, it is not possible to generate other than the empty symbol, since no RFID reader gates are involved in this situation, thus the empty symbol will always be generated in this case.

When an asset takes a self loop in a location state it is possible to emit the location symbol. This happens if the asset is in the vicinity of the RFID reader gate at the location and unintentionally gets read without being prepared for transport. This is assumed to happen with a low probability, since it requires the asset to be close to the loading ramp during its stay at the particular location and because the probability for the asset staying at a particular location is assumed to be high. If the asset does not emit the location symbol when taking the self loop it will emit the empty symbol, which happens with a high probability.

Initial State Probability Distribution

When an asset is introduced to the system, each of the different location states are considered equally likely as initial state. The transport states are considered impossible as initial states.

Initial Observation Probability Matrix

The probability of emitting a symbol corresponding to the initial location is the same as the probability of emitting a symbol when taking a normal transition into the particular location state, since it also involves the RFID reader gate. Therefor, the parameters for emitting the location symbol respectively the empty symbol are the same.

Vocabulary

The RFID reader gates at the different locations involved in the system are assigned an id. This id is mapped to an id for the location the gate belongs to. The location ids will be used in the vocabulary as the symbol associated with the location. Furthermore, the empty symbol is added to the vocabulary.

2.3 Validation of Simulation Model

With the derivations made in the previous sections it is possible to implement a hidden Markov model of the assets in Matlab. The model can then either make use of the initial parameters directly or it can make use of the forward-backward algorithm to estimate the true system parameters if there are output sequences available from the actual system.
In order to validate the parameters of the model, it is necessary to have output sequences from the actual system available. Since the registration of cc containers via RFID tags has not been implemented into the distribution system yet, data from the actual system is not available to the project group. Therefor, data from a similar system which already has been implemented will be used. The data to validate the model of the asset output against come from the Post Danmark registration of letter cages, which are transported between post offices and package distribution centres in Denmark. This system is not in the same scale as the distribution of cc containers in Europe. More specifically, the data received from the Post Danmark setup contains registrations from 22257 assets collected from eleven different locations.

A test has been conducted in order to validate the hidden Markov model of the system. The model is a Matlab script implementation of the model derived in this chapter. The model is a pure hidden Markov model, which means that each assets state- and output sequences are simulated independently. This has the effect that the output from the model does not capture the fact that assets are expected to be transported at least two together at all times. On the other hand the model is expected to be less time consuming than a full system model, and can thus be used to test different system parameters within a reasonable time frame. Later, when a set of parameters, which can be validated using the pure hidden Markov model, have been found, these will be validated with a model describing the full dynamics of the system. A complete description of the test can be found in appendix B, the main results and conclusions from the test will be summarised in this section.

In order to validate if the hidden Markov model of the system uses the correct parameters the test has been carried out using the Viterbi algorithm (described in chapter 4). More specifically, the γ-values produced by the algorithm at the last time index has been used. The maximal γ-value at the last time index is proportional to the probability of the output sequence used as input to the algorithm, given the optimal state sequence and the underlying hidden Markov model. It is expected that the distribution of these γ-values will be similar if the model matches the actual system. The distributions of the γ-values are compared using the rank sum test, which is described in appendix A.

First the simulation model is tested using the parameters calculated based on the Post Danmark data directly. The distribution of the γ-values calculated using the actual system output and the simulation model output can be found in Figure 2.18 and Figure 2.19 respectively. As it can be seen in the figures, the distributions does not appear to be similar, and since the p-value from the rank-sum test is zero, the hypothesis of the distributions being similar can readily be rejected.
Next the simulation model is tested using parameters estimated by the forward-backward algorithm,

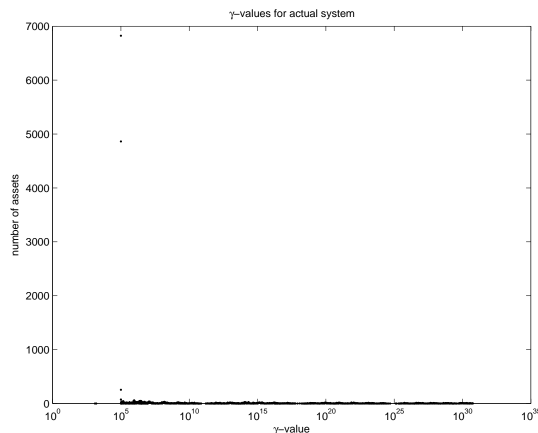


Figure 2.18: *Distribution of γ-values of Post Danmark data ($A = 0.8333$,$B = A$)*

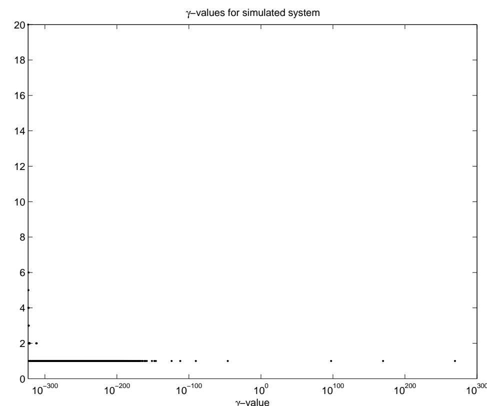


Figure 2.19: *Distribution of γ-values of simulated data ($A = 0.8333$,$B = A$)*

with the previous parameters as initial parameters. The distribution of the γ-values calculated using the actual system output and the simulation model output can be found in Figure 2.20 and Figure 2.21 respectively. As it can be seen in the figures, the distributions does not appear to be similar, and since the p-value from the rank-sum test is zero, the hypothesis of the distributions being similar can readily be rejected.
Finally the simulation model is tested using parameters tuned by hand. The distribution of the γ-values calculated using the actual system output and the simulation model output can be found in Figure 2.22 and Figure 2.23 respectively. As it can be seen in the figures, the distributions does not appear to be similar, but since the p-value from the rank-sum test is 0.54, the hypothesis of the distributions being similar cannot be rejected. Although the rank-sum test cannot reject the model parameters in the final test of the simulation model, the distributions of γ-values in Figure 2.22 and 2.23 does not appear to be similar. Based on the results from the different test it will though still be assessed that the latter parameters are those that fit the actual system best, and thus these are accepted as system parameters.
This concludes the modelling of the distribution system as a pure hidden Markov model. The next chapter deals with a SimEvent model of the system, which in contrast to the pure hidden Markov model also takes into account that assets are always transported at least two at a time.

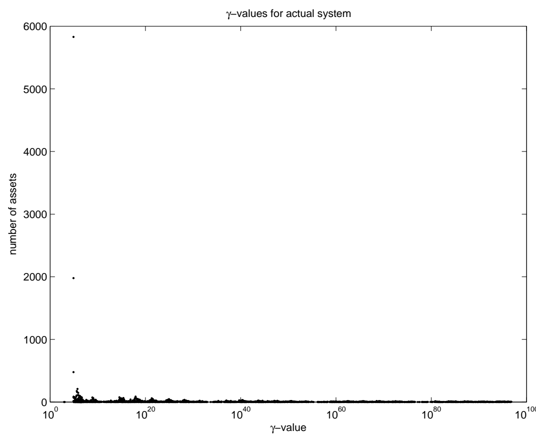


Figure 2.20: *Distribution of γ-values of Post Danmark data ($A = 0.76$,$B = 0.85$)*

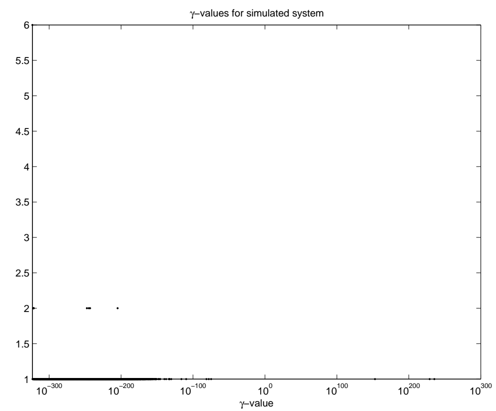


Figure 2.21: *Distribution of γ-values of simulated data ($A = 0.76$,$B = 0.85$)*

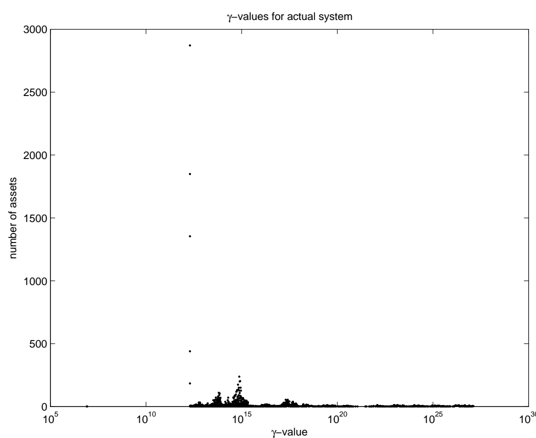


Figure 2.22: *Distribution of γ-values of Post Danmark data ($A = N/(N+1)$,$B = A$), where n is the number of assets in the system*

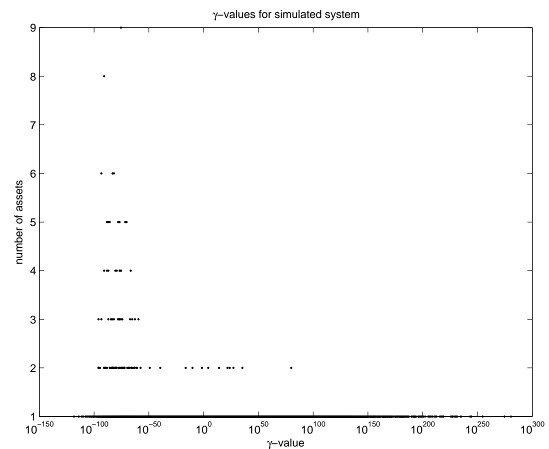


Figure 2.23: *Distribution of γ-values of simulated data ($A = N/(N+1)$,$B = A$), where n is the number of assets in the system*

Chapter 3

Simulink Model of System

This section describes the construction and properties of the simulation model of the distribution system. The simulation model is constructed in Matlab Simulink, using the SimEvents Toolbox version 2.1.

The SimEvents Toolbox, is an extension for the Matlab Simulink simulation environment, which adds features for simulation of DES. SimEvents uses events and entities to model a DES. An entity is a discrete item, which can be used to model items, such as network packets etc. The entities are routed through a network of different elements, such as queues, servers, gates and switches during the simulation. Entities can carry data, known in SimEvents as attributes. (The MathWorks, 2007) The size of the attribute, can be either a single character or a multidimensional array. The attributes does not use dynamic allocation of memory i.e the data size of the attribute is fixed at the initial size.

The SimEvents library works, in the same way as the ordinary Simulink library, and can interact with Simulink blocks. Discrete events can be used to trigger different actions in the SimEvents simulation. These events can be generated, based on entities departing from a block. Events can also be time based, either at a fixed interval or scheduled. SimEvents can also interact with Stateflow, which can generate events based on state machines. (The MathWorks, 2007) As the reader is presumed to have knowledge of Simulink, only the used blocks will be briefly described in the following sections.

3.1 Simulink Model

This section describe the constructed simulation model, and the relation to the physical plant, which is modelled. The plant is a combination of a distribution system of assets and a number of automated registration devices.

The plant operates without any input, but produces a number of outputs. However these output are error prone. The output of the system is a number of readings, produced when an asset either enter or leaves a location, thus being scanned in the RFID reader. This reading includes the following elements:

- Identification number of the cc container
- Time stamp of registration
- Location identification of where the registration occurred
- Other miscellaneous information, such as method of scanning etc.

However various errors occur in the system as described in the introduction in chapter 1.

1. The cc's pass through the reading portal, without being registered.

2. The cc's are registered without having passed through the reading portal.

The model consist of three major blocks. An initialisation block, a places block and a transport block. Furthermore, an observation generation block exists in the system. The observation generation block generates the observation sequence from the state sequence of each asset. The block diagram of the simulation model is shown in Figure 3.1. Assets are fed into the system, by the Initialise block. The assets are then fed into the locations block, which models the locations in the system. When an asset leaves the location block or makes a self loop in a location, the asset is routed through the observation block, which updates the observation sequence of the asset. When an assets makes self loops in the transport block no observations are generated. Two different probabilities are used, when the observation sequence is updated; the scan probability, which is the probability of a successful reading and a faulty scan probability, which is the probability of the RFID reader to output a reading, even when the asset has not passed through the reader. The state and observation sequences are stored into two of the attributes of each asset.

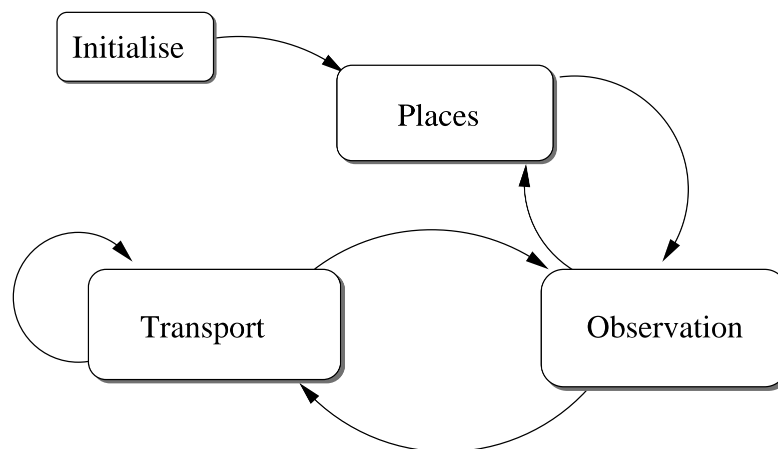


Figure 3.1: *Block diagram of simulation model divided into three major blocks, and an output block, which generates the observation sequences.*

The simulation model is, as stated above, divided into three blocks, the functionality of these blocks will be described in the following. The complete Simulink model is illustrated in Figure 3.2. The

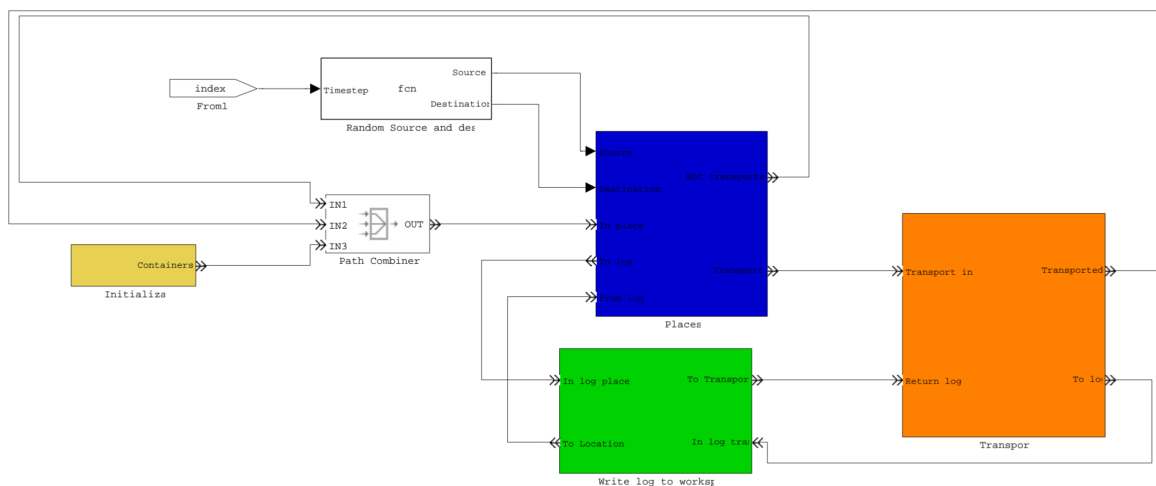


Figure 3.2: *The complete simulink model, the major blocks are colored orange, blue and yellow*

Simulink model uses entities to represent assets. Each asset has a number of attributes associated with it, these attributes include:

1. ID

2. Current location (state) L
3. Time stamp of last event T
4. State history (state sequence) S
5. Event history (observation sequence) E
6. A probability (self loop in location)
7. B probability (self loop in transport)
8. Scan probability
9. Faulty scan probability

And some additional attributes, due to the nature of the SimEvent toolbox. These additional attributes are used to control the flow of assets through the system.

The ID is the unique identification number of each asset. The current location is the state of the FSA in figure 2.16 on page 26 for the asset. The time stamp is the time step of the current event. The state history attribute is a matrix of past locations combined with corresponding time stamps. The observation sequence is constructed, by the output function, which uses the state sequence to generate the observation sequence, as illustrated in Equation 3.1, where *output* is the observation generation block . The A probability is the probability, at which an asset makes a self loop in a location state, and the B probability is the probability at which the asset makes a self loop in a transport state. Scan probability is the probability for a successful reading of the asset passing through a reading portal. The faulty scan probability is the probability that an asset generates a reading without being transported.

$$S = \begin{bmatrix} L(1) & T(1) \\ L(2) & T(2) \\ \vdots & \vdots \\ L(k) & T(k) \end{bmatrix} \xrightarrow{output} E = \begin{bmatrix} O(1) & T(1) \\ O(2) & T(2) \\ \vdots & \vdots \\ O(k) & T(k) \end{bmatrix} \tag{3.1}$$

Where:
$O(k)$, is the observation at time k, generated by the observation block

3.1.1 Model Description

An activity diagram for the Simulink model can be seen in Figure 3.3. The diagram illustrates how the model operates. The initial model parameters, which is the number of assets, number of locations and transition and output probabilities are loaded into the model workspace. When the simulation starts, entities are continuously generated in the Init block. When the number of entities generated is less than or equal to the number of assets specified for the simulation, the attributes are set. The entities are fed out of the Init block and into the system. When the desired number of assets is exceeded the following entities are discarded.

In order to have assets only to take a loop in either the transport block, the location block or to take a transition between the two blocks, a queue is put in both the transport block and the location block. Both the input and output can be blocked with gates, the input and output gates are controlled by the total number of assets in the two queues. A third queue exists in the places block, which is used to ensure that minimum two assets are transported together.

When assets exits either the places or transport blocks, the state- and observation sequences are updated.

When all assets are in one of the two queues, the time step is increased and a new source and destination for the transport are generated.

When the assets are in the places block, the order of the assets are randomized. For each asset in the places block, the current location is compared with the random generated source. All assets with a matching destination are routed out of the places block, this is done to model a transport from a specific location.

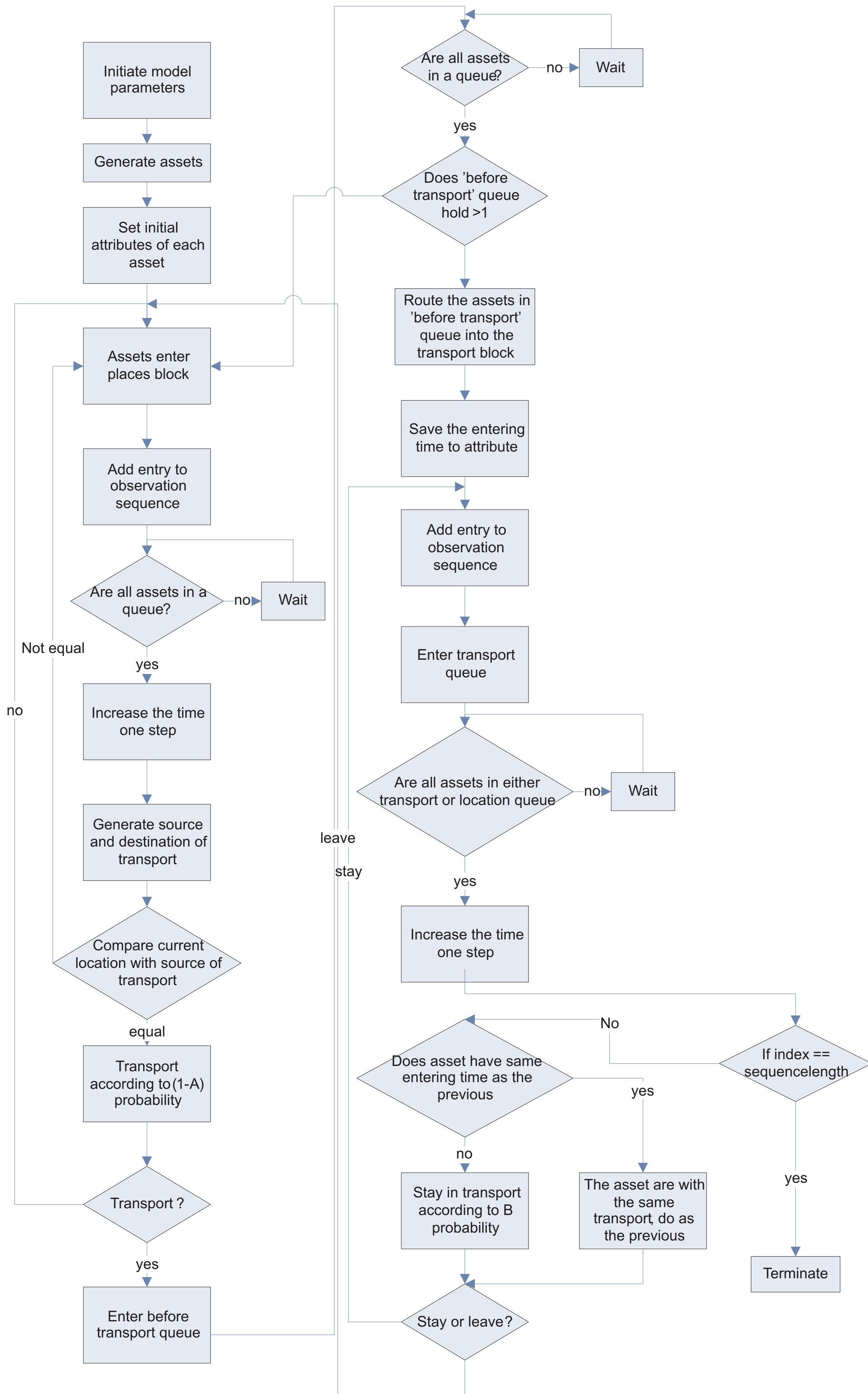


Figure 3.3: *Activity diagram for the simulation model*

3.1.2 Init Block

When the simulation is started, a specified number of entities are created in the *Init* block. The block uses the number of assets, and number of locations to initialise the system. The initial attributes are all assigned in the init block.

3.1.3 Places Block

The *Places* block is used to simulate, the different locations where the assets can be in the distribution system. It consist of a infinite server, which can hold infinitely many entities. To model the Markovian properties, the state sequence $S = \{s_0, s_1, s_2, \cdots, s_k\}$ of each asset has to have the same length, which means that each time an automaton changes state, k is increased and the state sequence for all assets has to be updated. This is achieved in the places block, by having all entities in the places block depart from the block and add a new entry to the state history attribute, which hold the state sequence S for the asset. Each time the assets are departing from the places block, the observation sequence is updated.

Whenever a transport event is occurring, the following information is generated; the location from which the transport is occurring, and the destination of the transport, which are randomly chosen, and ensured not to be coinciding.

There exists two outputs from the places block, one for the location from which a transport was chosen, and one for all other locations. This mechanism enables the system to change the current state for assets residing at a specific location, meanwhile keeping the state of all the other entities, while increasing the value of k for all entities. An output switch switches between the two outputs according to the current location attribute. When the assets enter the block, the output of the queue is blocked until all assets have returned to a queue, either in the places block or in the transport block. The index variable is increased, and a new transport is initiated as the output of the queue is unblocked.

When assets have the correct source location, they are routed to the 'before transport' queue, with the transport probability, entered as a parameter to the model. The assets are only transported when two or more assets are in the queue.

3.1.4 Transport Block

The transport block changes the location of the entering assets. An entry is added to the state sequence, each time an asset enters or loops in the transport block.

3.1.5 Observation Sequence Generation

As the history attribute of each asset, contains the actual state sequence, this should be mapped onto an observation sequence. This is done in an embedded Matlab block, which generates the following entries in the observation sequence;

When the state sequence includes two identical location entries, which indicate a self loop in a location, a zero output is generated, with a probability of $1-\omega_{failprob}$. And of a faulty reading with a probability of $\omega_{failprob}$.

$$P[O_k = 0)|q_k = q_{k-1} = x_i] = 1 - \omega_{failprob} \tag{3.2}$$

$$P[O_k = v_i)|q_k = q_{k-1} = x_i] = \omega_{failprob} \tag{3.3}$$

Where:
$\omega_{failprob}$ is the probability of a reading without the assets passes through the RFID reader

When the state sequence includes a transport state, the output is 0, with the probability $1-\omega_{scanprob}$, indicating that the asset has been transported, but not registered. The value of $\omega_{scanprob}$ and $\omega_{failprob}$.

$$P[O_k = v_i)|q_{k-1} = T_j, q_{k-2} = x_i] = \omega_{scanprob} \quad (3.4)$$
$$P[O_k = 0)|q_{k-2} = x_j, q_{k-1} = T_j] = 1 - \omega_{scanprob} \quad (3.5)$$
$$P[O_k = 0)|q_{k-1} = T_j, q_{k-2} = x_i] = 1 - \omega_{scanprob} \quad (3.6)$$
$$P[O_k = v_i)|q_{k-2} = x_j, q_{k-1} = T_j] = \omega_{scanprob} \quad (3.7)$$

Where:
$\omega_{scanprob}$ is the probability of a successful reading

3.1.6 Summary

The simulation model outputs, two arrays, one with the state sequence, and one with the corresponding observation sequence, which simulates the port readings of the distribution system.

Figures of all blocks included in the simulation model, can be found in appendix D. Now that the Simulink simulation model has been constructed the model has to be validated, this is done in the following section.

3.2 Validation of Simulink Model

This section deals with the validation of the SimEvent model used to simulate the system. The quality of the model will be evaluated by comparing its output with measurements from the actual system. The complete test report is found in appendix C.

The test is conducted similar to the validation of the hidden Markov model, described in section 2.3, with the parameters validated on the hidden Markov model.

The γ-values from the simulation model, and the test data are shown in Figure 3.4 and 3.5 respectively.

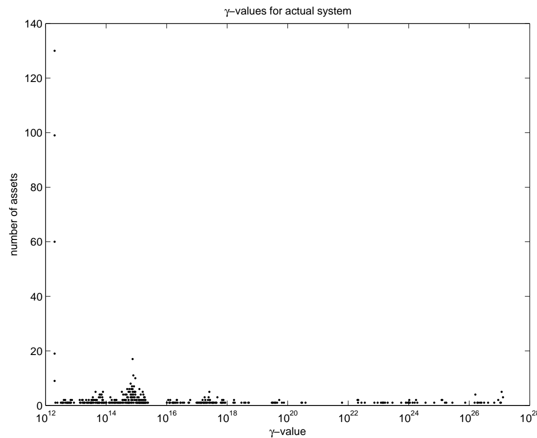


Figure 3.4: *γ-values generated by the Viterbi algorithm, run on data from Post Danmark*

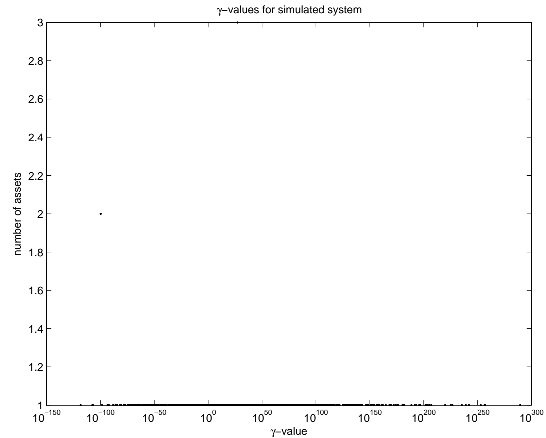


Figure 3.5: *γ-values generated by the Viterbi algorithm, run on data from the SimEvent simulation model*

The result of the rank sum test is a p-value of zero, which indicate that the H_0 hypothesis, that the underlying distributions are similar can be rejected. This can either mean that the simulation model is not applicable in modelling the system, or the test is simply too sensitive, which will mean that small modelling errors will result in rejection of the hypothesis.

Chapter 4

Estimator Design

This chapter deals with the analysis and design of the estimator to use in the system. As described in chapter 2, the dynamics of the distribution system can be modelled by a hidden Markov chain. One of the algorithms designed for use with hidden Markov chains deals with the problem of finding the best hidden state sequence given an output sequence from the hidden Markov chain. This is the Viterbi algorithm (Cassandras and Lafortune, 1999):

- *Decoding (The Viterbi Algorithm)*: Given an observation sequence O and a hidden Markov chain M_{ch}, discover the most likely hidden state sequence $Q = \{q_1, \cdots, q_k\}$.

This chapter describes how the Viterbi algorithm works and how it will be used to determine the state sequences of the assets in the system.

4.1 Viterbi Algorithm

Like the forward algorithm described in section 2.2.3, the Viterbi algorithm utilises the use of a forward trellis. But instead of calculating the probability of a given observation sequence, the Viterbi algorithm is used for calculating the most probable hidden state sequence for a given observation sequence. An example of the trellis used in the Viterbi algorithm can be seen in Figure 4.1. Each of the cells in the trellis is associated with a number $\gamma_k(j)$, which represents the probability of being in state x_j after k observations and passing through the most probable state sequence $Q = \{q_1, \cdots, q_{k-1}\}$, given the hidden Markov chain in question:

$$\gamma_k(j) = \max_{q_1, \cdots, q_{k-1}} P[q_1, \cdots, q_{k-1}, o_1, \cdots, o_k, q_k = x_j | M_{ch}] \tag{4.1}$$

This probability can be calculated recursively by using the following expression:

$$\gamma_k(j) = \max_{i=1}^{n} \gamma_{k-1}(i) \cdot \phi_{ij} \cdot P[o_k | x_i \to x_j] \tag{4.2}$$

This recursive expression is almost identical to that of the forward algorithm, with the difference that the Viterbi algorithm uses the maximum over the previous path probabilities, instead of the sum over all the previous path probabilities.
Furthermore, a backpointer $\chi_k(j)$ is used in the algorithm to keep track of the of the path of states that has lead to each state. This is used to trace back the best path to the beginning. The value of $\chi_k(j)$ is the index of the state x_i which maximises $\gamma_k(j)$:

$$\chi_k(j) = \operatorname*{argmax}_{i=1}^{n} \gamma_{k-1}(i) \cdot \phi_{ij} \cdot P[o_k | x_i \to x_j] \tag{4.3}$$

With the Viterbi probabilities $\gamma_k(j)$ and the backpointer $\chi_k(j)$ defined it is now possible to derive the Viterbi algorithm (Rabiner, 1989):

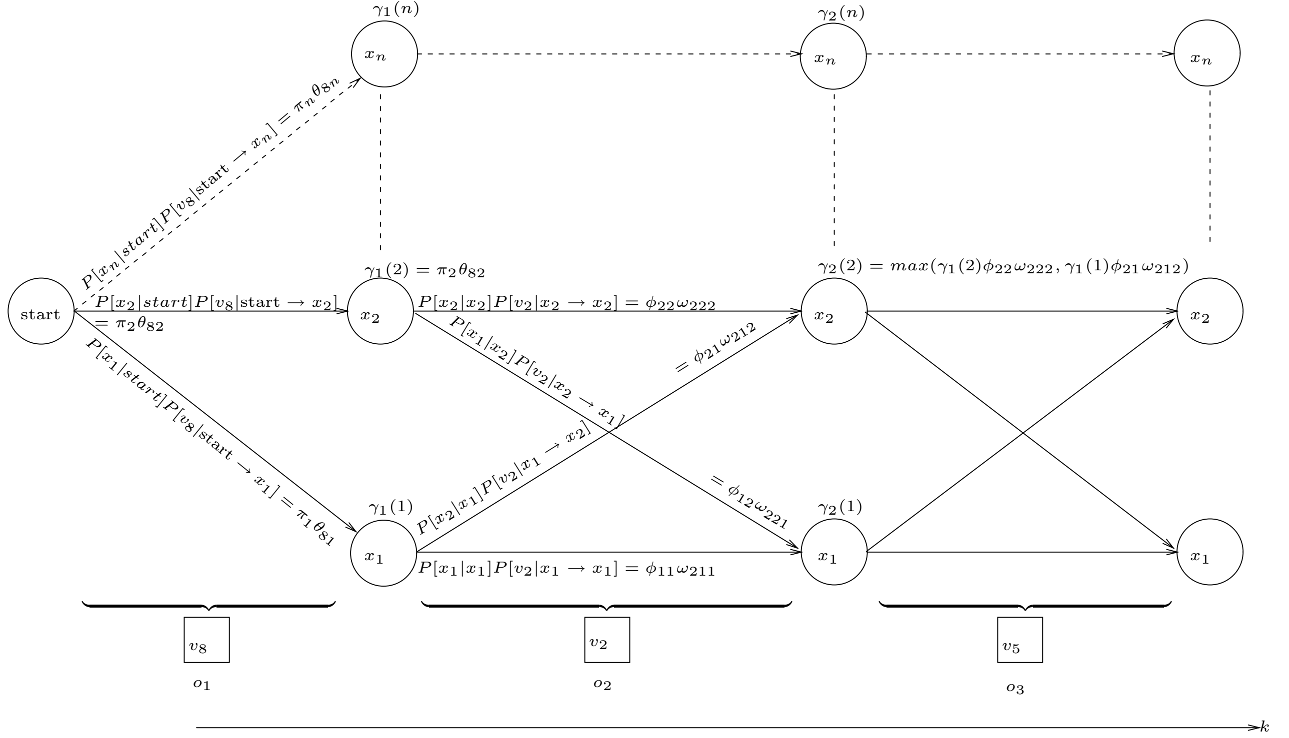


Figure 4.1: *Sample trellis for Viterbi algorithm (Jurafsky and Martin, 2008)*

1. Initialisation:

$$\gamma_1(j) = \pi_j \cdot P[o_1|\text{start} \rightarrow x_j], \qquad 1 \leq j \leq n \tag{4.4}$$

$$\chi_1(j) = 0 \tag{4.5}$$

2. Recursion:

$$\gamma_k(j) = \max_{i=1}^{n} \gamma_{k-1}(i) \cdot \phi_{ij} \cdot P[o_k|x_i \rightarrow x_j], \qquad 1 \leq j \leq n,\ 1 < k \leq K \tag{4.6}$$

$$\chi_k(j) = \operatorname{argmax}_{i=1}^{n} \gamma_{k-1}(i) \cdot \phi_{ij} \cdot P[o_k|x_i \rightarrow x_j], \qquad 1 \leq j \leq n,\ 1 < k \leq K \tag{4.7}$$

3. Termination:

$$P^* = \gamma_K(j) = \max_{i=1}^{n} \gamma_K(i) \tag{4.8}$$

$$q_K^* = \chi_K(j) = \operatorname{argmax}_{i=1}^{n} \gamma_K(i) \tag{4.9}$$

4. Path backtracking:

$$q_k^* = \chi_{k+1}(q_{k+1}^*), \qquad k = K-1, K-2, \cdots, 1 \tag{4.10}$$

Where:
P^* is the probability of the observation sequence O given the optimal state sequence Q^* and the Markov chain M_{ch}
q_K^* is the start of the backtrace, that is the index of the $K-1$'th state in Q^*

The Viterbi algorithm described above determines the single best state sequence for the given hidden Markov chain and observation sequence, or in other words; it finds the state sequence Q which maximises $P[Q|O, M_{ch}]$. The complexity of the algorithm is $\mathcal{O}(n^2k)$, where n is the number of states in the system and k is the sequence length. A change in the algorithm has been made such that the γ-values are normed with the smallest of the γ-values at each time step. This way it is avoided that

the γ-values turns to zero because of machine representation, while keeping the order between the γ-values the same.
With the algorithm defined it is possible to implement it in Matlab and test if it is implemented correctly.

4.1.1 Implementation of Viterbi Algorithm

A test of the implementation of the algorithm can be found in appendix E. The main results and conclusions from the test will be summarised in this section.
The test is conducted by simulating the output from the system using the pure hidden Markov model of the system. The A and B parameters used in the Viterbi algorithm are varied, while the same parameters are kept fixed in the simulation of the system. The error percentages of the Viterbi algorithm are then measured at the different parameters. If the algorithm is correctly implemented, the algorithm which match the system parameters should be optimal regarding error percentages or at least at par with the best. The error percentage is measured by counting how many of the states in the state sequence is estimated wrong.
The simulation has been run on values of the A and B parameters in the Viterbi algorithm ranging from 0.05 to 0.95 with an interval of 0.05, which gives a total number of test points of 361. Five different tests has been run. In each of the tests the A and B parameters used by the hidden Markov model, are fixed. The values of the parameters used by the hidden Markov model are: $A = 0.25, B = 0.25$; $A = 0.25, B = 0.75$; $A = 0.50, B = 0.50$; $A = 0.75, B = 0.25$ and $A = 0.75, B = 0.75$ respectively. The plot in Figure 4.2 shows the results from the test where the A and B parameters in the model are 0.25. As it can be seen in the figure, there is a large part of the surface where the error percentages are all lower than in the rest of the plot, and the Viterbi parameters that match the system parameters are placed in this part of the surface.
The same tendencies apply for the other four tests, it is therefor concluded that the algorithm is

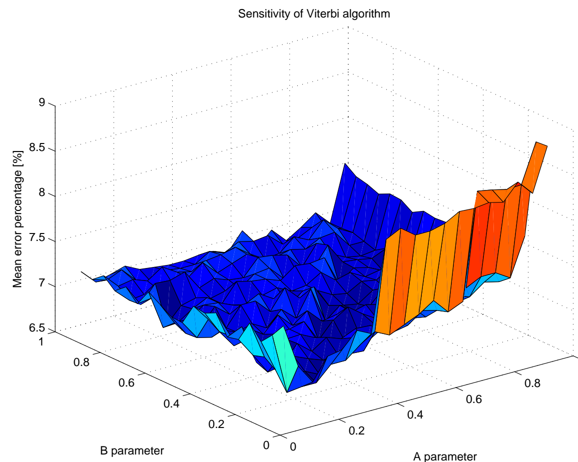


Figure 4.2: *Error percentages at different A and B parameters of Viterbi algorithm at system parameters: $A = 0.25$ and $B = 0.25$*

implemented correctly.

4.1.2 Test of Viterbi Algorithm Time Consumption

With the implementation of the Viterbi algorithm verified it is necessary to test the time consumption of the algorithm in order to assess if it is possible to use the algorithm as state sequence estimator in a system in the scale of the distribution system in question, where approximately 4e6 assets and 20e3 locations are present.
A test have been run where the time consumption of the algorithm has been measured at different sequence lengths and at different numbers of locations in the system. The test is run on simulated

data, which comes from the simulation model implemented in Matlab Simulink.
The entire test report can be found in appendix F. The main results and conclusions from the test will be summarised in this section. The results of the test is shown in Figure 4.3 which shows the calculation time for the algorithm. As it can be seen in Figure 4.3 the calculation time for the Viterbi algorithm is linear in the sequence length and quadratic in the number of locations in the system, this can be seen more clearly in the cross section plots shown in Figure 4.4 and Figure 4.5, which shows a number of cross sections of the plot in Figure 4.3. This fits the expected computational complexity. The plot in Figure 4.4 shows that it takes about 16 seconds to calculate the state sequence of the asset if there are 55 locations in the system and the sequence length is 200, which corresponds the topmost line in the plot.
Since the full scale system is expected to consist of 20e3 locations and 4e6 assets, the calculation time has been extrapolated to a system consisting of 20e3 locations. The plot in Figure 4.6 shows the

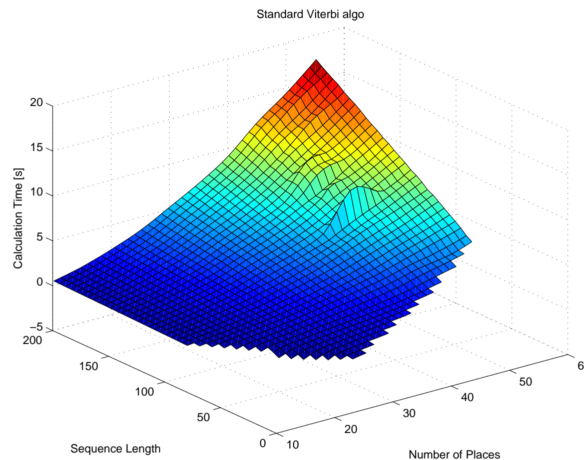


Figure 4.3: *Plot of calculation time for the Viterbi algorithm*

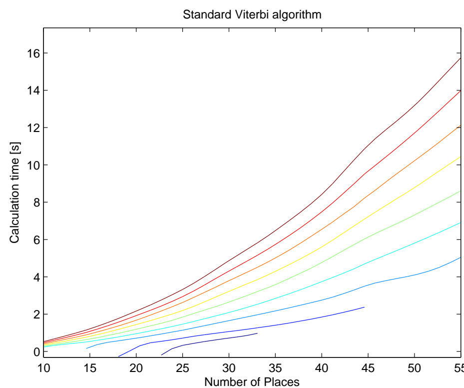


Figure 4.4: *Plot of cross sections at different sequence lengths of Figure 4.3*

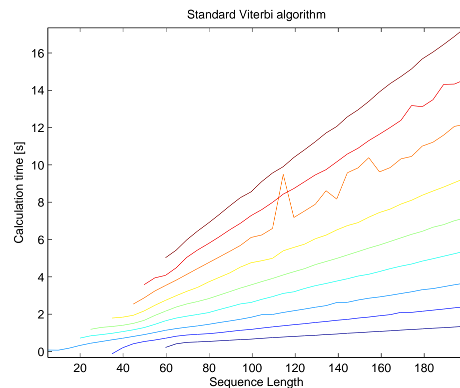


Figure 4.5: *Plot of cross sections at different numbers of locations of Figure 4.3*

measured data along with a Matlab polyfit of the topmost line in Figure 4.4. The plot in Figure 4.7 shows the calculation time extrapolated to 20e3 locations using the polynomial found with polyfit. The value estimated at 20e3 locations is with a mean (μ) of 2.01e6 seconds and a standard variation (σ) of 3.88e4 seconds. Using the extrapolated calculation time it is estimated that it will take the algorithm between 1.93e6 and 2.09e6 ($\mu \pm 2\sigma$) seconds to calculate the state sequence of a single asset at a 2σ level of confidence if there is 20e3 locations in the system and the sequence length is 200. If this number is scaled to a system consisting of 4e6 assets it would take between 245e3 and 265e3 years to calculate the state sequence of all assets on a single computer.
If it is assumed that a computer is present at each location in the system, such that the computational

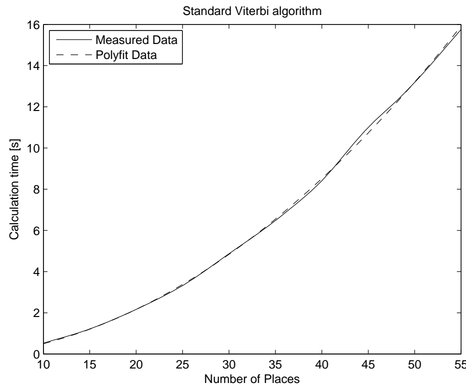


Figure 4.6: *Illustration of measured data along with the polyfit at a sequence length of 200*

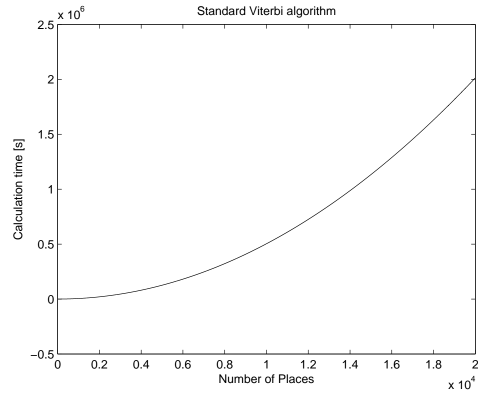


Figure 4.7: *Extrapolation of calculation time using polyfit at a sequence length of 200*

burden can be distributed to each of these, it would take between 12 and 13 years to calculate the state sequences. This is infeasible to be useful since a sequence length of 200 corresponds to a time span of just above four days if a day is split into intervals of half an hour. If the Viterbi algorithm is to be used for model based estimation of state sequences in a system of this scale it has to be simplified in order to reduce its computational complexity. Therefor, an approach is needed to decrease the computational complexity of the algorithm.

4.2 Customised Viterbi Algorithm

As it is concluded in the previous section, that the Viterbi algorithm is unusable as the time consumption is much greater than the time covered by the length of the state sequence even if the algorithm is distributed. Because of this fact, the algorithm has to be simplified in order to reduce the computational time.

This section deals with a customisation of the Viterbi algorithm. The section starts with a description of how the system matrices can be reduced to data structures of fixed sizes, and is concluded with a description of the customised Viterbi algorithm to be used in the system.

4.2.1 Reduction of System Parameters

Due to the nature of the system, the system parameters can be reduced to data structures much smaller than the ordinary system matrices used in hidden Markov models. The reason for this is illustrated in Figure 4.8. The figure shows part of the asset automaton described in section 2.2.2. The

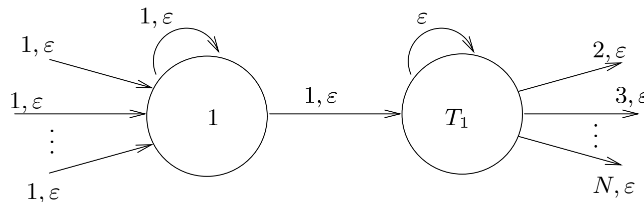


Figure 4.8: *Illustration of a single location state along with its corresponding transport state, the symbols alongside the arrows indicate the possible output symbols for the specific transition*

part illustrated in Figure 4.8 models one specific location and its neighbourhood. As it can be seen in the figure the location is modelled by a state representing the specific location or warehouse, this is the state marked '1'. Furthermore, it consists of an associated transport state marked 'T_1', which models the event that the asset leaves the location. The neighbourhood of the location is illustrated

by the arrows going into the location state and the arrows coming out of its associated transport state. The symbols alongside the arrows indicates the outputs which are possible to emit when taking the transition associated with the arrow.

Transition Matrix

When the initial parameters for the asset automaton is set up, the transitions of the same type are considered equally likely. That is; it is for instance equally likely for an asset to take the self loop in the location state no matter which location the asset is located at. This in turn means that there will be redundant parameters in the system matrices. In fact it is only necessary to describe the state transitions in the system by four parameters, instead of the entire matrix Φ. These parameters are: the probability of taking a self transition in the location state ($P[x_j \rightarrow x_j]$), the probability of taking a transition from the location state to the corresponding transport state ($P[x_j \rightarrow T_j]$), the probability of taking a self transition when in a transport state ($P[T_j \rightarrow T_j]$) and the probability of taking the transition from a transport state to a location state ($P[T_j \rightarrow x_i]$).

Observation Matrix

By making similar considerations it can be realised that the output distribution matrix Ω, can be reduced to a vector consisting of two parameters: the probability of emitting the symbol v_j when entering or leaving the location state ($P[v_j|T_i \rightarrow x_j], P[v_j|x_j \rightarrow T_j]$) and the probability of emitting the symbol v_j when taking the self transition in a location state ($P[v_j|x_j \rightarrow x_j]$). The two first situations can be described by the same parameter, since they both require the asset to be transported through the RFID reader gate at the physical location, thus resulting in the same probability of emitting the symbol corresponding to the specific location. The latter output parameter is a special case since it describes non intended readings of the RFID tag when transporting it locally around within a location without the intention of transports to another location. The probabilities of emitting the empty symbol (ε) when taking a transition, can be calculated from the other parameters. Furthermore, the probability of emitting ε when taking the self transition in the transport state is always 1 since no RFID readings are generated while the assets are being transported, so this is not necessary to be passed as a parameter to the algorithm.

Initial Distribution Vector

Finally, the initial state distribution $\boldsymbol{\pi}$, can be described by a scalar, since the different locations are considered equally likely as initial state. Thus all the parameters necessary to describe the system are fixed size and does not grow with the number of states in the system, as is the case with the system matrices described in section 2.2.2.

4.2.2 Derivation of Custom Viterbi Algorithm

Because of the nature of the system, it is possible to reduce the complexity of the Viterbi algorithm, by customising it for use with the logistic system of interest. The considerations used in the derivation of the algorithm are similar to those used in the derivation of the reduced system parameters described previously. Since the parameters of the different parts of the asset automaton describing a particular location are the same, many of the states in the system will be assigned the same γ-value in the Viterbi trellis. Thus, this value can be calculated once and then applied to all the states for which this particular γ-value applies.

The only way a state can obtain a γ-value different from other states, is if the output sequence from the Markov chain contains a symbol corresponding to the particular location. As it can be seen in Figure 4.8, the three possibilities for the Markov chain to generate the output '1' is; if the transition from another transport state is taken into location state '1', if the location '1' state makes a self transition or lastly if the location '1' state makes a transition to its associated transport state 'T_1'. These considerations applies to arbitrary location states. This means that if the output from the

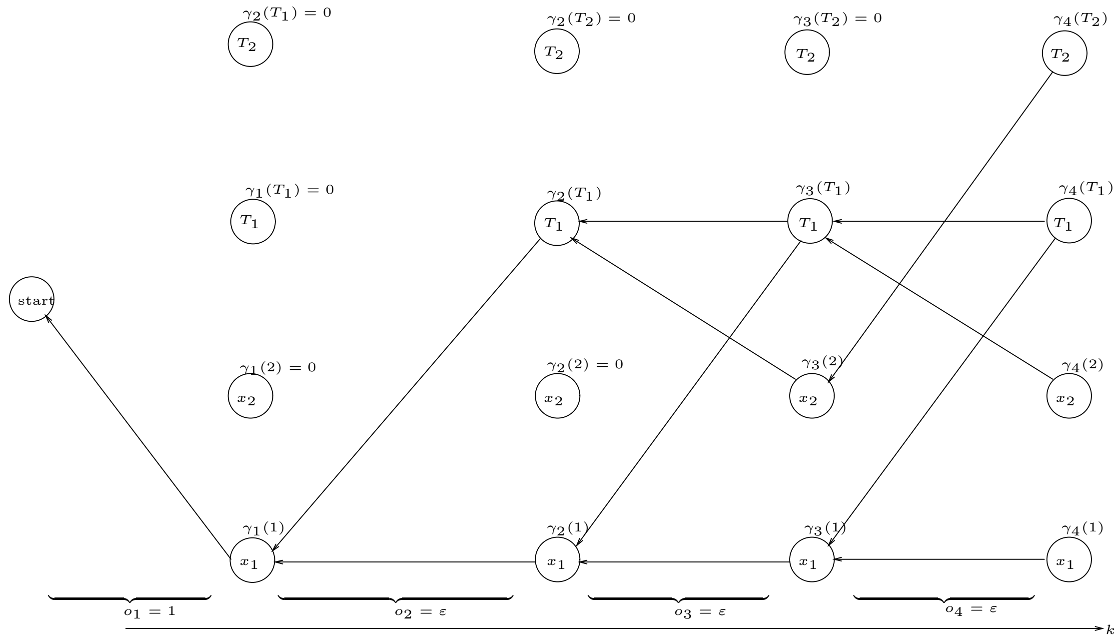


Figure 4.9: *Example of Viterbi trellis in the distribution system, where the output vector starts with a non-empty output. The arrows indicate (potential) backpointers*

Markov chain is '1', the only possibilities for the present state is either location '1' or its transport state 'T_1', all other states are impossible solutions and can therefor be assigned a γ-value of zero.

For every empty symbol following another empty symbol in the output sequence, the probability distribution of the states in the state sequence will be spread out on all states. This is illustrated in Figure 4.9. The figure shows part of an example Viterbi trellis, where the arrows indicate possible candidates for backpointers. The example uses an output sequence, where the initial symbol is a valid location symbol and the rest are empty outputs. From the figure it can be seen that if the output sequence starts with a location symbol it is only possible for the asset to be in the state corresponding to the given symbol. The more empty outputs following a valid symbol, the more the probability distribution will be spread out on all the states. If the second symbol is an empty symbol it is possible to be located in the transport state corresponding to the location. If the next symbol is also empty, it can be possible to be in one of the other location states, and because of this these location states will be assigned a non-zero γ-value. This γ-value will be the same for all other states than the state for which the previous non-empty symbol applied, and because of this it is only necessary to calculate one γ-value to apply to all of these states.

If yet another empty symbol follows, it is furthermore possible that the automaton is in the one of the remaining transport states, so these will in turn be assigned a non zero γ-value. Because of the symmetry in the system all location states not corresponding to the latest non empty output symbol will be assigned the same γ-value in the Viterbi trellis. Likewise, all the transport states not corresponding to the latest non empty measurement will be assigned the same γ-value. For transport states there are two possible candidates for the χ- and γ-values if the output was an empty symbol, depending on which transition maximises the γ-value. Because of this two arrows are shown as backpointers on the figure for these transport states.

Figure 4.10 shows a trellis example where the output sequence starts with an empty symbol. As illustrated in the figure this means that the asset can have started out in anyone of the locations and because of this, the probabilities are evenly distributed among the location states. If another empty symbol follows it is furthermore possible for the asset to be in the transport states with equal probability. This will continue until a valid output is produced in the output sequence.

Finally, Figure 4.11 shows an example of a trellis with two different location symbols in the output sequence separated by a number of empty symbols. When this is the case the location state corresponding to the latter output will have its backpointer directed at the transport state of the state which has produced the former output.

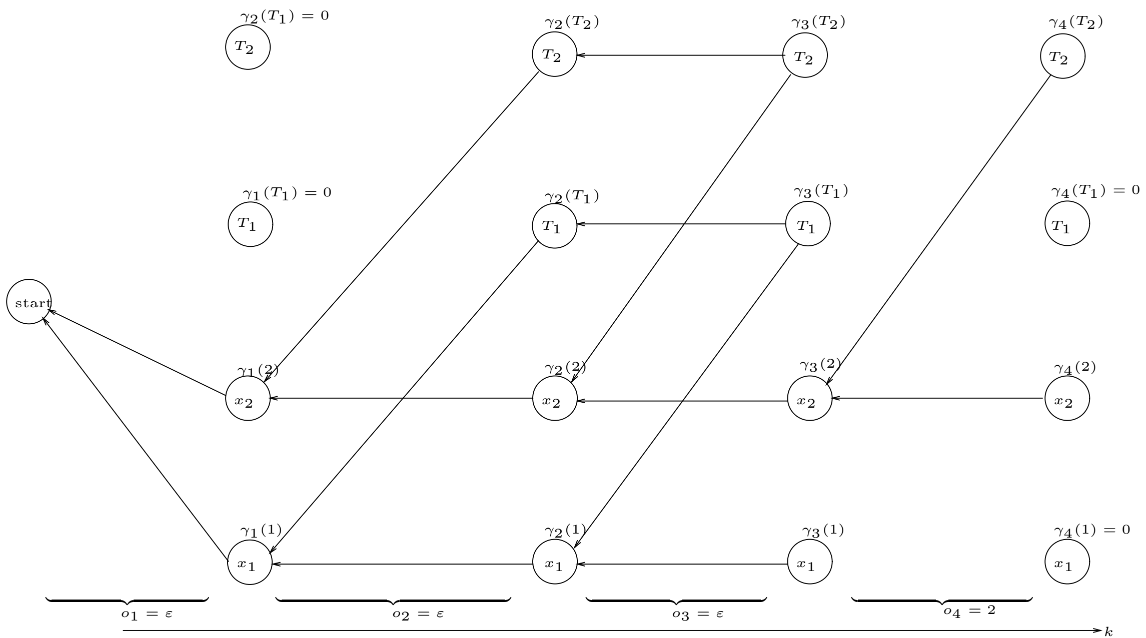


Figure 4.10: *Example of Viterbi trellis in the distribution system, where the output vector starts with an empty output*

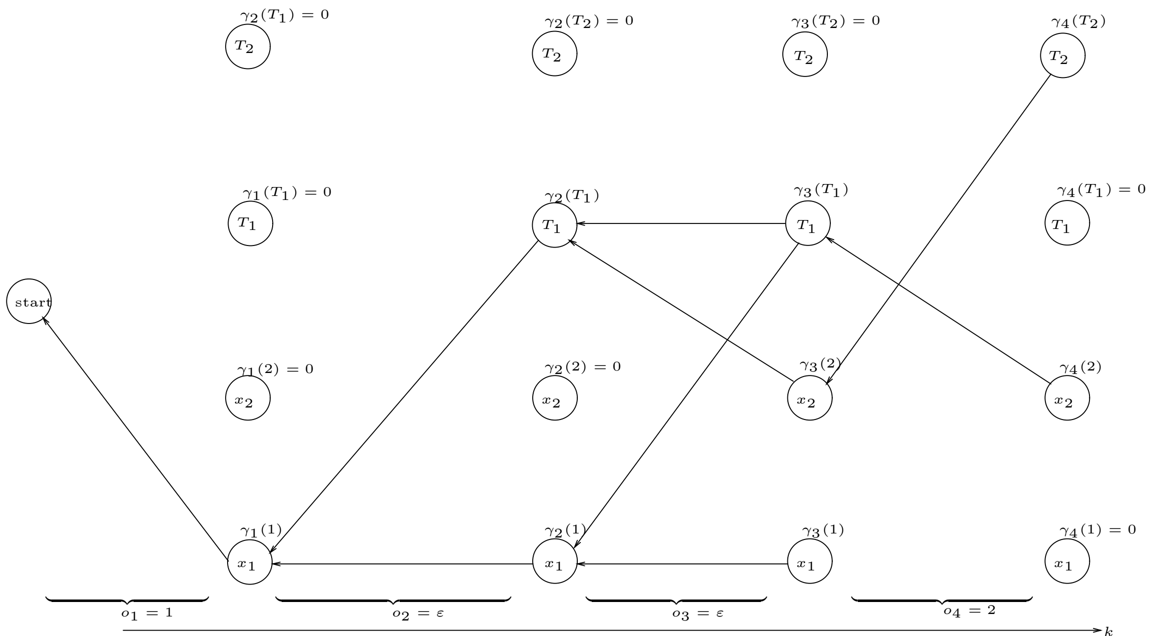


Figure 4.11: *Example of Viterbi trellis in the distribution system, where the output vector starts with a non-empty output, and has another non-empty output at a later time index*

These considerations lead to the conclusion, that it is only necessary to calculate maximum four γ-values for each time index in order to fill the Viterbi trellis. The location and transport state corresponding to the latest non empty measurement are treated as special cases and are evaluated by themselves. The rest of the location states are assigned the same γ-value and so are the rest of the transport states.
The customised Viterbi algorithm is listed below. The states denoted by x are location states and the states denoted by T are transport states.

1. Initialisation:

- if $o_1 = \varepsilon$:

$$\forall j \quad \gamma_1(j) = P[\text{start} \rightarrow x_j] \cdot P[\varepsilon|\text{start} \rightarrow x_j], \quad \gamma_1(T_j) = 0 \tag{4.11}$$

$$\forall j \quad \chi_1(j) = 0, \quad \chi_1(T_j) = 0 \tag{4.12}$$

- if $o_1 = v_i \neq \varepsilon$

$$\gamma_1(i) = P[\text{start} \rightarrow x_i] \cdot P[v_i|\text{start} \rightarrow x_i], \quad \forall j \neq i \quad \gamma_1(j) = 0, \quad \forall j \quad \gamma_1(T_j) = 0 \tag{4.13}$$

$$\forall j \quad \chi_1(j) = 0, \quad \chi_1(T_j) = 0 \tag{4.14}$$

2. Recursion:

- if $o_k = v_j$:
  - if previous non-empty measurement was v_j:

$$\gamma_k(j) = \gamma_{k-1}(j) \cdot P[x_j \rightarrow x_j] \cdot P[v_j|x_j \rightarrow x_j], \quad \chi_k(j) = j \tag{4.15}$$

$$\gamma_k(T_j) = \gamma_{k-1}(j) \cdot P[x_j \rightarrow T_j] \cdot P[v_j|x_j \rightarrow T_j], \quad \chi_k(T_j) = j \tag{4.16}$$

$$\forall i \neq j \quad \gamma_k(i) = 0 \tag{4.17}$$

  - if previous non-empty measurement was v_i:

$$\gamma_k(j) = \gamma_{k-1}(T_i) \cdot P[T_i \rightarrow x_j] \cdot P[v_j|T_i \rightarrow x_j], \quad \chi_k(j) = T_i \tag{4.18}$$

$$\gamma_k(T_j) = \gamma_{k-1}(j) \cdot P[x_j \rightarrow T_j] \cdot P[v_j|x_j \rightarrow T_j], \quad \chi_k(T_j) = j \tag{4.19}$$

$$\forall h \neq j \quad \gamma_k(h) = 0 \tag{4.20}$$

- if $o_k = \varepsilon$
  - if previous non-empty measurement was v_j:

$$\gamma_k(j) = \gamma_{k-1}(j) \cdot P[x_j \rightarrow x_j] \cdot P[\varepsilon|x_j \rightarrow x_j], \quad \chi_k(j) = j \tag{4.21}$$

$$\gamma_k(T_j) = \max(\gamma_{k-1}(j) \cdot P[x_j \rightarrow T_j] \cdot P[\varepsilon|x_j \rightarrow T_j], \gamma_{k-1}(T_j) \cdot P[T_j \rightarrow T_j] \cdot P[\varepsilon|T_j \rightarrow T_j]) \tag{4.22}$$

$$\chi_k(T_j) = \text{argmax}(\gamma_{k-1}(j) \cdot P[x_j \rightarrow T_j] \cdot P[\varepsilon|x_j \rightarrow T_j], \gamma_{k-1}(T_j) \cdot P[T_j \rightarrow T_j] \cdot P[\varepsilon|T_j \rightarrow T_j]) \tag{4.23}$$

$$\forall i \neq j \quad \gamma_k(i) = \gamma_k(T_j) \cdot P[T_j \rightarrow x_i] \cdot P[\varepsilon|T_j \rightarrow x_i], \quad \chi_k(i) = T_j \tag{4.24}$$

$$\gamma_k(T_i) = \max(\gamma_{k-1}(i) \cdot P[x_i \rightarrow T_i] \cdot P[\varepsilon|x_i \rightarrow T_i], \gamma_{k-1}(T_i) \cdot P[T_i \rightarrow T_i] \cdot P[\varepsilon|T_i \rightarrow T_i]) \tag{4.25}$$

$$\chi_k(T_i) = \text{argmax}(\gamma_{k-1}(i) \cdot P[x_i \rightarrow T_i] \cdot P[\varepsilon|x_i \rightarrow T_i], \gamma_{k-1}(T_i) \cdot P[T_i \rightarrow T_i] \cdot P[\varepsilon|T_i \rightarrow T_i]) \tag{4.26}$$

  - if there have been no non-empty measurements:

$$\forall j \quad \gamma_k(j) = \gamma_{k-1}(j) \cdot P[x_j \rightarrow x_j] \cdot P[\varepsilon|x_j \rightarrow x_j], \quad \chi_k(j) = j \tag{4.27}$$

$$\gamma_k(T_j) = \max(\gamma_{k-1}(j) \cdot P[x_j \rightarrow T_j] \cdot P[\varepsilon|x_j \rightarrow T_j], \gamma_{k-1}(T_j) \cdot P[T_j \rightarrow T_j] \cdot P[\varepsilon|T_j \rightarrow T_j]) \tag{4.28}$$

$$\chi_k(T_j) = \text{argmax}(\gamma_{k-1}(j) \cdot P[x_j \rightarrow T_j] \cdot P[\varepsilon|x_j \rightarrow T_j], \gamma_{k-1}(T_j) \cdot P[T_j \rightarrow T_j] \cdot P[\varepsilon|T_j \rightarrow T_j]) \tag{4.29}$$

3. Termination: Same as in standard Viterbi algorithm.

4. Path backtracking: Same as in standard Viterbi algorithm.

The complexity of the algorithm can be verified to be $\mathcal{O}(k)$, where k is the length of the observation sequence. As opposed to the standard Viterbi algorithm which has complexity $\mathcal{O}(n^2k)$, where n is the total number of states in the system. The algorithm has been verified to produce the same γ- and χ-matrices as the standard Viterbi algorithm.

4.2.3 Test of Custom Viterbi Algorithm

The custom Viterbi algorithm has been tested in order to determine if its performance in terms of the error percentage is similar to the performance of the standard Viterbi algorithm, such that it is possible to use as a replacement for the standard Viterbi algorithm as state sequence estimator in the system and if it is able to fulfil the requirement of estimating the state correct in 98.75 % of the time. Furthermore, the time complexity of the algorithm has been tested in order to verify that it has the expected complexity, which is $\mathcal{O}(k)$ where k is the sequence length, and to verify that the algorithm is feasible for use in a system in the scale of the distribution of cc containers in Europe, which consist of approximately 4e6 assets and 20e3 locations. The full test reports can be found in appendix G and H, for the test of the error percentage and time complexity respectively. The main results and conclusions will be summarised in this section.

Error Percentage

The results from the test of the error percentage is shown in Figure 4.12 and Figure 4.13, which show the results from the standard Viterbi algorithm and the customised Viterbi algorithm respectively. As

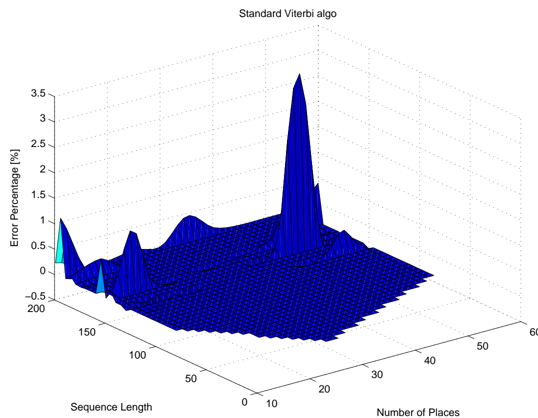


Figure 4.12: *Plot of estimation error percentage for the standard Viterbi algorithm*

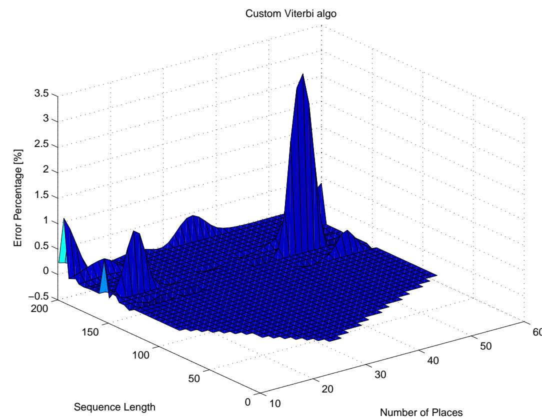


Figure 4.13: *Plot of estimation error percentage for the customised Viterbi algorithm*

it can be seen in the two figures the two different algorithms produce exactly the same results on the same output sequences which is a strong indication that the customised algorithm produces the same estimations as the standard algorithm for the system in question. It can also be seen that the number of errors in the estimated sequence is generally low if the sequence is long or if there is many locations in the system. The results show a mean percentage of erroneous states in the estimated sequences of approximately 0.05 with a standard deviation of 1.89. These results indicate that the estimator currently is able to fulfil its requirement when considering the mean value. But taking the relatively high standard deviation into account it is concluded that it is not able to fulfil its requirement.

Time Consumption

Figure 4.14 shows that the custom algorithm is linear in the sequence length as expected but it is not constant in the number of locations in the system as described in section 2.2.2. This can be seen more clearly in the cross section plots in Figure 4.15 and Figure 4.16. The reason for the calculation time of the customised Viterbi algorithm not being entirely constant in the number of locations in the system is expected to be caused by the choice of implementation. The algorithm treats the states corresponding to the latest non-empty measurement as special cases and calculates special probabilities for those states, all other states of the same type are assigned the same probabilities due to the symmetry of the system. Because of this the algorithm is supposed to have complexity $\mathcal{O}(k)$. Attempts to simplify the γ- and χ matrices used in the algorithm has not been done. Therefor, the latter probabilities

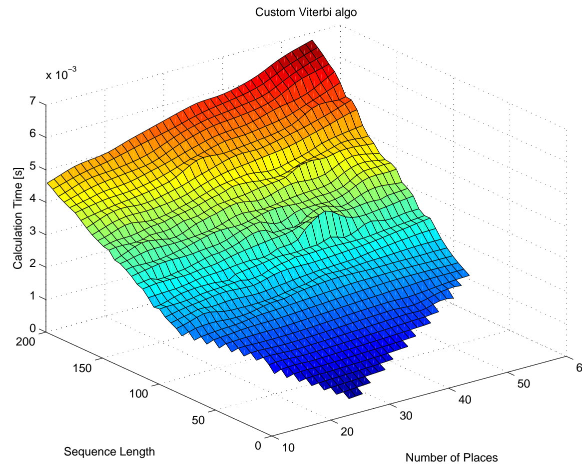


Figure 4.14: *Plot of calculation time for the customised Viterbi algorithm*

are first calculated and then assigned to the states using array assignments in Matlab. The time consumption of the array assignment is expected to be proportional to the number of states and thus the number of locations in the system, and because of this the time consumption of the algorithm is not constant in the number of locations in the system. The plot in Figure 4.17 shows the measured

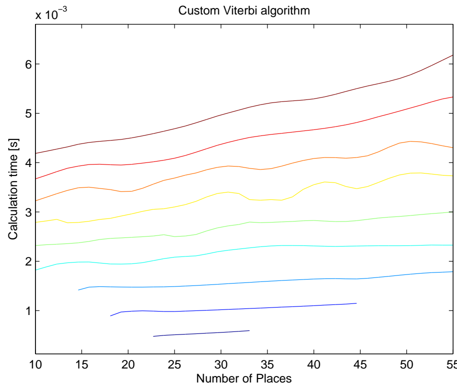


Figure 4.15: *Plot of cross sections at different sequence lengths of Figure 4.14*

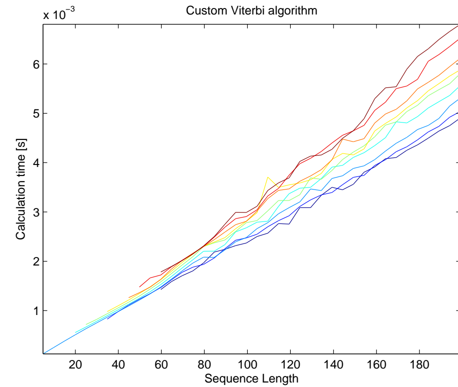


Figure 4.16: *Plot of cross sections at different numbers of locations of Figure 4.14*

data along with a Matlab polyfit of the topmost line in Figure 4.15. The plot in Figure 4.18 shows the calculation time extrapolated to 20e3 locations using the polynomial found with polyfit. The value estimated at 20e3 locations is with a mean (μ) of 0.822 seconds and a standard variation (σ) of 0.016 seconds. From Figure 4.18 it is estimated that it will take the algorithm between 0.790 and 0.854 ($\mu \pm 2\sigma$) seconds to calculate the state sequence of a single asset at a 2σ level of confidence if there is 20e3 locations in the system and the sequence length is 200. If this number is scaled to a system consisting of 4e6 assets it would take between 36.5 and 39.5 days to calculate the state sequence of all assets on a single computer. If it is assumed that a computer is present at each location in the system, such that the computational burden can be distributed to each of these, it would take between 158 and 171 seconds to calculate the state sequences. This is well within the four days that a sequence length of 200 corresponds if a day is split into intervals of half an hour. Alternatively, one could have the computational burden distributed to ten computers which would bring the calculation time to approximately four days. Therefor it is concluded that the algorithm is feasible for use in the system if it is implemented on a distributed platform.

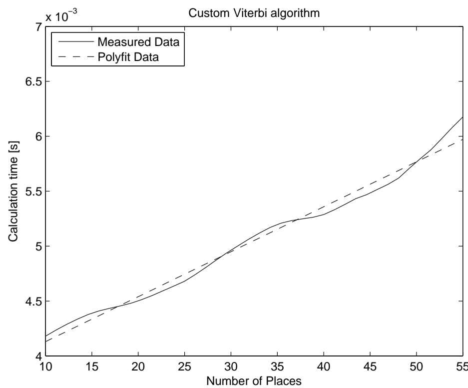


Figure 4.17: *Illustration of measured data along with the polyfit at a sequence length of 200*

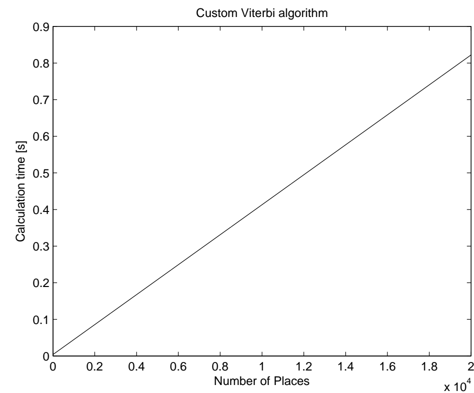


Figure 4.18: *Extrapolation of calculation time using polyfit at a sequence length of 200*

4.3 Graphical User Interface

A GUI has been constructed, with the intention of creating an easy way to manipulate the parameters used in the Simulink model of the distribution system, as well as creating an easy way of running the algorithms and presenting the results. An illustration of the GUI is shown in Figure 4.19. A full description of the GUI, can be found in appendix K.

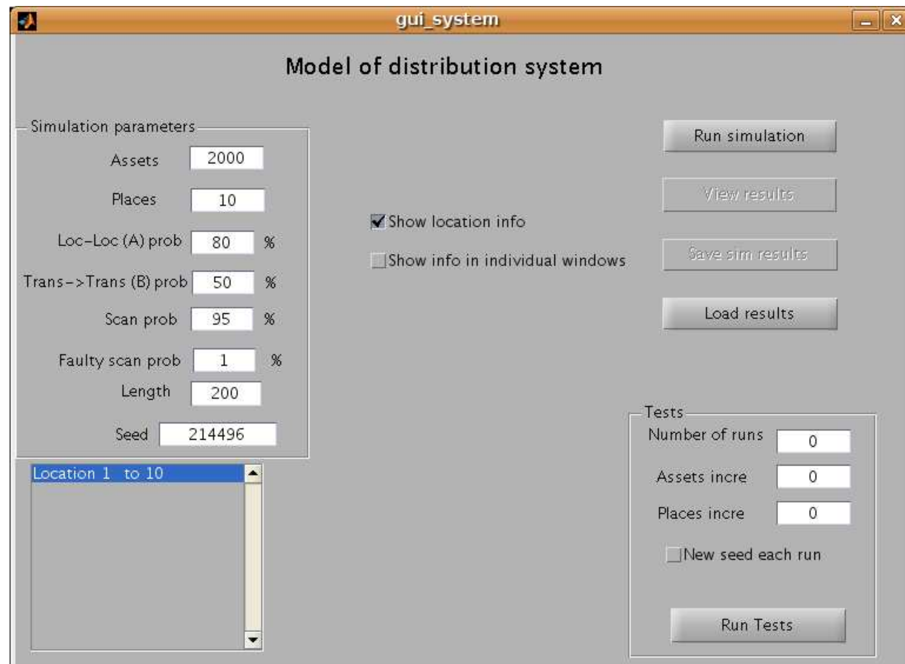


Figure 4.19: *The main window of the graphical user interface*

The next chapter describes the design of a distributed version of the customised Viterbi algorithm.

Chapter 5

Distribution of Algorithm

This section describes the practical issues, which has to be considered if the designed algorithm is to be implemented to work in a full scale system, which consists of approximately 20e3 locations. This would make the algorithm in its current form very time- and resource consuming, as can be seen in section 4.2.3. This fact leads to analysis of other implementations methods for the algorithms. If a microprocessor is present at each location in the distribution system, these processors could be used to process a fraction of the overall computation problem, thus making the algorithm distributed, an illustration of the two different approaches is shown in Figure 5.1 and Figure 5.2. The distributed system would be self scalable, as when a new location is introduced into the system, an extra processor would be available. The following chapter, will include an analysis of the terms, which should be considered to distribute the Viterbi algorithm between multiple computers.

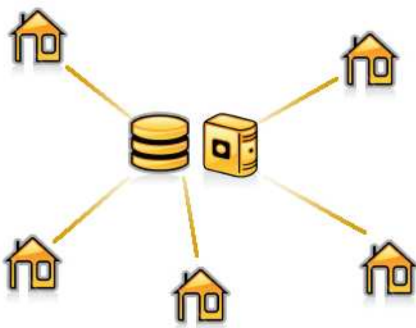


Figure 5.1: *Non distributed algorithms, data are transferred to a central server for data processing*

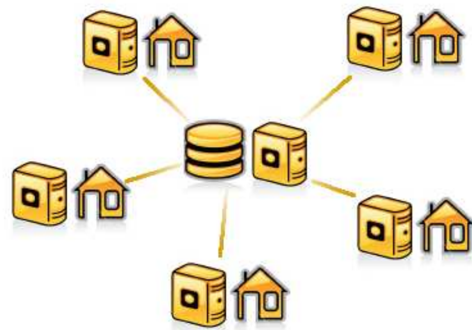


Figure 5.2: *Distributed system, where data are processed locally, and results are merged at central server*

5.1 Distribution Paradigms

Some different methods for distribution of the algorithm exists. In this section some of the methods will be described and analysed, ending up with advantages and disadvantages for each approach. In the distribution setup, multiple computers are cooperating in processing the data. However the data processing can be shared in different manners. Three different methods are considered:

1. Client-server setup; each client manages a specific set of assets, and the server has the information about which client manages which assets. The clients sends the observations to the server and

the server relays the observations to the managing client. On request from the server, possibly from the user, the client runs the Viterbi algorithm. The clients transfers the estimation result to the server, which merges the estimation data.

2. Peer-to-peer setup (1); each client manages a specific set of assets, information of observations are received from the other peers. The estimation result is transferred to the server upon request. This approach will require some occasional load balancing.

3. Peer-to-peer setup (2); each client manages varying assets, depending on where the asset have been observed. When an asset is observed at a location, a request is sent the server, which relays this request to the old managing location, which replies directly to the new managing location, with the observation sequence. The estimation result is transferred to the server upon request.

These approaches all have advantages and disadvantages, including different bandwidth demands, and managing complexity. The client-server based approach is low in managing complexity, as the managing location for a specific asset is static in runtime, and all communication from the locations is always to or from the server. Demand for bandwidth is large at the server location, and an equally low demand exists at the clients. The first peer-to-peer setup, reduces bandwidth demand at the server location, as data is transferred directly from the observation location, and to the managing location of the observed asset. The second peer-to-peer approach, has a high bandwidth demand compared to the previous approach, as the state sequence should be transferred between locations, as assets are transported. Furthermore, the second peer-to-peer setup require more computational resources at locations where many assets are observed than at locations where few assets are observed, which can be an advantage or disadvantage depending on ones point of view, since it can be assumed that locations where many assets are observed will have more financial resources.
From these considerations the first peer-to-peer setup has been chosen. This is done because it can be designed such that it distributes the computational load evenly to the peers in the system, while keeping the bandwidth requirement for the server low. The next section describes the designed load balancing algorithm and an analysis of its steady state distribution properties.

5.2 Load Balancing

When the Viterbi algorithm is distributed between the locations in the system, some kind of load balancing is necessary in order for the system to perform its best, at all times. One approach to load balancing will be analysed in the following section.

One approach for load balancing, is a selfish load balancing algorithm; suppose the managing of a set of N assets are to be shared as equally as possible amongst a set of m locations. Each time two locations are communicating they share information on the number of assets they each manage. Assets would then be expected to migrate from the overloaded location to the underloaded until the allocation becomes balanced. The locations act selfishly without any centralised control.

Expressions for the steady state values of the mean and variance of the load at each node are derived in order to examine if the load balance algorithm leads to an even distribution of the management of assets to the nodes in the system. The mean value should converge to N/m in a system consisting of N assets and m nodes in order to get an even distribution. The steady state value of the variance should converge to zero as time goes towards infinity, since this means that there will be low or no difference in the number of assets being managed by two arbitrarily chosen nodes.

5.2.1 Mean Value of Load

In order to derive the steady state value of the mean load at each node, an expression of the derivative of the mean value is needed. In order to derive such an expression, the following variables are defined: f_{ij} which is a Poisson process with parameter λ_{ij} and is binary, which describes the probability of node i and j communicating; c_i which is a Poisson process with parameter α_i and is binary, which describes the probability of a new asset being introduced to the system to be managed by node i ; d_i which is a Poisson process with parameter β_i and is binary, which describes the probability of an asset managed by node i being removed from the system; $b_i(t)$ which is the load at node i at time t

and lastly δ which is the fraction of the difference in loads between node j and i to transfer.
With the variables just defined the following equations for derivative of mean value of the load at node i are derived:

$$b_i(t+dt) \quad = \quad b_i(t) + c_i - d_i + \delta \sum_{j=1}^{m} f_{ij}(b_j(t) - b_i(t)) \tag{5.1}$$

Because $E[c_i] = \alpha dt$, $E[d_i] = \beta dt$ and $E[f_{ij}] = \lambda_{ij} dt$, taking expected values on both sides of Equation (5.1) leads to:

$$E[b_i(t+dt)] \quad = \quad E[b_i(t)] + \alpha_i dt - \beta_i dt + \delta \sum_{j=1}^{m} \lambda_{ij} dt (E[b_j(t)] - E[b_i(t)]) \Rightarrow \tag{5.2}$$

$$\frac{d}{dt} E[b_i(t)] \quad = \quad \alpha_i - \beta_i + \delta \sum_{j=1}^{m} \lambda_{ij} (E[b_j(t)] - E[b_i(t)]) \Rightarrow \tag{5.3}$$

$$\frac{d}{dt} E[\overline{b}(t)] \quad = \quad \Delta + \Lambda E[\overline{b}(t)] \tag{5.4}$$

Where:
$\overline{b}(t)$ is the load vector for the system at time t

It follows from the expressions above that if the load at node j is greater than the load at node i assets are added to node i. If the opposite is the case, then assets are taken away from node i. The vector Δ is given by the following:

$$\Delta = \begin{bmatrix} \alpha_1 - \beta_1 \\ \vdots \\ \alpha_m - \beta_m \end{bmatrix} \tag{5.5}$$

The matrix Λ is given by:

$$\Lambda = \delta \begin{bmatrix} \lambda_{11} - \sum_{j=1}^{m} \lambda_{1j} & \lambda_{12} & \lambda_{13} & \cdots & \lambda_{1m} \\ \lambda_{21} & \lambda_{22} - \sum_{j=1}^{m} \lambda_{2j} & \lambda_{23} & \cdots & \lambda_{2m} \\ \vdots & & & & \vdots \\ \lambda_{(m-1)1} & \cdots & \lambda_{(m-1)(m-2)} & \lambda_{(m-1)(m-1)} - \sum_{j=1}^{m} \lambda_{(m-1)j} & \lambda_{(m-1)m} \\ \lambda_{m1} & \lambda_{m2} & \cdots & \lambda_{m(m-1)} & \lambda_{mm} - \sum_{j=1}^{m} \lambda_{mj} \end{bmatrix} \tag{5.6}$$

In order to find the steady state value of the mean load the derivative is set to zero, which leads to the following:

$$\begin{aligned} \frac{d}{dt} E[\overline{b}(t)] &= 0 \Rightarrow \\ \Delta + \Lambda E[\overline{b}(t)] &= 0 \end{aligned} \tag{5.7}$$

If the system is closed, i.e. there is no flow of assets into or out from the system, then the vector Δ will be zero. If this is the case, then it follows from Equation (5.7) that the derivative of the expected value of the load vector will be zero if the load vector consists of equal values. This can be seen by realising that the row sums of the Λ matrix is zero, since all the rest of the elements are subtracted in the summation on the diagonal.

5.2.2 Variance of Load

Just like in the derivation of the expression for the steady state value of the mean load, an expression for the derivative of the variance is needed in order to find the steady state value. First of all an expression for the squared value of the load is needed since:

$$\text{Var}[b_i(t)] \quad = \quad E[b_i^2(t)] - E^2[b_i(t)] \tag{5.8}$$

Since an expression for the second term on the right side in steady state was derived in section 5.2.1 this can be assumed to be constant and because of this it is only necessary to find an expression for the first term. The squared value of the load is given by the following expression:

$$\begin{aligned} b_i^2(t+dt) &= b_i^2(t) + 2b_i(t)c_i - 2b_i(t)d_i + 2b_i(t)\delta \sum_{j=1}^{m} f_{ij}(b_j(t) - b_i(t)) + \\ & c_i^2 + 2c_i d_i + 2c_i\delta \sum_{j=1}^{m} f_{ij}(b_j(t) - b_i(t)) + d_i^2 - \\ & 2d_i\delta \sum_{j=1}^{m} f_{ij}(b_j(t) - b_i(t)) + \delta^2 \sum_{j=1}^{m} \sum_{p=1|p=j}^{N} f_{ij} f_{ip}((b_j - b_i)(b_p - b_i)) \end{aligned} \quad (5.9)$$

Binary variables (c_i, d_i and f_{ij}) which are squared are given by the values themselves. Furthermore, terms which depend on dt^2 are eliminated since terms depending on dt will dominate these as $dt \rightarrow 0$. This leads to the following expression:

$$\begin{aligned} b_i^2(t+dt) &\approx b_i^2(t) + 2b_i(t)\left(c_i - d_i + \delta \sum_{j=1}^{m} f_{ij}(b_j(t) - b_i(t))\right) + \\ & c_i + d_i + \delta^2 \sum_{j=1}^{m} f_{ij}(b_j - b_i)^2 \end{aligned} \quad (5.10)$$

Taking expected values leads to the following expression:

$$\begin{aligned} E[b_i^2(t+dt)] - E[b_i^2(t)] &\approx 2dt\left(\alpha_i E[b_i(t)] - \beta_i E[b_i(t)] + \delta \sum_{j=1}^{m} \lambda_{ij}(E[b_j(t)]E[b_i(t)] - E[b_i^2(t)])\right) + \\ & dt(\alpha_i + \beta_i) + dt\delta^2 \sum_{j=1}^{m} \lambda_{ij}(E[b_j^2] + E[b_i^2] - 2E[b_j(t)]E[b_i(t)]) \Rightarrow \quad (5.11) \\ \frac{d}{dt}E[b_i^2(t)] &\approx 2\left(\alpha_i E[b_i(t)] - \beta_i E[b_i(t)] + \delta \sum_{j=1}^{m} \lambda_{ij}(E[b_j(t)]E[b_i(t)] - E[b_i^2(t)])\right) + \\ & (\alpha_i + \beta_i) + \delta^2 \sum_{j=1}^{m} \lambda_{ij}(E[b_j^2] + E[b_i^2] - 2E[b_j(t)]E[b_i(t)]) \quad (5.12) \end{aligned}$$

Replacing $E[b_i(t)]$ and $E[b_j(t)]$ with the steady state value μ_{ss}, leads to the following expression:

$$\begin{aligned} \frac{d}{dt}E[b_i^2(t)] &\approx 2\left(\alpha_i \mu_{ss} - \beta_i \mu_{ss} + \delta \sum_{j=1}^{m} \lambda_{ij}(\mu_{ss}^2 - E[b_i^2(t)])\right) + \\ & (\alpha_i + \beta_i) + \delta^2 \sum_{j=1}^{m} \lambda_{ij}(E[b_j^2(t)] + E[b_i^2(t)] - 2\mu_{ss}^2) \end{aligned} \quad (5.13)$$

If it is assumed that the system is closed, i.e: no assets enter or leave the system, then α_i and β_i can be assumed to be zero, which gives the following:

$$\begin{aligned} \frac{d}{dt}E[b_i^2(t)] &\approx 2\delta \sum_{j=1}^{m} \lambda_{ij}(\mu_{ss}^2 - E[b_i^2(t)]) + \\ & \delta^2 \sum_{j=1}^{m} \lambda_{ij}(E[b_j^2(t)] + E[b_i^2(t)] - 2\mu_{ss}^2) \end{aligned} \quad (5.14)$$

Under the assumption that $\mu_{ss}^2 \leq E[b_i^2(t)]$ and that the distribution of assets at location i and j is equal, which it is in steady state, the following has to apply in order for the variance to converge:

$$2\delta \geq \delta^2 \Rightarrow \quad (5.15)$$

$$2 \geq \delta \quad (5.16)$$

If the left side of Equation (5.14) is set to zero, which is the case in steady state, it follows that $\mu_{ss}^2 = E[b_i^2(t)]$ is a solution to the problem. This in turn means that the variance is zero according to the definition in Equation (5.8), which is the desired result. The conclusion is that the chosen load balancing scheme will give an even distribution of the load in the system.
Now that the distribution paradigm has been chosen and the load balancing scheme has been assessed to have the desired properties, it is possible to analyse and design the software to use in the distributed version of the algorithm. This is covered in the following chapter.

Chapter 6

Distributed Software

In chapter 5 it was chosen to use a peer-to-peer setup in the distributed estimator. The setup should distribute the computational load evenly amongst the peers in the system, and a selfish load balancing scheme was chosen for this purpose. This chapter covers the analysis, design, implementation and finally the test of the distributed version of the estimator.
As this is a proof of concept of distribution of the Viterbi algorithm, the software will not include all the features that are necessary for the system to work optimal, which includes various kinds of error handling, such as network related faults and sanity check of user input. Furthermore, no graphical user interface has been constructed.

The requirements for the peer software, and for the software, which should be running on the server, will be listed in the following:

- The overall functionality of the distributed system, is to equally share the data processing between all nodes in the system.
- Each peer should have a list, containing the managing location, of each asset, this list is in the following called global list. The global list should be updated when a load balancing routine is executed.
- Peer software should include a load balancing routine.
- When a new peer is connected, the peer should receive a global list from the server.
- When a peer is either not present in the system anymore, or out of service, the server should broadcast an updated global list.
- When an asset is observed, the time of observation and observation location, is transferred to the managing location of that particular asset, according to the global list.
- When the connection to the receiving peer, cannot be established, the problem should be reported to the server.
- When a location is not managing a specific asset anymore, due to load balancing, all data concerning the asset should be transferred to the new managing location.
- State sequences should be transmitted to the server upon requests.

6.1 Description of Events

This section contains a list of events, which can happen in the system, and the corresponding actions.

6.1.1 Events:

- An asset is producing a reading - *When an asset passes through the RFID reading port, data is sent to the computer*
- User requests current status - *When the user, requests to have the state sequences for either all assets or for some assets*
- Data is sent to non managing location - *A RFID reading is sent to a non managing location, due to non updated global list*
- Network error - *When the connection is lost, or cannot be established*
- A new location joins the network - *A new peer connects to the network*

6.1.2 Actions:

- When a measurement is received - *The reading is sent to the managing location*
- When the user requets the current status - *The server broadcasts request of current status, and replies to the user*
- Measurement data is received, for a non managed asset - *Data is relayed to correct managing location, an updated list is sent to the sending location*
- When a network error occour - *Server is notified*
- When a new peer connects - *Global list is sent to new peer*

6.2 Analysis of Distributed Software

This section, will present the analysis of the software for the distributed Viterbi algorithm.

6.2.1 Use case Analysis

The section will contain use case descriptions, and flow diagrams describing the operation of the software. The constructed use case diagram for the system is shown in Figure 6.1.

Use case: Receive Data from RFID Port

The use case is initiated by the reading port. When an asset x passes through the reading port at location a, a look up is done to find the managing location, if this is not local the reading is transferred to the location b, which manages asset x. Figure 6.2 is illustrating the behaviour of the software, for the use case.

Use case: Receive Data from Another Location

The use case is initiated, as the network receives asset data from another location. The location b receives data about asset x from location a. If location b manages asset x, the load balancing routine is executed, and the data is processed, else the data is sent to location c, by location b, and the correct global list is sent to location a. Figure 6.3 is illustrating the behaviour of the software for the use case.

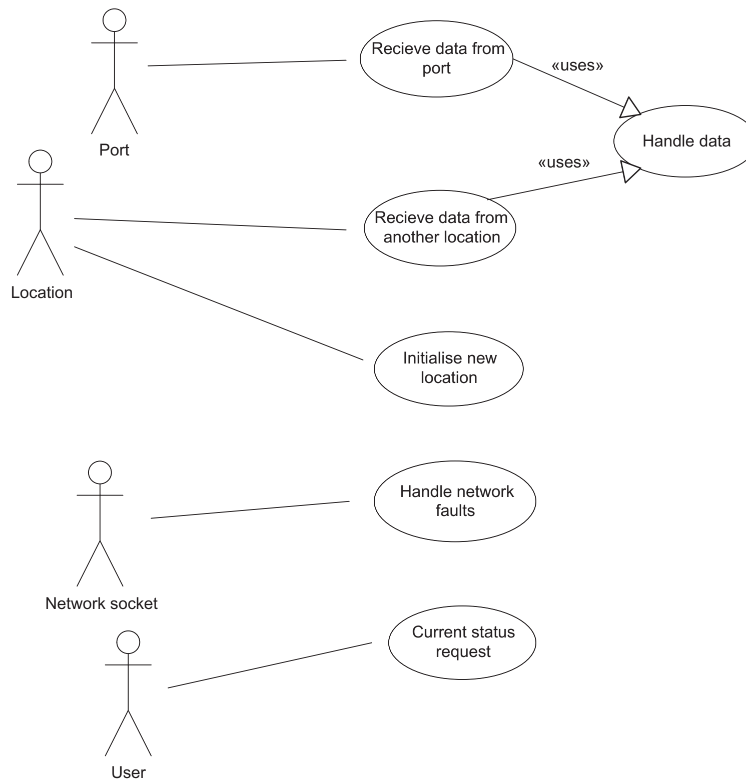


Figure 6.1: *Use case diagram of distributed system*

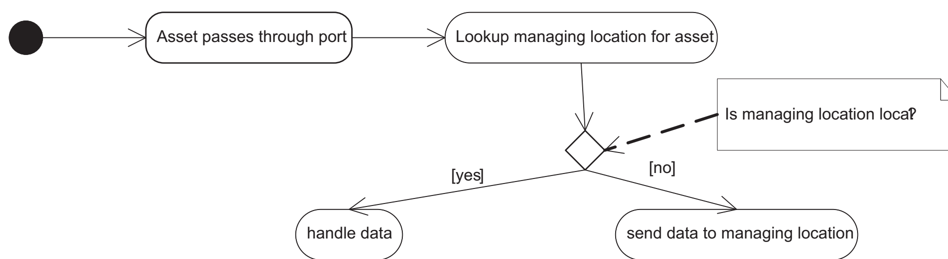


Figure 6.2: *Activity diagram for use case: Receive data from port*

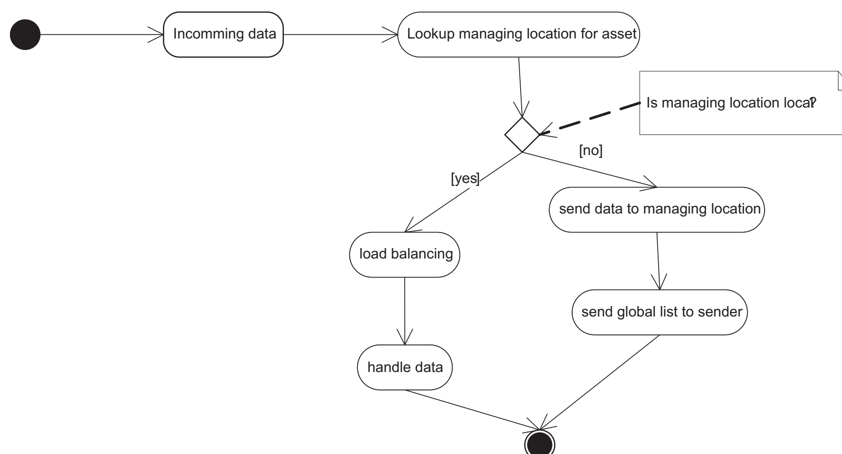


Figure 6.3: *Activity diagram for use case: Receive data from another location*

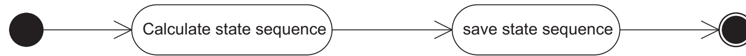


Figure 6.4: *Activity diagram for use case: Handle data*

Use case: Handle Data

The handle data use case, is used by the two previous use cases. When the use case is initiated data is ready to be processed. The object of the use case is to add entries into the observation sequence, and upon request calculate the state sequence for an asset, and save this in the asset data. Figure 6.4 is illustrating the behaviour of the software for the use case.

Use case: Handle Network Faults

The use case is initiated by the network interface, when an error occurs. The use case uses the fault number, to decide which action the system should take. In case of lost connection, the system tries to reconnect to the other peer, if this is not possible, the fault is handled as an unable to connect error, in which case the data is sent to the server. Figure 6.5 is illustrating the behaviour of the software for the use case.

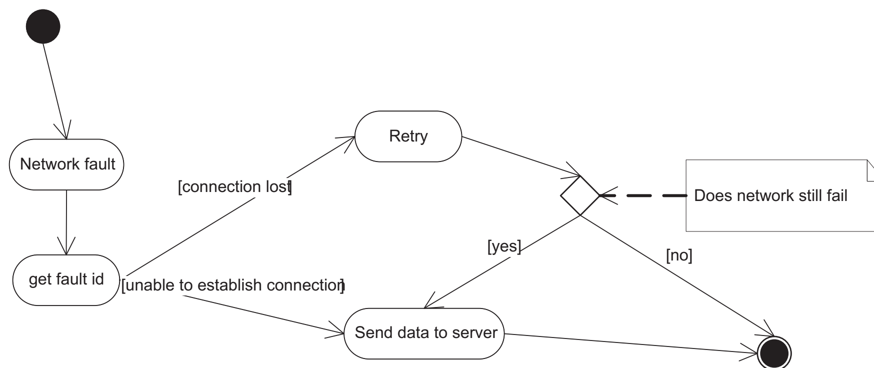


Figure 6.5: *Activity diagram for use case: Handle network faults*

Use case: Current Status Request

The use case is initiated by the user, when the user requests the status of the system, from the user interface. The object of the use case is to receive the current state sequence for all assets. When the server has received the available information, the state sequences should be merged and displayed to the user. The activity diagram for the use case is shown in Figure 6.6.

Use case: Initiate New Location

The use case is initiated by a location when it connects to the network. When a location is connected to the network, it connects to the server and receives the current global list, covering all assets in the system. Figure 6.7 illustrates the behaviour of the software.

6.2.2 Class Diagram Analysis of Distributed Software

Figure 6.8 shows a class diagram which has been constructed based on the use cases. The diagram includes a package *Distributed system*, containing all the classes specific for the system. The standard classes used for i/o handling and networking is not included in the package. The classes used in the system, are described in the following.

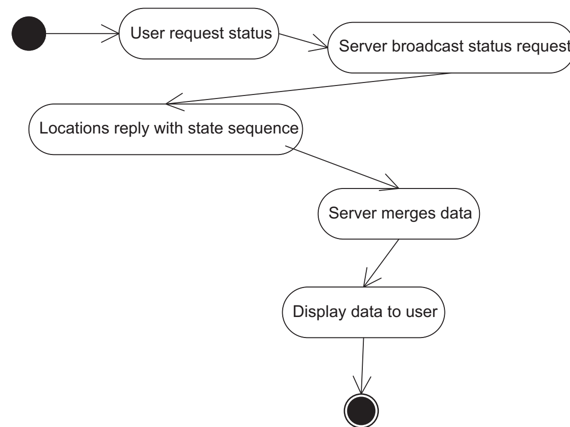


Figure 6.6: *Activity diagram for use case: Current status request*

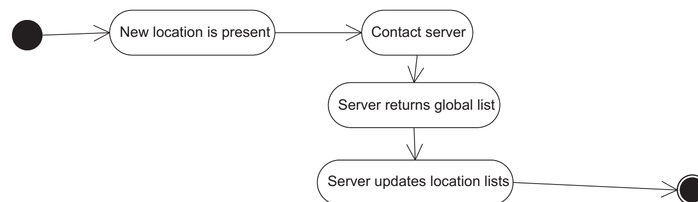


Figure 6.7: *Activity diagram for use case: Initiate new location*

Asset Class

The class holds all the attributes associated with a specific asset. Each asset is associated with an ID, and a type of asset. The type of asset is a parameter, which would be checked if the ID of the asset is not present in the system to present an error to the user, or for the system to handle the error, as the RFID tag registered could be a tag of a different system. The asset object also include the observation sequence and the calculated state sequence for the asset. The operations for the asset class is:

addToObsSeq() adds an entry to the observation sequence.

calcStateSeq(), calcStateSeqOld() calculates the state sequence, using the custom Viterbi algorithm and standard Viterbi algorithm respectively.

calcInitPar(),calcInitParOld() calculates the initial parameters for the custom and standard Viterbi algorithm respectively.

getStateSeq() combines the two preceding functions i order to calculate, both the parameters and the state sequence.

setSystemPar() manipulates the system parameters; number of assets, number of locations and observation probability.

Location Class

The class is associated with each location, with the following attributes; 'location ID' which is a number representing the location, 'physical location', holds a description of the physical location, where the reading port is placed. 'Type', the type attribute, holds the type of reading port, which can be fixed or movable. The blacklist attribute holds the list of locations which are faulty at the moment. Information on faulty locations is important, when a faulty location comes online again, it should contact the server to get the updated lists, as the location might not know that it has been faulty. The global list is a list of all assets in the system, with the corresponding managing location. The managing list is a list of local managed assets. The operands for the location class are:

loadbalancingReq() requests another location for load balancing.

updateGlobalList() updates the global list for the current location.

addToGlobalList() adds an entry to the global list.

sendGlobalList() sends the global list.

findAssetManager() finds the asset in the global list and returns the managing location.

sendLoadnumber() sends the current number of local managed assets.

sendAssetData() sends all information for a certain asset to another location.

addAssetToManagingList() adds an asset to the list of locally managed assets.

removeFromManagingList() deletes assets from the list of local managed assets.

getAssetFromManagingList() returns data for a specific asset.

editGlobalList() edits an entry in the list of all assets.

sendMeasurement() sends an observation for a specific asset.

addObsToAsset() adds an entry to the observation sequence for a specific asset.

initialise() sends a initialise request to the server.

updateBlacklistList() updates the list of currently faulty locations.

setLocationAttributes() manipulates the attributes associated with the location class.

Server Class

The server class holds the status of the server, this is mainly thought to be used if multiple servers are used in the system. The status attribute can hold the current status of the server; active as server, inactive, offline etc. The physicalLocation attribute holds the physical location of the server. The blacklistList is a list with all the faulty locations. The lastStatusRequest attribute hold the date and time of the last status request, where all data has been collected from all locations and merged together. The locationIPAdresses attribute holds all the IP addresses of the peers currently present in the system. The assetData attribute holds the asset data from the last status request. The operands in the server class are;

getAssetInfo() broadcasts a message, on which all location replies with the current data for all assets.

updateBlacklistList() updates the blacklistList locally and broadcasts the new list to all locations.

updateGlobalList() updates the global list.

showAssetData() output the current assetData for all assets to the user.

6.2.3 Dynamic Modelling of the Distributed Software

This section will present a more detailed description of the system behaviour. The states, presented in the activity diagrams in Figure 6.2,6.3,6.4,6.5,6.6 and Figure 6.7 will be described in greater detail, by the use of sequence diagram showing the interaction of the software elements.

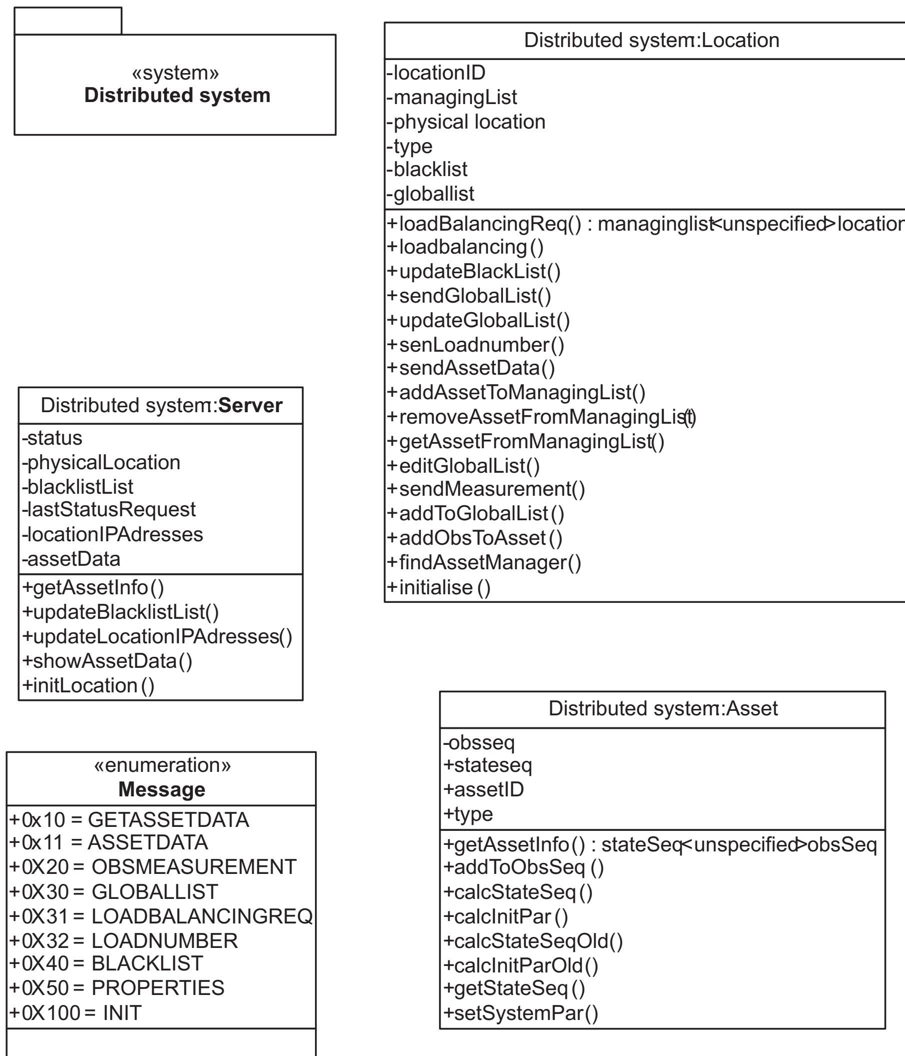


Figure 6.8: *Class diagram for the distributed system*

Non-faulty Operation

The sequence diagram shown in Figure 6.9 illustrates the operation of the system, where no network faults occurs. The diagram shows the interaction between the classes. Three location classes are present in the figure; one at the port location, and two other locations. The sequence diagram first shows the interaction between the locations, in the case where the managing location is the scanning location. In the second case where this is not true, the measurement is sent to the managing location, in this case location 2. If the global list of location 1 is outdated, and location 2 is no longer the manager of the asset, the data is further transmitted to location 3. When a successful transmission of data has occurred, the load balancing routine is run, thus balancing the number of assets between the two locations.

Faulty Operation

The sequence diagram shown in Figure 6.10 illustrates the interaction between the classes, when a location loses its connection to the network. The first event is a port event, as an asset passes through a port. The reading is passed to the client at the location, which either handles the data, or sends it to the managing location. If an error message from the network is returned, indicating that the message could not be delivered, the data is sent to the server. The server tries to contact the faulty

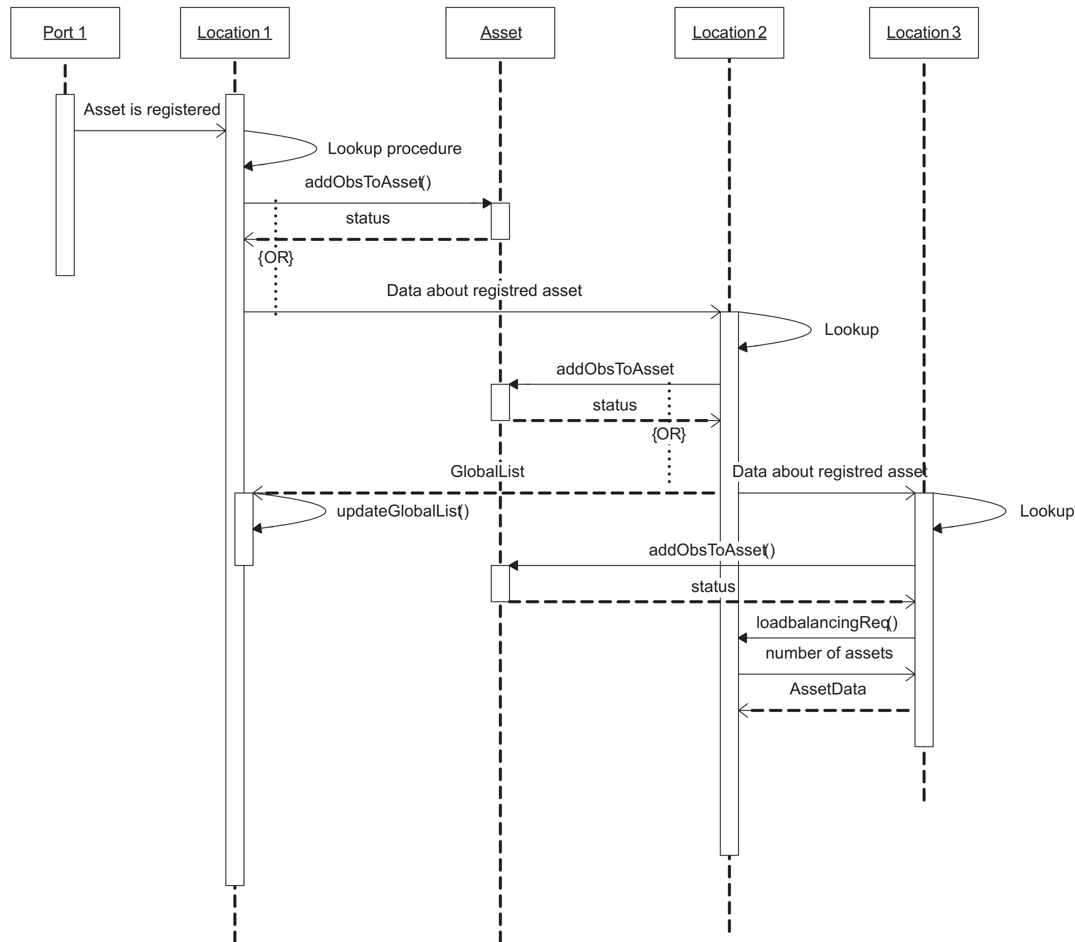


Figure 6.9: *Sequence diagram for normal operation*

location, if this is successful, the system continues in normal operation. If the network returns an error message, the server broadcasts a message saying that location 2 is down. And a randomly chosen location is now the manager of the assets of location 2. When the faulty location is repaired, and goes online again, it registers itself with the server, which broadcasts a message that location 2 should be removed from the blacklist. Measurements are store locally in a buffer at the faulty location, until the connection is reestablished.

Load Balancing

The *loadbalancingReq()* routine is called, when a non-faulty communication has occurred. The purpose of the routine, is as described previously to balance the computational load between all computers in the network. This is done, based entirely on the number of assets managed by each location. The two locations exchange their individual number of assets managed. Based on these numbers, the location which is over loaded, sends asset, from its managing list, such the two locations are equally loaded, the receiving location adds the asset to its managing list. Both locations updates their global lists locally, and the receiving location sends the updated global list to the server.

Distributed Viterbi Algorithm

The distributed algorithm, should work just as the non distributed algorithm, with the only exception that each peer should be time synchronised with the other peers, since the time stamps on the observations at one peer should fit the corresponding time at another peer. Each peer should know,

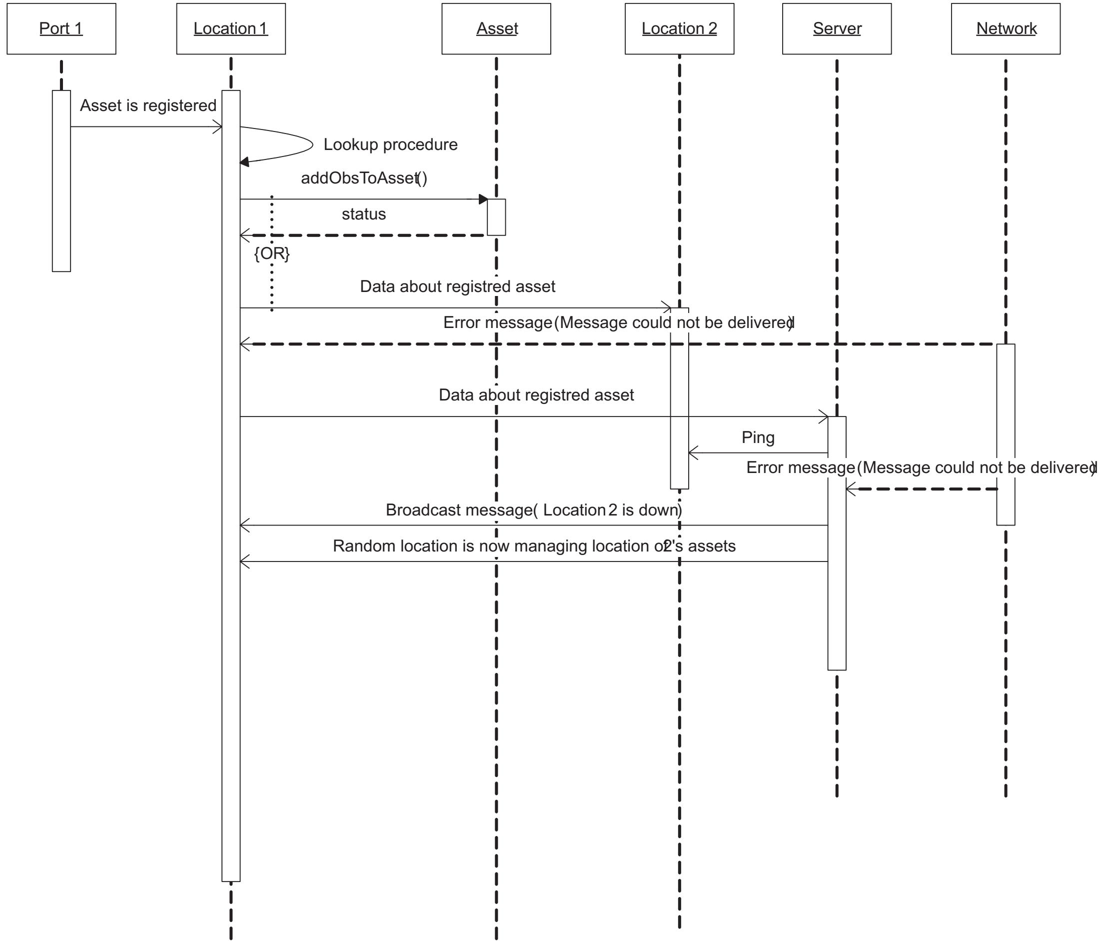


Figure 6.10: *Sequence diagram for faulty operation*

when an event has occurred somewhere in the system, in order to construct observation sequences with the same length for all assets in the system. A method to achieve this would be to occasionally to broadcast status messages containing information on which events has occurred at a location. Each asset has an array associated, this array holds the observation sequence. The length of the array is fixed to 48 slots per day, one slot for every 30 minutes. When an observation should be added to the sequence, the slot corresponding to the current time is used. The unused slots are zero padded, which ensures that the observation sequences all have the same length k.

$$O = o_1, \cdots, o_k \tag{6.1}$$

where k is derived from the equation

$$k = \left\lceil \frac{T_{reset}}{T_{slot}} \right\rceil \tag{6.2}$$

Where:

T_{reset} is seconds elapsed since last reset of the index counter k [s]

T_{slot} is the length of each time slot in seconds [s]

The normal Viterbi or the custom Viterbi algorithm is then run on the output sequence, and the optimal state sequence is calculated.

The software is now designed and is ready to be implemented.

6.3 Implementation of Distributed Algorithm

This section covers the implementation of the distributed Viterbi algorithm, and the constructed software needed in order to distribute the algorithm among multiple computers.

The software which has been described in the previous chapter, excluding various error handling routines has been implemented in C++, using Eclipse as IDE. The construction of the software follows the structure outlined in the software design. Only minor changes has been made, which can be seen in the updated class diagram, in Figure 6.11. A doxygen generated documentation can be found on the enclosed CD-ROM.

As the setup is intended as a proof of concept, and for test purposes of the algorithms, in a distributed way, various error handling has not been implemented, which means that no action is taken, in case of network failures in the system, and thus none of the error handling routines described in the analysis chapter has been implemented.

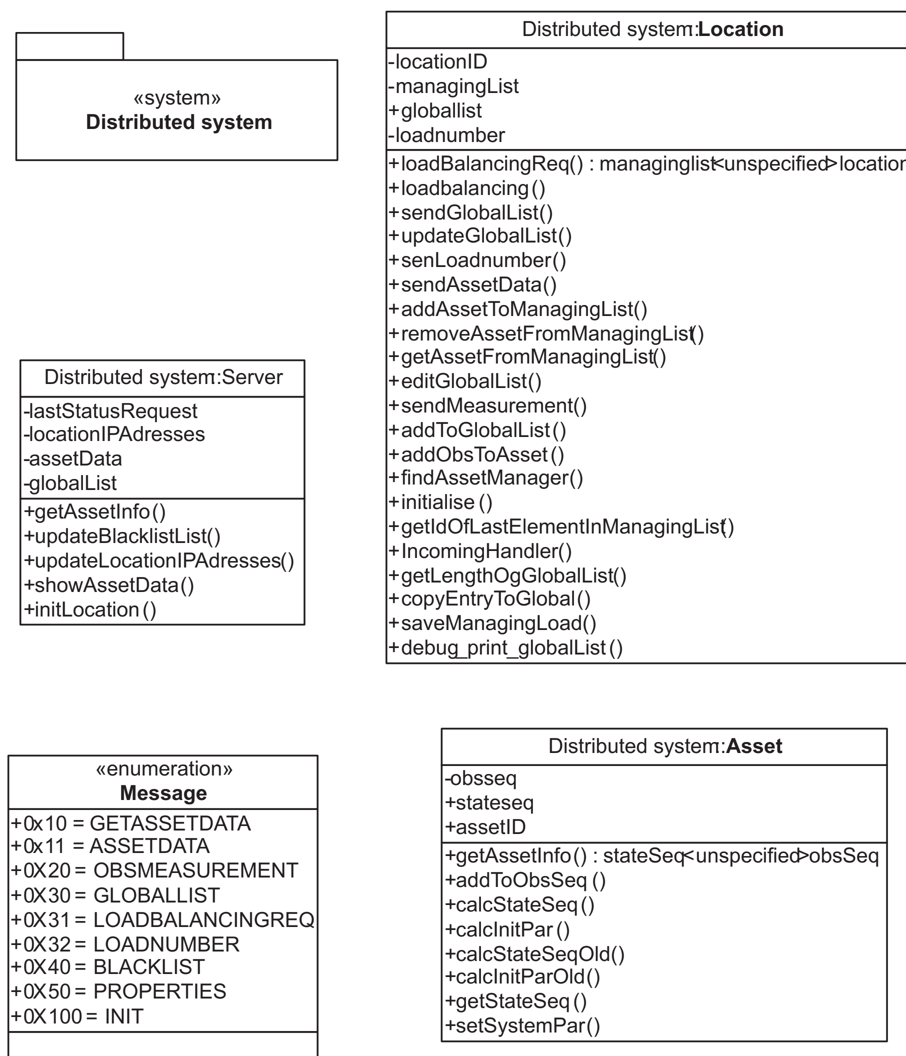


Figure 6.11: *Class diagram, of the implemented system*

The software needed for the algorithm to work has been implemented. The following sections, describe the operation, and tests of the software.

6.3.1 Description of Distributed Software

The software is intended as the peripheral software needed for the Viterbi algorithm to work correctly in a distributed setup. This includes handing data between the different clients, and inserting observations into the corresponding asset observation sequence. The two versions of the Viterbi algorithm has been ported from Matlab code, and into C++ code.

Location lists Two lists exists at each location, a managing list and a global list. The managing list is a linked list containing the assets which are currently managed by the location. The global list contains an entry for each asset in the system, the entry includes the asset id, the ip address of the managing location and a timestamps, which indicates the time of last edit of that particular entry. When assets are transferred to or from a client, the managing list and the global list are edited correspondingly.

Server

The server maintains a list of clients in the network. When a client sends an init command to the server, the server replies to the client with a location id, followed by the latest version of the global list. The first client which sends a init request, is treated differently, as the server does not have a global list at this time. The server hands assets to the first client, and adds a new entry in the global list for each asset. When all assets have been handed, the server has generated a global list. This list is sent to new connecting clients. When the user requests data, the server sends a command to all clients to prepare state sequences and return these to the server. The server combines all the received data, and present this to the user.

Clients

Each client initially sends an init request to the server, and receives a location id and a global list. When the RFID port generates a measurement, the client runs through its managing list, if it finds the asset id, the measurement is added to the corresponding asset. If the asset is not found in the managing list, the asset is searched for in the global list, and the measurement is sent to the managing location. When data is received from another location, the data is treated in the same way as if it was received from the port. When a measurement is received from another location a load balancing request is sent.

The load balancing request includes the number of assets currently managed by the location. When a load balancing request is received, the loadnumber in the request is compared to the local loadnumber. If the result of the comparison is greater than one, the calculated number of assets to be transferred is sent to the client which requested the load balancing along with a flag, indicating either to transfer assets or expect assets to be sent. The assets are transferred, the receiver of the assets sends the updated global list to the server and the load balancing is complete.

When a client receives an asset data request from the server, the Viterbi algorithm is run on all managed assets and the result is transferred to the server.

6.4 Test of Distributed Software

The system has been tested with multiple clients, these tests are described in this section. To be able to test the system, a test stub has been made, simulating port readings at each location. The test stub reads a line in a file indicating a timestamp, an asset and a location ID. If the location ID correspond to the location itself this reading is then sent to the location itself, through the loopback device. This implementation handles readings from the locations own port and from other locations equal. The stub uses simulated data which means that the data from the different assets are already synchronised.

6.4.1 Module Tests

Module tests have been performed on the load balancing routine and on the Viterbi algorithm. These two modules are the only ones tested since they use all the operands in the different classes.

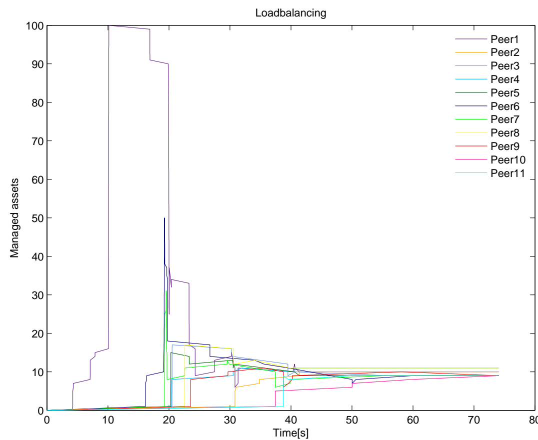


Figure 6.12: *Loadbalancing run with a sequence length of 100*

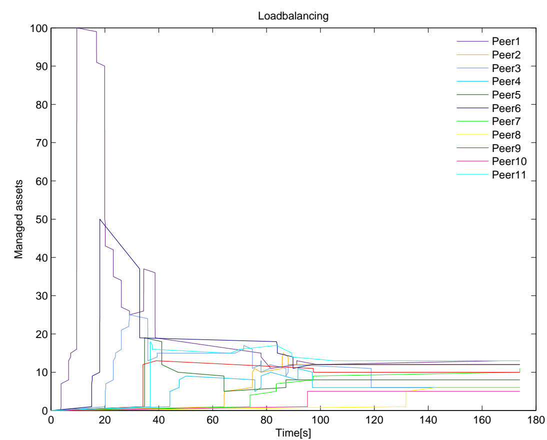


Figure 6.13: *Loadbalancing run with a sequence length of 20*

Test of Load Balancing Routines

The test of the load balancing routines, test the load balancing algorithm as well as the interaction with the network and asset routines. The aim of the test is to verify the load balancing routine, including the handing over of assets between the clients. The test is conducted with 11 clients, and a server. The complete test report is found in appendix J.

The results of the test is shown in Figure 6.12 and Figure 6.13. Figure 6.12 shows that the number of managed assets converges towards an equilibrium given enough interaction between the peers. The interactions needed is dependent on the number of assets and the number of locations in the system. As seen in Figure 6.12, which is made with a sequence length of 100, where the equilibrium is reached within 100 s for 100 assets and 11 peers. Figure 6.13 illustrates the same setup, with a sequence length of 20. In this case the load balancing did not reach the equilibrium before the peers reached the end of the measurement files. The conclusion is that the load balancing algorithm is able to balance the number of asset between the peers in the system given enough interaction between the peers.

Test of C++ Implementation of Viterbi Algorithms

The implementation of both the standard and custom Viterbi algorithm in C++ has been tested in terms of the error percentage in order to confirm that they behave similar to the implementations of the same algorithms in Matlab. Furthermore, the time consumption of both the algorithms has been tested in order to further establish the results from the Matlab implementations. These were that it is infeasible to use the standard Viterbi algorithm in the system and that it is a necessity that the custom algorithm is implemented on a distributed platform in order to make it usable in a system consisting of 4e6 assets and 20e3 locations as is the case with the distribution of cc containers in Europe. The full test report can be found in appendix I and the main results and conclusions will be summarised in this section.

Error Percentage The results of the test of the error percentage is shown in Figure 6.14 for the standard algorithm and in Figure 6.15 for the customised algorithm. When comparing with the corresponding plots of the error percentages of the Matlab implementations of the algorithms, which are shown in Figure 4.12 and Figure 4.13 on page 52, it can be seen that the C++ implementations of the algorithms performs exactly like the Matlab implementations. Based on this it is concluded that the implementation in C++ is correct.

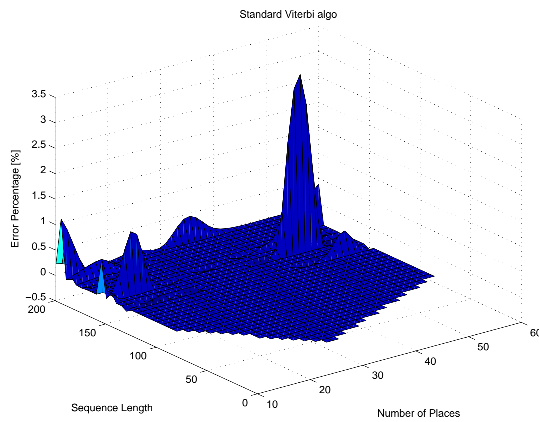


Figure 6.14: *Plot of estimation error percentage for the standard Viterbi algorithm in C++*

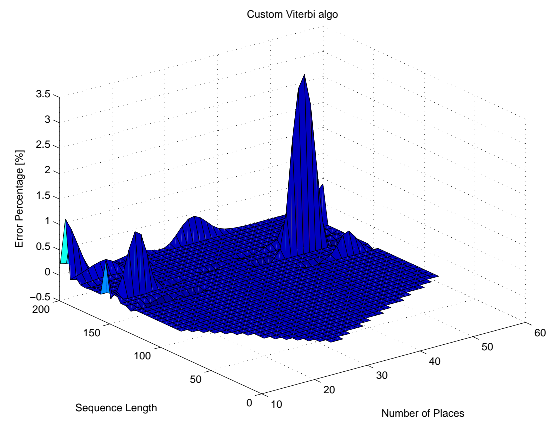


Figure 6.15: *Plot of estimation error percentage for the customised Viterbi algorithm in C++*

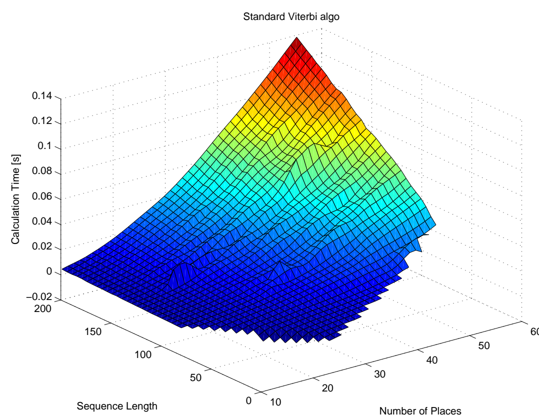


Figure 6.16: *Plot of calculation time for the standard Viterbi algorithm in C++*

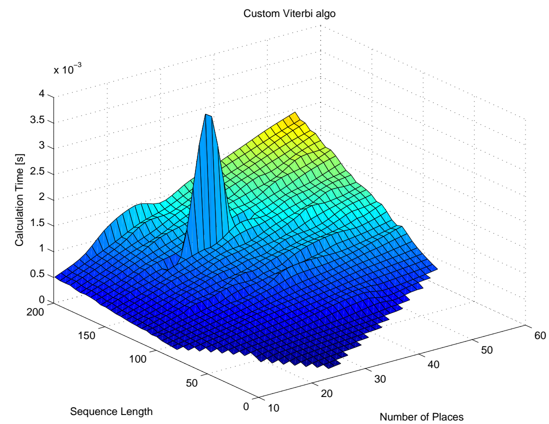


Figure 6.17: *Plot of calculation time for the customised Viterbi algorithm in C++*

Time Consumption The results of the time consumption test is shown in Figure 6.16 and Figure 6.17. It is apparent from Figure 6.16 and Figure 6.17 that the time consumption in the C++ implementation is lower for both the algorithms compared to the Matlab implementation, which can be seen when comparing with Figure 4.3 and Figure 4.14. The spike in Figure 6.17 is assumed to be caused by an outlier in the results, which may be caused by scheduling. The plots in Figure 6.18 and Figure 6.19 shows a number of cross sections of the plot in Figure 6.16, as it can be seen, the plots confirm that the computational complexity of the Viterbi algorithm is quadratic in the number of locations in the system and linear in the sequence length. In Figure 6.17 it is seen that the time consumption of custom algorithm is not constant in the number of locations. As described in section 4.2.2, the algorithm treats the states corresponding to the last non-empty measurement as special cases and calculates special probabilities for those states, all other states of the same type are assigned the same probabilities due to the symmetry of the system, thus it is expected that the time consumption of the algorithm will be constant in the number of places. The assignment of probabilities to states which produce the same probability, is done using for loops in the C++ implementation of the custom Viterbi algorithm, which becomes time consuming when the number of locations in the system increase, and because of this the time consumption is not constant.

The plots in Figure 6.20 and Figure 6.21 shows a number of cross sections of the plot in Figure 6.17, as it can be seen, the plots confirm that the computational complexity of the custom Viterbi algorithm is linear in the sequence length, but that it is also linear in the number of locations opposed to being constant as described in section 4.2.2.

The plot in Figure 6.22 shows the measured data along with a Matlab polyfit of the topmost line in

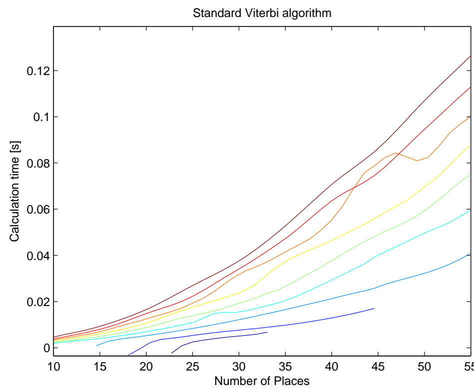


Figure 6.18: *Plot of cross sections at different sequence lengths of Figure 6.16*

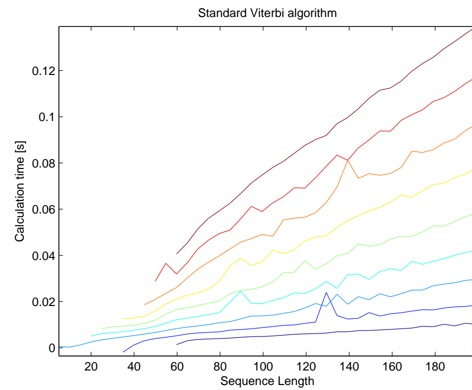


Figure 6.19: *Plot of cross sections at different numbers of locations of Figure 6.16*

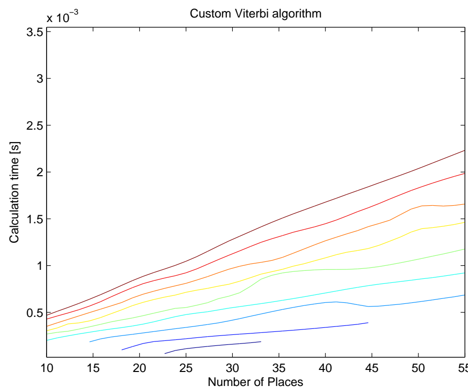


Figure 6.20: *Plot of cross sections at different sequence lengths of Figure 6.17*

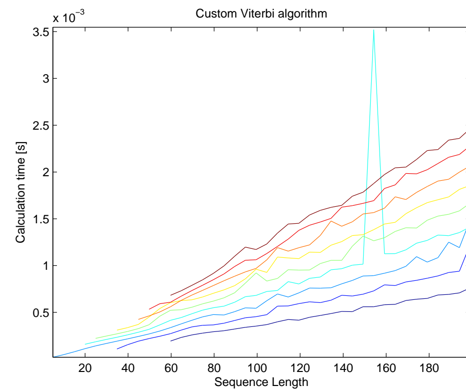


Figure 6.21: *Plot of cross sections at different numbers of locations of Figure 6.17*

Figure 6.18. The plot in Figure 6.23 shows the calculation time extrapolated to 20e3 locations using the polynomial found with polyfit. The value estimated at 20e3 locations is with a mean (μ) of 1.62e4 seconds and a standard variation (σ) of 320.26 seconds. Even though the C++ implementation of the Viterbi algorithm is faster than the Matlab implementation it is still infeasible for use in the system, which can be concluded from the plot of the extrapolation of the calculation time in Figure 6.23. Based on the extrapolation it will take between 1.97e3 and 2.13e3 ($\mu \pm 2\sigma$) years to calculate the state sequences for 4e6 assets when there is 20e3 locations in the system and the sequence length is 200. This can be reduced to between 36 to 39 days if it is assumed that the computational burden can be distributed to 20e3 computers.

The plot in Figure 6.24 shows the measured data along with a Matlab polyfit of the topmost line in Figure 6.20. The plot in Figure 6.25 shows the calculation time extrapolated to 20e3 locations using the polynomial found with polyfit. The value estimated at 20e3 locations is with a mean (μ) of 0.79 seconds and a standard variation (σ) of 0.004 seconds. From the extrapolation of the calculation time of the custom Viterbi algorithm to 20e3 locations, it can be calculated that it will take between 36 to 37 days for a single computer to calculate the state sequences of 4e6 assets if the sequence length is 200. This time can be reduced to between 156.4 to 159.6 seconds if the computational burden can be distributed to 20e3 computers. Since a sequence length of 200 corresponds four days if the time is split into intervals of half an hour, it is concluded that it is feasible to use a distributed version of the algorithm as state sequence estimator in the system.

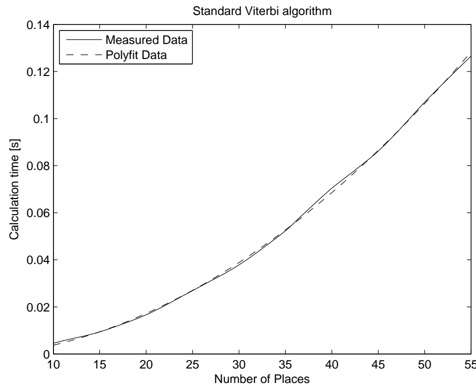


Figure 6.22: *Illustration of measured data along with the polyfit at a sequence length of 200*

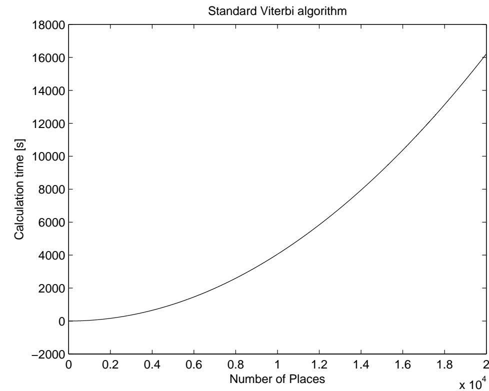


Figure 6.23: *Extrapolation of calculation time using polyfit at a sequence length of 200*

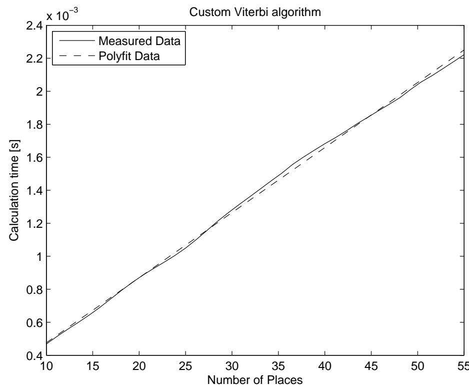


Figure 6.24: *Illustration of measured data along with the polyfit at a sequence length of 200*

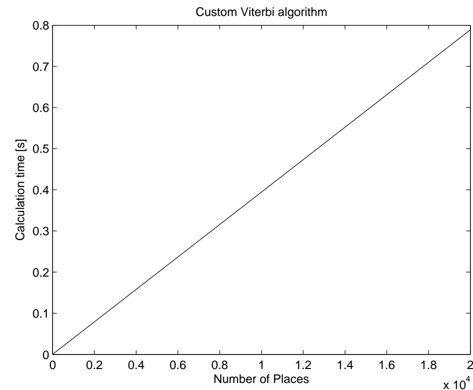


Figure 6.25: *Extrapolation of calculation time using polyfit at a sequence length of 200*

6.4.2 System Test

The complete system has been tested, using two different server modes; one where the state sequences are calculated on the peers, and one where each peer sends the observation sequences to the server, and the server calculates the state sequences. Due to network errors, in the transfer of the sequences, and due to time constraints it has not been possible to verify that the correct state sequences are transferred. It has been possible to verify by inspection that state sequences are calculated, and are correctly transferred if only one peer is present in the network. Due to these errors a comparison of the time consumption of the two server modes has not been conducted. Furthermore, the fact that all the peers run on the same two computers, means that the test will not show the correct result.

Discussion

When a distributed system should be implemented in a large scale system, a more appropriate startup routine should be implemented. In the current implementation, assets are initiated at the server, and sent to the first location, with state and observation sequences statically allocated. In order to reduce the data needed to be transferred, in the startup routine, only the essential data concerning the asset should be sent. When a measurement for the asset exists, the complete data structures could then be initiated on the peer, thus reducing the data needed to be transferred in the startup of the system.

Error handling routines should be added to the software, in order for the system to work as intended, this both includes network error handling and general error handling in the system. The general error handling routines includes handling of asset which are no longer physically present in the system.

Conclusion

Based on the test conducted on the distributed software it is concluded that it is possible to distribute the algorithm, with a reduction in the time consumption of the calculations. It has not been possible to actually test the time difference between running the Viterbi algorithm distributed at the peers, and centrally at the server. This has not been possible due to the fact that all 11 peers and the server is run on two machines only, thus the peers and server are running in different threads on the same processor. But as shown in section 4.2.3 the distributed algorithm is much faster, when the data are extrapolated to a full scale system. The load balancing routine has been tested, and shows that the number managed by each peer is converging against an equilibrium.

Chapter 7

Conclusion

The system in question is a closed system where assets circulate between multiple parties. The registration of the assets is error prone, as the registration is done with RFID technology. This work focuses on construction of a system which can correct the data, using a Viterbi algorithm. Two simulation models have been constructed, and attempted validated against test data from Post Danmark. The result of the validation is that neither of the two simulation models can be validated, by the used validation method.

The standard Viterbi algorithm is very time consumptive when many assets are included in the system, as the time consumption is quadratic in the number of assets. In a system with 4e6 assets and 20e3 locations, calculation of the estimated state sequences for a sequence length corresponding to approximately four days, would take approximately 265e3 years to calculate on a single computer, which is unusable for a billing system. A customised version of the Viterbi algorithm has been constructed, which is able to calculate state sequences with the same error percentage as the standard algorithm, using less computational time. Based on the error percentage, it is assumed that the algorithms produce the same state sequences. The customised algorithm uses approximately 39 days to calculate a state sequence of 200 for all assets on a single computer. In a distributed setup, where a computer is present on each location, it would take between 158 and 171 seconds to calculate the state sequences. This is well within the four days that a sequence length of 200 corresponds if a day is split into intervals of half an hour.

It has been shown that it is possible to distribute the algorithm between multiple peers in the system, and calculate the same state sequences, as when centrally calculated. The implemented load balancing algorithm has been verified to reach an equally distributed load of each peer, within 100 s for 11 peers. Due to networking problems it has not been possible to test all features in the distributed software, with all peers.

The results show a mean percentage of erroneous states in the estimated sequences of approximately 0.05 with a standard deviation of 1.89. The algorithm is thus not able to meet the requirement of a success rate of 98.75%, with the current input data, and the current method of testing, taking the relatively high standard deviation into account.

In order to meet all requirements in the specification a FDI system should be implemented in order to detect errors in the system, such as faulty RFID tags, or assets from third party suppliers. This part of the problem has not been investigated during the project work.

7.1 Future Work

The algorithms has not been able to meet the requirement of a success rate of 98.75%, with the current method of testing. The current method of testing compares each entry in the estimated state sequence and the correct state sequence divided with the total sequence length. As the system should be used for billing, a more valid method of testing would be to treat location states and the

corresponding transport state as one state, thus letting the participants pay for the time assets spend on transportation away from the given location, and then compare the number of timesteps the asset is estimated to be in each location, during a whole billing period. If this test show that the estimator still is not able to keep the requirements, a number of tests could be run, varying the quality of the input data thus varying the probability of successful readings, such a test would indicate what the quality of the input data should be in order to meet the requirement of 98.75%.

The memory usage, and computational time of the customised Viterbi algorithm could be further reduced by reducing the γ- and χ matrices to vectors as done with the transition probability matrix.

The computational time, clearly indicate that distributing the algorithm among multiple peers is further reducing the computational time needed for calculation of the state sequences for all assets in the system. This is of course a matter of system size if the additional complexity and bandwidth requirements are worth spending in order to reduce the computational time needed by the algorithm. The tests show that using the customised algorithm a number of ten processors would keep the computational time below the sequence length, such a computer would be possible to keep at a central place.

If the distributed algorithm should be used in a billing system additional features should be added to the system, where the main issue is various kinds of error handling, and improvements of the network communication.

Bibliography

Cassandras, Christos G and Lafortune, Stephane. Introduction to Discrete Event Systems. Springer, first edition, 1999. ISBN 0-7923-8609-4.

Jurafsky, Daniel and Martin, James H. Speech and Language Processing: An Introduction to Natural Language Processing Computational Linguistics and Speech Recognition. Prentice Hall, second edition, 2008. ISBN 0-13-187321-0.

Rabiner, Lawrence R. A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of The IEEE, 77(2):257–274, 1989.

Ross, Sheldon M. Introductory Statistics. Elsevier Academic Press, second edition, 2005. ISBN 0-12-597132-X.

The MathWorks, Inc. Getting Started with SimEvents. The MathWorks, Inc, 2007.

Appendix A

Rank-Sum Test

This section is based on (Ross, 2005). In order to test if two distributions are identical, the rank-sum test is used if the distributions are not normal. The distribution of the γ-values produced by the output sequences from the model and the system, can be seen in Figure A.1 and Figure A.2. When considering the distribution of the γ-values, it is evident that they are not normally distributed and thus the rank-sum test is appropriate when examining if the distributions are identical. The rank-sum

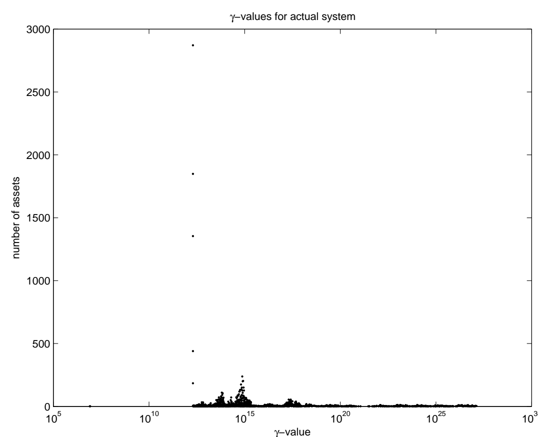


Figure A.1: *Distribution of γ-values for the output sequences from the actual system*

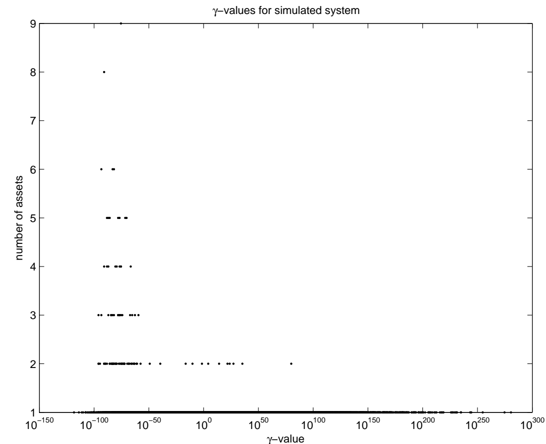


Figure A.2: *Distribution of γ-values for the output sequences from the simulated system*

test is used to test the hypothesis that two underlying population distributions are the same.
Suppose that independent samples of sizes n and m are drawn from the two populations. First the $n+m$ samples from the two distributions are ranked from the smallest value to the largest, this means that the smallest value is given rank 1, the second smallest rank 2 and so on. One of the sample vectors is designated as the first sample, it does not matter which sample vector. The test then makes use of the test statistic TS, which is defined as the sum of the ranks of the first sample.
The hypothesis H_0 to test, is that the two distributions are identical, and the value of the test statistic TS is assumed to be t. The objective is then to reject H_0 if the value t of the test statistic TS is either significantly large or significantly small, this means that the significance-level-α test will call for rejection of H_0 if either; $P[TS \leq t] \leq \alpha/2$ or $P[TS \geq t] \leq \alpha/2$. The probabilities are calculated under the assumption that the hypothesis H_0 is true, which means that the hypothesis will be rejected if the rank sum of the first sample is too small or too large to be explained by chance. The test will call for rejection of H_0 if the p-value of the data set is less than or equal to α. The p-value is given by:

$$p = 2\min(P[TS \leq t], P[TS \geq t]) \tag{A.1}$$

In practice, the significance level is not set in advance but the p-value is calculated and then a conclusion is drawn about the H_0 hypothesis. If the p-value is significantly small, then the hypothesis is rejected and if it is significantly large the hypothesis can be readily accepted. In order to calculate the probabilities in Equation (A.1) it is necessary to know the distribution of the test statistic TS when H_0 is true.
Suppose that the size of the first sample vector is n. When all $n+m$ values comes from the same distribution as it is assumed in H_0, it follows that the set of ranks in the first sample will have the same distribution as a random selection of n of the values $1, 2, \cdots, n+m$. This can be used to show that if H_0 is true, the mean $E[TS]$ and variance $Var[TS]$ of the test statistic TS are given by the following expressions:

$$E[TS] = \frac{n(n+m+1)}{2} \tag{A.2}$$

$$Var[TS] = \frac{nm(n+m+1)}{12} \tag{A.3}$$

It can be shown that when the sizes n and m of the sample vectors are larger than about 7, TS will be approximately normally distributed when H_0 is true, with mean and variance as given in Equation (A.2) and (A.3) respectively.
If there are any ties between the samples, the rank of the data should be the average of the ranks of all the data of the same value. For instance, if the first-sample data are; 2,4,4,6 and the second-sample data are; 5,6,7, then the ordered sample data are; 2,4,4,5,6,6,7. Thus, the value 4 has both rank 2 and 3 which gives an average rank of 2.5 for the value 4. Likewise, the average rank of the value 6 will be 5.5 since this value has both rank 5 and rank 6. Using this it can be calculated that the sum of ranks of the first-sample data will be; 1+2.5+2.5+5.5=11.5. Otherwise the test is run exactly as it would be if there were no ties.

Appendix B

Validation of Hidden Markov Model

This appendix describes the test conducted to validate the hidden Markov model of the assets in the distribution system.

B.1 Purpose

The purpose of the test is to validate the pure hidden Markov model of the system, either with a set of parameters calculated using the equations derived to find the system parameters, or with a set of parameters found using the forward-backward algorithm.

B.2 Theory

In order to validate if the hidden Markov model of the system uses the correct parameters a test has been carried out using the Viterbi algorithm. More specifically, the γ-values produced by the algorithm at the last time index has been used. The maximal γ-value at the last time index is proportional to the probability of the output sequence used as input to the algorithm, given the optimal state sequence and the underlying hidden Markov model. It is expected that the distribution of these γ-values will be similar if the model matches the actual system. The distributions of the γ-values are compared using the rank sum test.
The theory behind the hidden Markov model of the system can be found in section 2.2. The theory behind the derivation of the initial system parameters can be found in section 2.2.5. The theory behind the forward-backward algorithm can be found in section 2.2.4. The theory behind the rank sum test can be found in appendix A.
The initial parameters used in the hidden Markov model are found based on the data from the Post Danmark setup using the equations derived in section 2.2.5. First the number of time steps per day (F) has been found from dividing the sequence length of the measurements in the converted data file by the number of days the file covers:

$$\begin{aligned} F &= \frac{\text{number of timesteps}}{\text{number of days}} \Rightarrow \\ F &= \frac{1357}{34.5} \Rightarrow \\ F &\approx 39 \end{aligned} \tag{B.1}$$

Then the number of assets per transport M is estimated from the data file, by counting the number of assets making the same output in the same time step, adding an additional 1/19'th of this number since 5 % of the readings are missed in average, and taking the mean over all time steps.

Then the sum of the time fractions spent in transport and location states ($T_l^F + T_t^F$) is found using the proposition:

$$\begin{aligned} T_l^F + T_t^F &= \frac{N}{F \cdot M} \Rightarrow \\ T_l^F + T_t^F &= \frac{22257}{39 \cdot 56.8} \Rightarrow \\ T_l^F + T_t^F &\approx 10 \end{aligned} \tag{B.2}$$

Where:
N is the number of assets in the system

This is equally divided between T_l^F and T_t^F, this results in the following probabilities;

$$\begin{aligned} A &= \frac{5}{5+1} \\ A &= 0.8333 \end{aligned} \tag{B.3}$$

And equal for the B probability.

B.3 Setup

The test is run using a Matlab script implementation of the hidden Markov model of the asset automaton. A wrapper script has been constructed in order to simulate as many assets as there is present in the system. The output from the model is run through an altered version of the Viterbi algorithm, which outputs the maximum γ-value at the last time step. The data from the Post Danmark setup is run through a script in order to make the Post Danmark registrations comparable with the model output. The Post Danmark data is then run through the altered Viterbi algorithm as well. This gives two vectors with γ-values, which are then run through a rank-sum test. The rank-sum test then shows if it is likely that the γ-values produced by the two outputs are similarly distributed.

B.4 Equipment

The test is run on an Intel T2330@1.6GHz Core 2 Duo based laptop with 2GB system memory, running Mathworks Matlab 7.5.0.338 on Ubuntu 7.10.

B.5 Results

First the simulation model is tested using the parameters calculated based on the Post Danmark data directly. The distribution of the γ-values calculated using the actual system output and the simulation model output can be found in Figure B.1 and Figure B.2 respectively. As it can be seen in the figures, the distributions does not appear to be similar, and since the p-value from the rank-sum test is zero, the hypothesis of the distributions being similar can readily be rejected. Next the simulation model is tested using parameters estimated by the forward-backward algorithm, with the previous parameters as initial parameters. The distribution of the γ-values calculated using the actual system output and the simulation model output can be found in Figure B.3 and Figure B.4 respectively. As it can be seen in the figures, the distributions does not appear to be similar, and since the p-value from the rank-sum test is zero, the hypothesis of the distributions being similar can readily be rejected. Finally the simulation model is tested using parameters tuned by hand. The distribution of the γ-values calculated using the actual system output and the simulation model output can be found in Figure B.5 and Figure B.6 respectively. As it can be seen in the figures, the distributions does not appear to be similar, but since the p-value from the rank-sum test is 0.54, the hypothesis of the distributions being similar cannot be rejected.

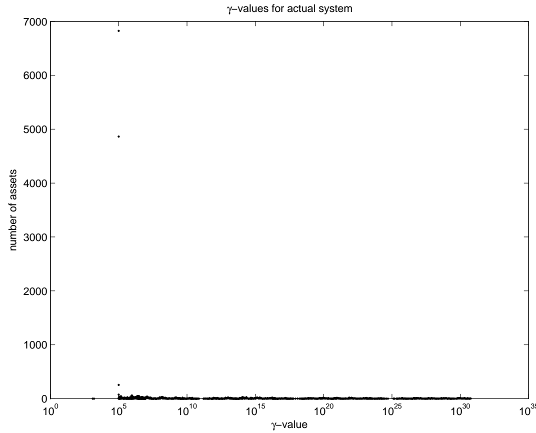


Figure B.1: *Distribution of γ-values of Post Danmark data ($A = 0.8333$,$B = A$)*

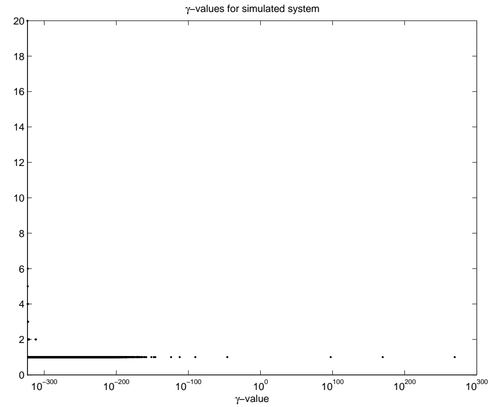


Figure B.2: *Distribution of γ-values of simulated data ($A = 0.8333$,$B = A$)*

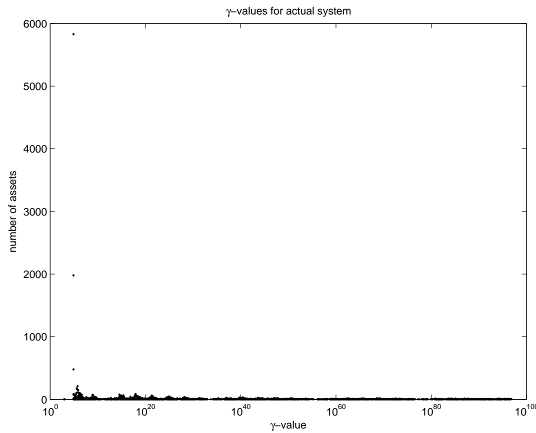


Figure B.3: *Distribution of γ-values of Post Danmark data ($A = 0.76$,$B = 0.85$)*

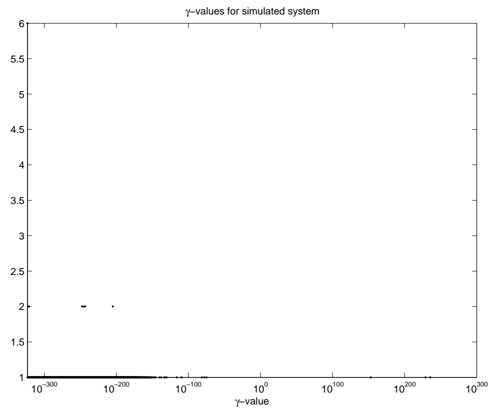


Figure B.4: *Distribution of γ-values of simulated data ($A = 0.76$,$B = 0.85$)*

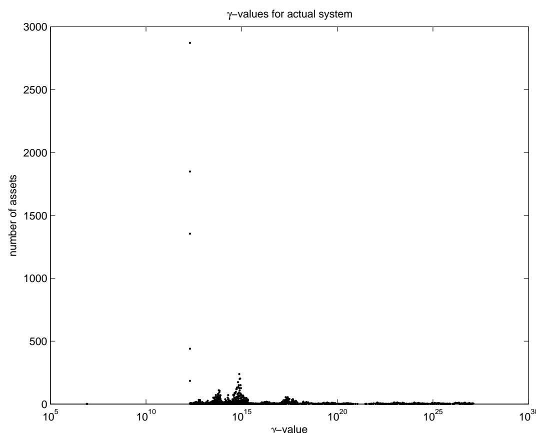


Figure B.5: *Distribution of γ-values of Post Danmark data ($A = N/(N+1)$,$B = A$), where n is the number of assets in the system*

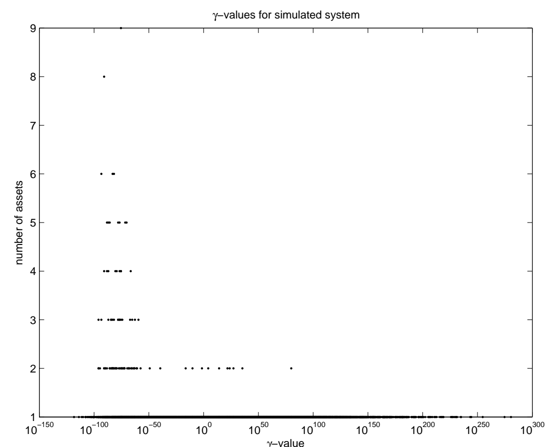


Figure B.6: *Distribution of γ-values of simulated data ($A = N/(N+1)$,$B = A$), where n is the number of assets in the system*

B.6 Discussion

Although the rank-sum test cannot reject the model parameters in the final test of the simulation model, the distributions of γ-values in Figure B.5 and B.6 does not appear to be similarly distributed. Based on the results from the different test it will though still be assessed that the latter parameters are those that fit the actual system best, and thus these are accepted as system parameters.

B.7 Conclusion

It has not been possible to find parameters for the model which result in a distribution plot for the model similar to the actual system. The parameters tested in the final test could not be rejected by the rank-sum test though. Because of this, these parameters will be accepted as the system parameters.

Appendix C

Validation of Simulink Model

This appendix deals with a validation of the simulation model, which is constructed in simulink.

C.1 Purpose

The purpose of the validation, is to determine, if the model is producing the same observation sequences, as the real distribution system.

C.2 Theory

The theory behind the Viterbi algorithm is found in chapter 4. The theory behind the rank sum test is found in appendix A. In order to validate the simulation model, outputs from both the model and the actual system has to be compared, it has been chosen to compare the distribution of the γ-values produced by the Viterbi algorithm, the γ-value is a measure of the probability that the output sequence origins from the model in question. Because the distribution of the γ-values are not known the rank-sum test is used. The rank-sum test is able to determine if two test vectors origins from populations with the same distributions.

C.3 Setup

The validation test is based on the result of a rank sum test. The rank sum test is run with a test vector consisting of the γ-values produced by the Viterbi algorithm, when run on data from the simulation model, and when run on data from the test setup respectively. The simulation is run with 1000 assets and eleven locations with a sequence length of 100. A data segment with similar parameters has been drawn from the test data from the actual system.

C.4 Results

The γ-values from the simulation model, and the test data are shown in Figure C.1 and C.2 respectively. The rank-sum test results in a p-value of zero.

C.5 Discussion

As the rank-sum test results in a p-value of zero, the H_0 hypothesis can be rejected. This can either mean that the simulation model is not applicable in modelling the system, or the test is simply too sensitive, which will mean that small modelling errors will result in rejection of the hypothesis.

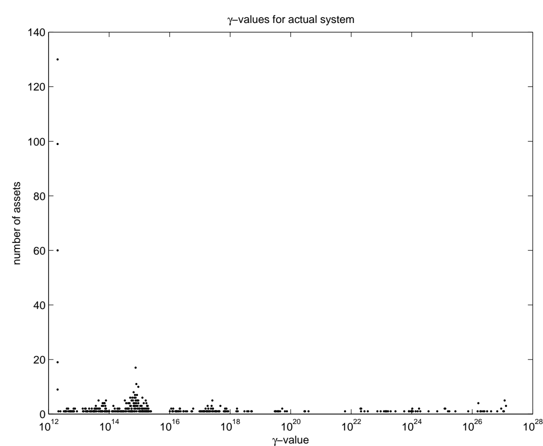


Figure C.1: *γ-values generated by the Viterby algorithm, run on data from Post Danmark*

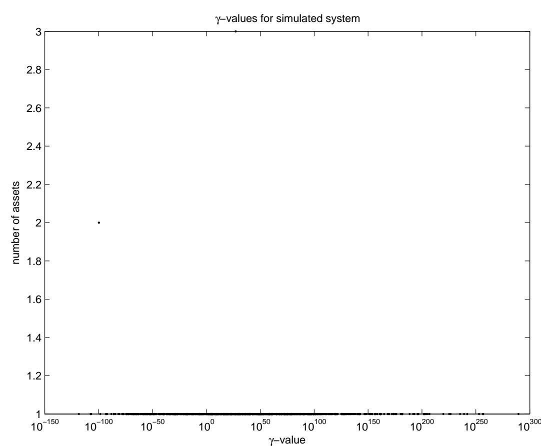


Figure C.2: *γ-values generated by the Viterby algorithm, run on data from the SimEvent simulation model*

C.6 Conclusion

The rank-sum test results in a p-value of zero, which indicate that the H_0 hypothesis can be rejected.

Appendix D

Simulink Model

This appendix includes all figures of the complete simulink model, which simulates the distribution system. The discription of the simulation model is found in chapter 3

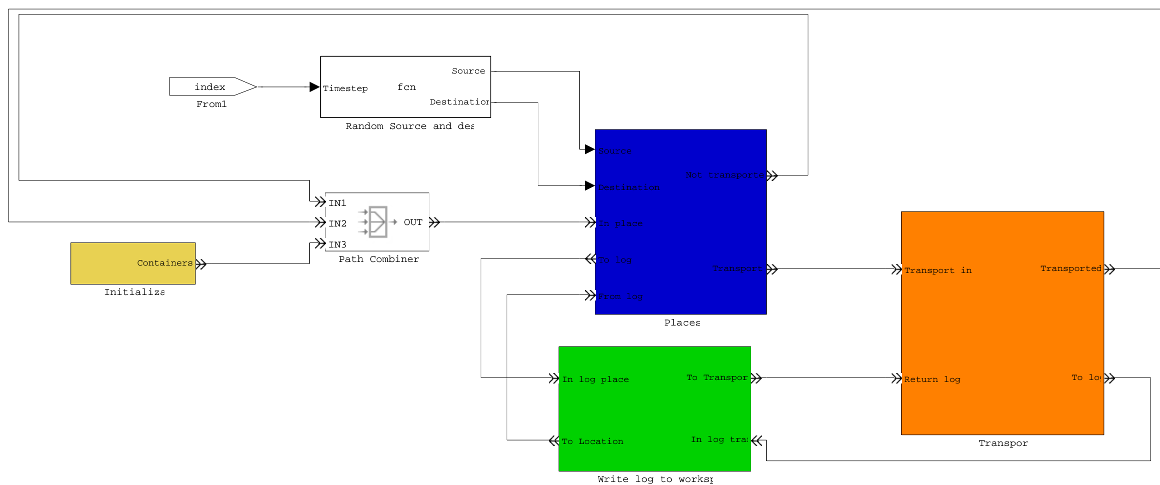


Figure D.1: *The complete simulink model*

distribution_system_autonom_fast/ Random Source and dest

```
1: function [Source,Destination] = fcn(locations, seed,Timestep)
2: % This block supports the Embedded MATLAB subset.
3: % See the help menu for details.
4: %rand('twister',seed);
5: persistent seeded;
6: if isempty(seeded)
7:     seeded = true;
8:     rand('twister',seed);
9: end
10: persistent step;
11: persistent S;
12: if isempty(S)
13:     S=1;
14: end
15: persistent D;
16: if isempty(D)
17:     D=1;
18: end
19:
20: if isempty(step)
21:     step = Timestep;
22: end
23: if (step ~= Timestep)
24:     step = Timestep;
25:     S = floor(1+locations.*rand(1,1));
26:     D = S;
27:         while(S==D)
28:         D = floor(1+locations.*rand(1,1));
29:         end
30:
31: Source = S;
32: Destination = D;
33:
34: else
35: Source = S;
36: Destination = D;
37:
38: end
39: %A=index;
```

Figure D.2: *Random source and destination*

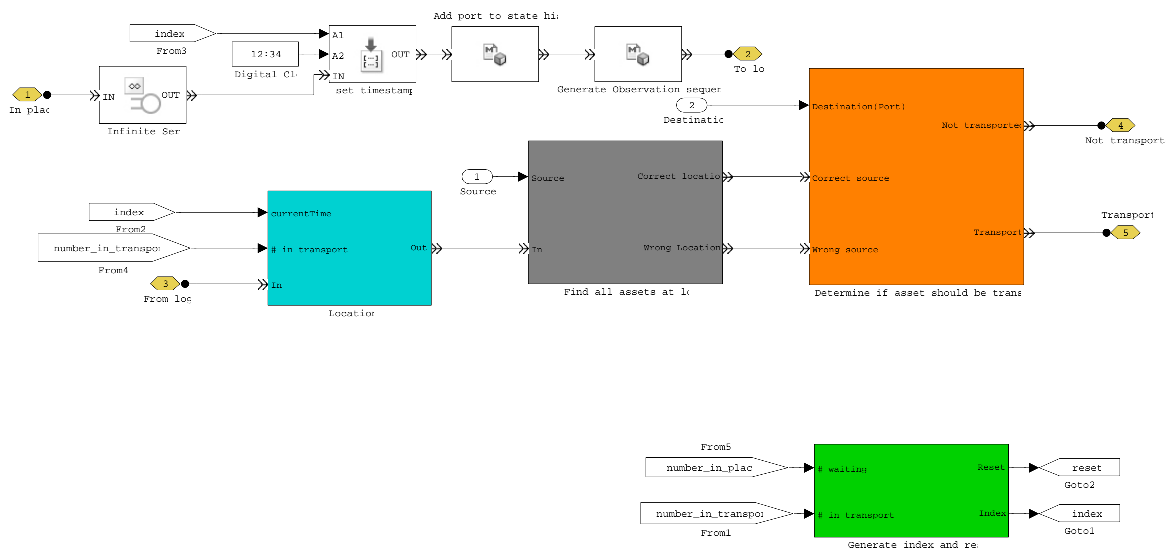


Figure D.3: *Places*

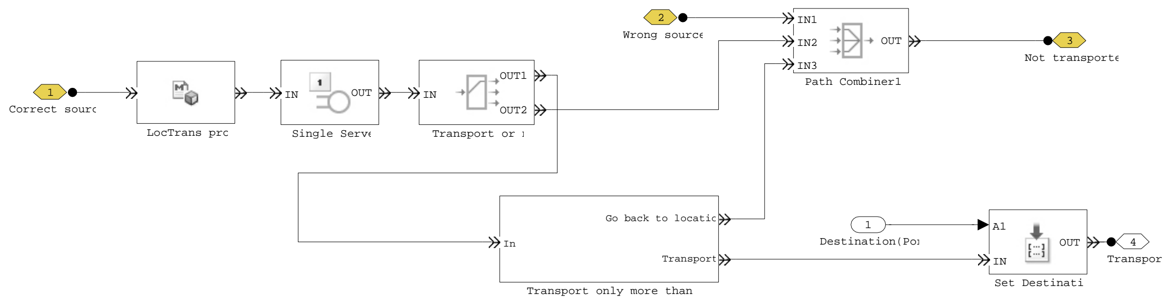


Figure D.4: *Determine if asset should be transported*

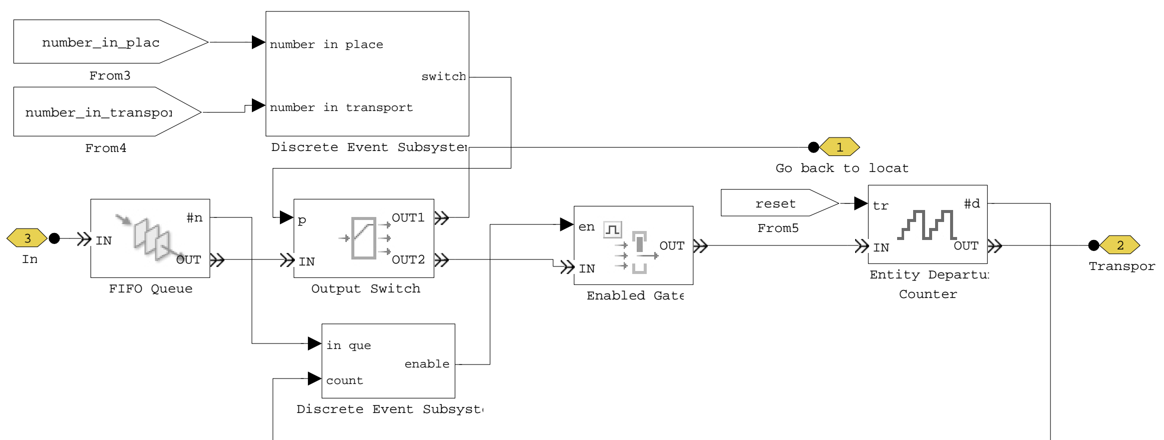


Figure D.5: *Transport only more than one*

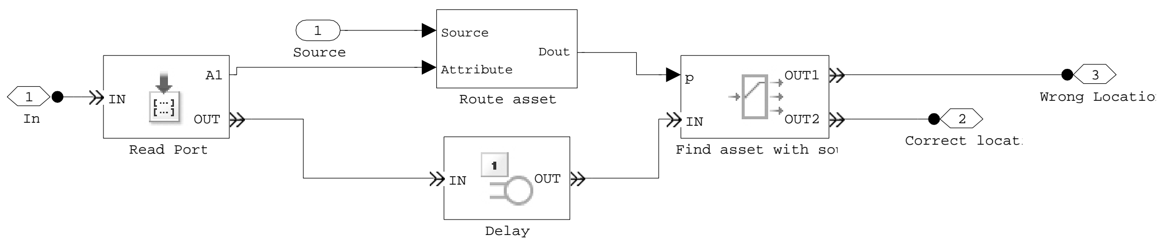


Figure D.6: *Find all assets at location*

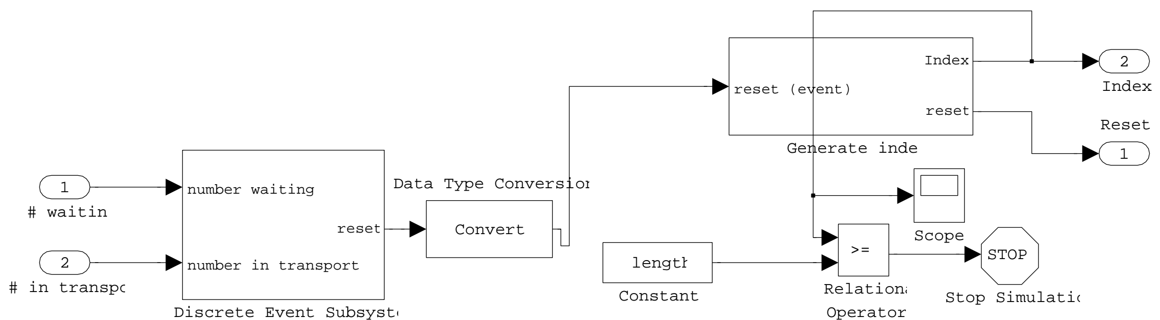


Figure D.7: *Generate index and reset*

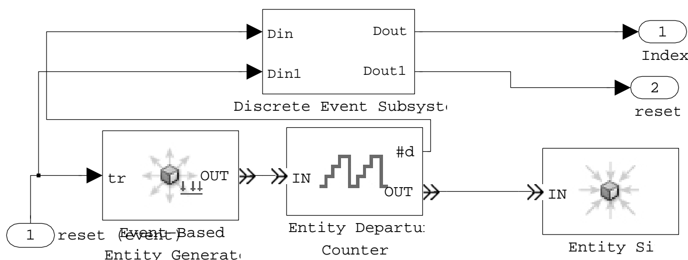


Figure D.8: *Generate index*

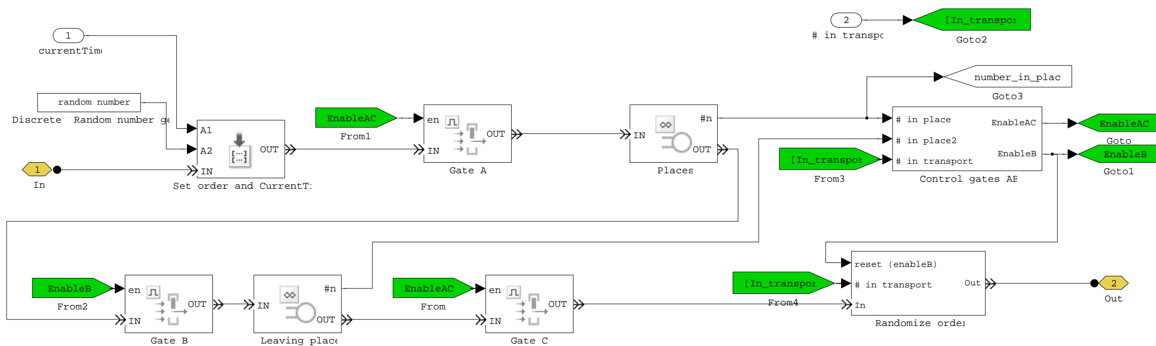


Figure D.9: *Location*

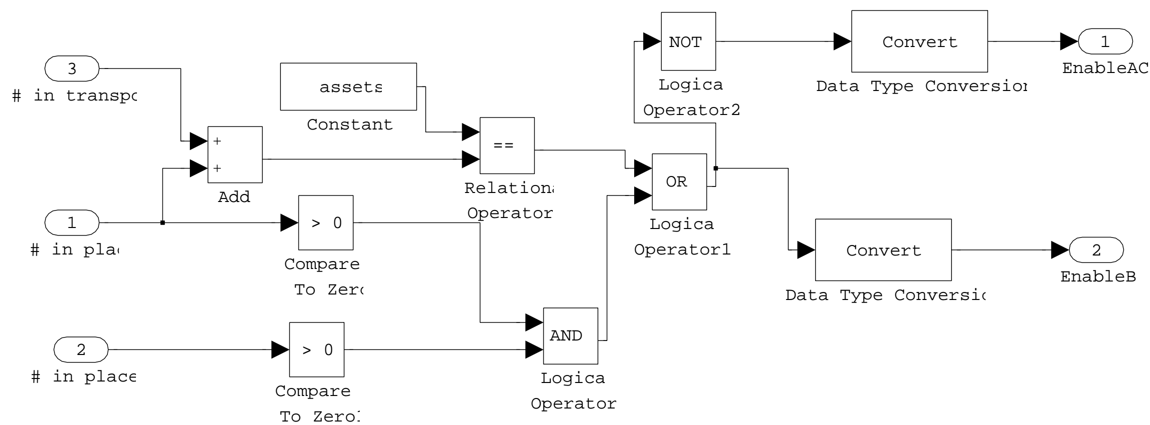


Figure D.10: *Control gates ABC*

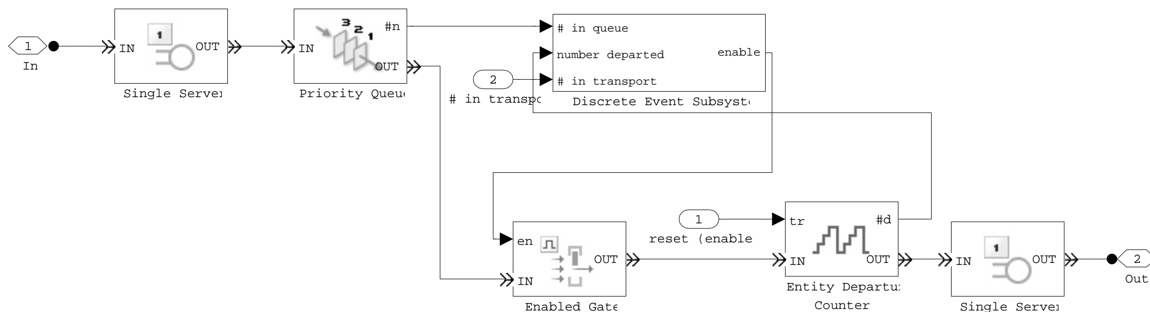


Figure D.11: *Randomize order*

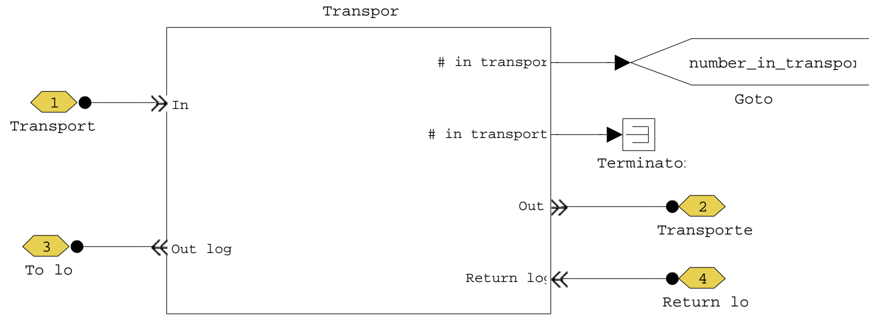


Figure D.12: *Transport*

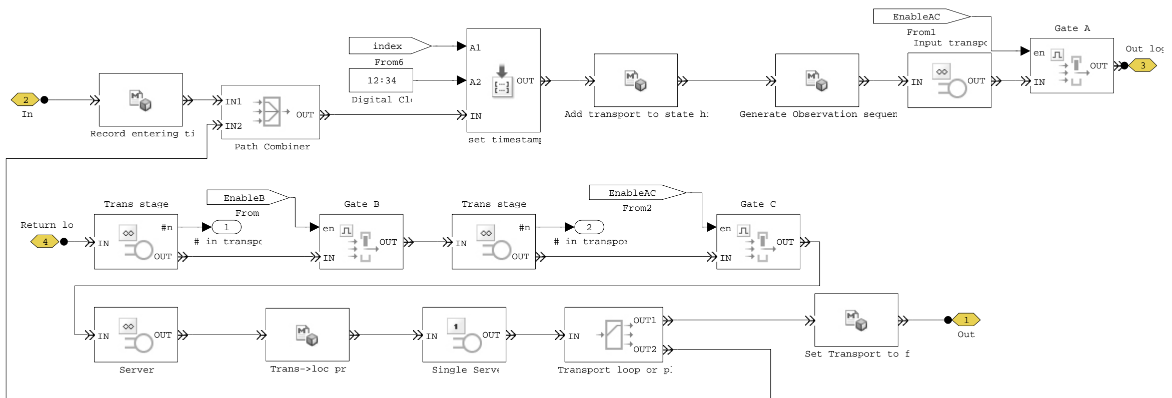


Figure D.13: *Transport*

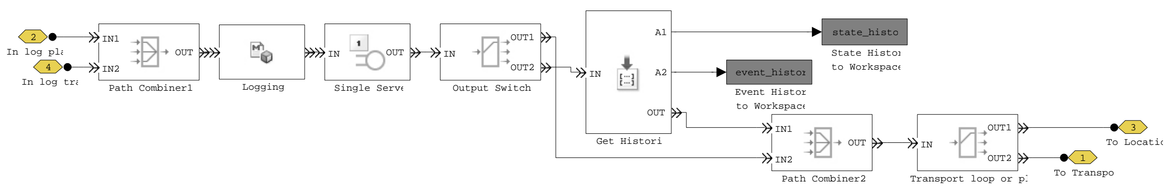


Figure D.14: *Write log to workspace*

Appendix E

Test of Viterbi Implementation

This appendix deals with a test of the Viterbi algorithm which is conducted in order to establish if the algorithm is correctly implemented.

E.1 Purpose

The purpose of the test is to establish if the Viterbi algorithm has been correctly implemented in Matlab.

E.2 Theory

The theory behind the state sequence estimator is found in section 4.1. The estimator consists of a Matlab function implementation of the Viterbi algorithm, which can be used to determine the most likely state sequence from the output of a hidden Markov process, such as the one used to describe the dynamics of the system. The A and B parameters used in the algorithm are varied, while the same parameters are kept fixed in the simulation of the system. The error percentages of the Viterbi algorithm are then measured at the different parameters. If the algorithm is correctly implemented, the algorithm which match the system parameters should be optimal regarding error percentages or at least at par with the best. The error percentage is measured by counting how many of the states in the state sequence is estimated wrong.

E.3 Setup

The test has been performed by generating simulated outputs from the pure hidden Markov model which generate outputs for each of the assets in the system which are uncorrelated with the other asset outputs. The simulation has been run on values of the A and B parameters in the Viterbi algorithm ranging from 0.05 to 0.95 with an interval of 0.05, which gives a total number of test points of 361. Five different tests has been run. In each of the tests the A and B parameters used by the hidden Markov model, are fixed. The values of the parameters used by the hidden Markov model are: $A = 0.25, B = 0.25$; $A = 0.25, B = 0.75$; $A = 0.50, B = 0.50$; $A = 0.75, B = 0.25$ and $A = 0.75, B = 0.75$ respectively. Each of the simulations are run with 100 assets in the system and with a sequence length of 100. The output from the tests are the mean error percentage taken over all assets.

E.4 Results

The plot in Figure E.1 shows the results from the test where the A and B parameters in the model are 0.25. As it can be seen in the figure, there is a large part of the surface where the error percentages are all lower than in the rest of the plot, and the Viterbi parameters that match the system parameters are placed in this part of the surface. The plot in Figure E.2 shows the results from the test where the

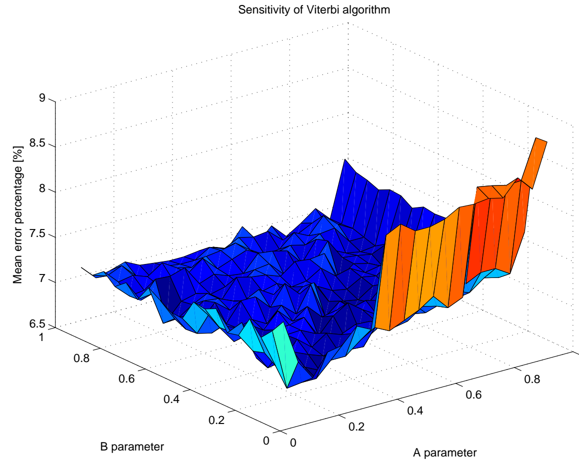


Figure E.1: *Error percentages at different A and B parameters of Viterbi algorithm at system parameters: $A = 0.25$ and $B = 0.25$*

A parameter in the model is 0.25 and B is 0.75. As it can be seen in the figure, there is a large part of the surface where the error percentages are all lower than in the rest of the plot, and the Viterbi parameters that match the system parameters are placed in this part of the surface.
The plot in Figure E.5 shows the results from the test where the A parameter in the model is 0.75 and B is 0.25. As it can be seen in the figure, there is a large part of the surface where the error percentages are all lower than in the rest of the plot, and the Viterbi parameters that match the system parameters are placed in this part of the surface. The plot in Figure E.4 shows the results

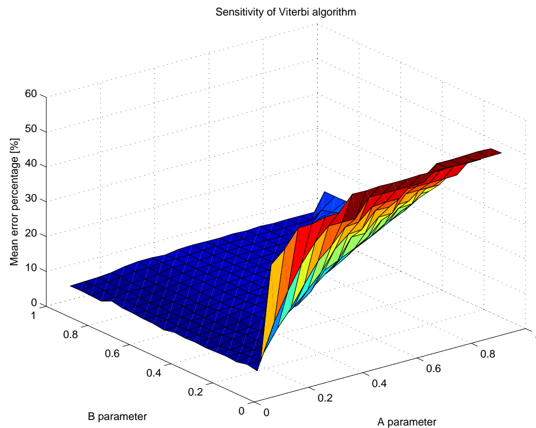


Figure E.2: *Error percentages at different A and B parameters of Viterbi algorithm at system parameters: $A = 0.25$ and $B = 0.75$*

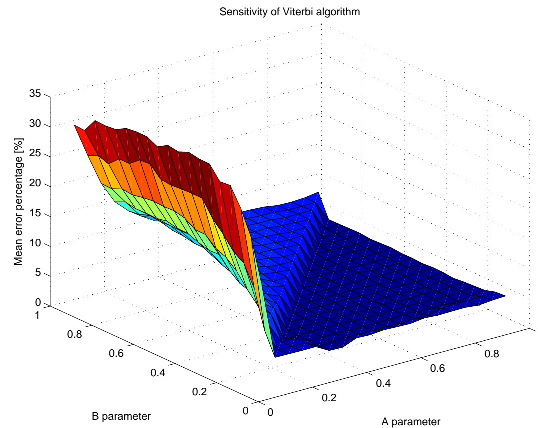


Figure E.3: *Error percentages at different A and B parameters of Viterbi algorithm at system parameters: $A = 0.75$ and $B = 0.25$*

from the test where the A and B parameters in the model are 0.50. As it can be seen in the figure, there is a large part of the surface where the error percentages are all lower than in the rest of the plot, and the Viterbi parameters that match the system parameters are placed in this part of the surface.
The plot in Figure E.5 shows the results from the test where the A and B parameters in the model are 0.75. As it can be seen in the figure, there is a large part of the surface where the error percentages are

all lower than in the rest of the plot, and the Viterbi parameters that match the system parameters are placed in this part of the surface.

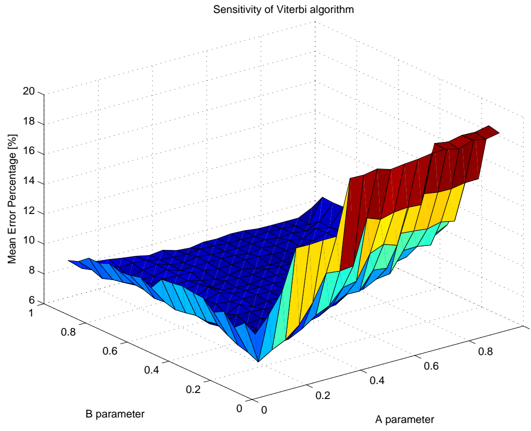


Figure E.4: *Error percentages at different A and B parameters of Viterbi algorithm at system parameters:* $A = 0.50$ *and* $B = 0.50$

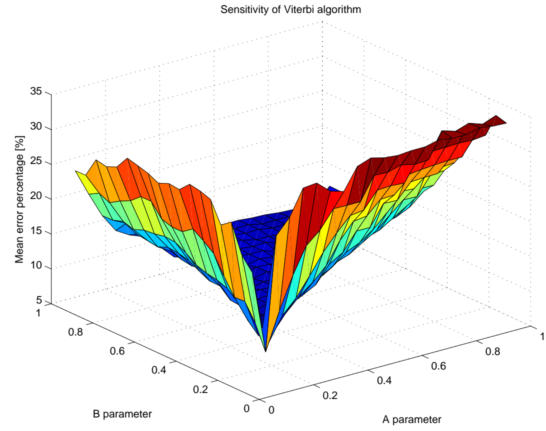


Figure E.5: *Error percentages at different A and B parameters of Viterbi algorithm at system parameters:* $A = 0.75$ *and* $B = 0.75$

E.5 Discussion

From the results shown in the plots it is determined that the implementation of the Viterbi algorithm is correct since the algorithm with the parameters which match the system parameters in all cases are at least at par with the algorithm with the best performance.

E.6 Conclusion

It is concluded that the implementation of the Viterbi algorithm is correct.

Appendix F

Test of Time Consumption of Viterbi Algorithm

This chapter deals with a test of the time consumption of the implementation of the Viterbi algorithm used to estimate state sequences of hidden Markov models.

F.1 Purpose

The purpose of the test is to assess if it is possible to use the Viterbi algorithm on a system which is in the scale of the distribution system in question. The distribution system of cc's throughout Europe consist of approximately 4e6 assets and 20e3 locations.

F.2 Theory

The theory behind the state sequence estimator is found in section 4.1. The estimator consists of a Matlab function implementation of the Viterbi algorithm, which can be used to determine the most likely state sequence from the output from a hidden Markov process, such as the one used to describe the dynamics of the system. As described in section 4.1 the calculation complexity of the Viterbi algorithm is $\mathcal{O}(n^2k)$, where n is the number of states in the system and k is the length of the output sequence.

F.3 Setup

The test has been conducted on the output from the SimEvent simulation model, such that it has been possible to verify the estimated state sequence against the actual sequence. In order to estimate the state sequence for the assets, the Viterbi algorithm have been used. The test has been performed on simulated data, from different simulations using different numbers of locations in the simulations. Different sequence lengths in the estimations are obtained by truncating the output sequences at the last non-empty output. This is done in order to verify the calculation complexity of the algorithm. The system parameters used in the Viterbi algorithm has been the same as those used in the simulation of the system. The estimator used in the test is a Matlab function implementation and the calculation time for the algorithm is calculated using the tic/toc function in Matlab.

F.4 Equipment

The test is run on an Intel T2330@1.6GHz Core 2 Duo based laptop with 2GB system memory, running Mathworks Matlab 7.5.0.338 on Ubuntu 7.10.

F.5 Results

The results of the test is shown in Figure F.1 which shows the calculation time for the algorithm. As it can be seen in Figure F.1 the calculation time for the Viterbi algorithm is linear in the sequence length and quadratic in the number of locations in the system, this can be seen more clearly in the cross section plots shown in Figure F.2 and Figure F.3, which shows a number of cross sections of the plot in Figure F.1. The plot in Figure F.4 shows the measured data along with a Matlab polyfit

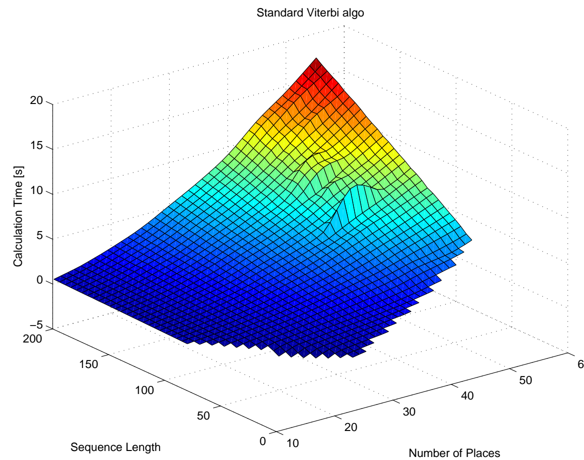


Figure F.1: *Plot of calculation time for the Viterbi algorithm*

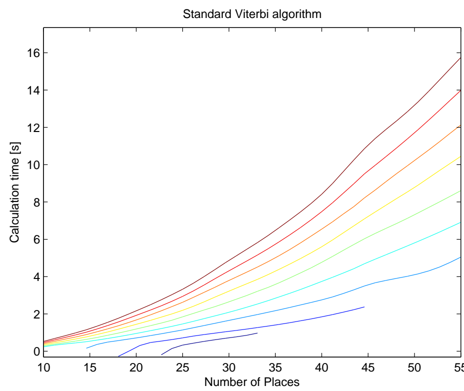


Figure F.2: *Plot of cross sections at different sequence lengths of Figure F.1*

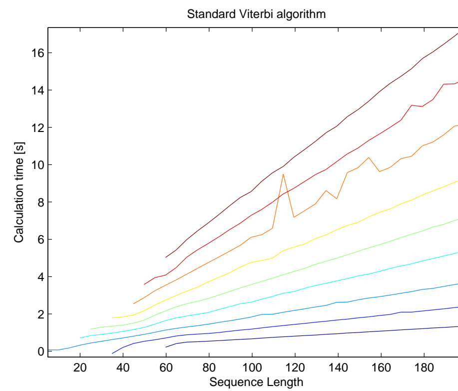


Figure F.3: *Plot of cross sections at different numbers of locations of Figure F.1*

of the topmost line in Figure F.2. The plot in Figure F.5 shows the calculation time extrapolated to 20e3 locations using the polynomial found with polyfit. The value estimated at 20e3 locations is with a mean (μ) of 2.01e6 seconds and a standard variation (σ) of 3.88e4 seconds.

F.6 Discussion

From Figure F.5 it is estimated that it will take the algorithm between 1.93e6 and 2.09e6 ($\mu \pm 2\sigma$) seconds to calculate the state sequence of a single asset at a 2σ level of confidence if there is 20e3 locations in the system and the sequence length is 200. If this number is scaled to a system consisting of 4e6 assets it would take between 245e3 and 265e3 years to calculate the state sequence of all assets on a single computer. If it is assumed that a computer is present at each location in the system, such that the computational burden can be distributed to each of these, it would take between 12 and 13

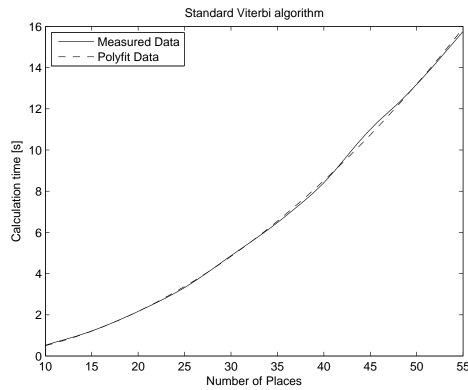


Figure F.4: *Illustration of measured data along with the polyfit at a sequence length of 200*

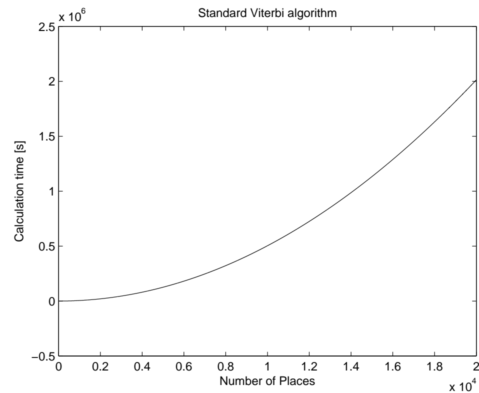


Figure F.5: *Extrapolation of calculation time using polyfit at a sequence length of 200*

years to calculate the state sequences. This is infeasible to be useful since a sequence length of 200 corresponds to a time span of just above four days if a day is split into intervals of half an hour. If the Viterbi algorithm is to be used for model based estimation of state sequences in a system of this scale it has to be simplified in order to reduce its calculation complexity.

F.7 Conclusion

From the test of the time complexity of the Viterbi algorithm it is concluded that the algorithm at its current state is not feasible for use as estimator in a system in the scale of the distribution system in question. Therefor, an approach is needed to decrease the computational complexity of the algorithm.

Appendix G

Test of Error Rate of Viterbi Algorithm

This chapter describes the test of the error rate of the designed state sequence estimator, which is used to determine the state sequences of the individual assets in the system.

G.1 Purpose

The purpose of the test is to determine how often the state sequence estimator is able to find the correct state sequence for an asset automaton, given the output from the automaton in interest. The result from the test will be the percentage of states which are estimated faulty from the entire state sequence. Furthermore, the test is performed on both the standard Viterbi algorithm as well as the customised Viterbi algorithm on the same output sequences in order to confirm that the customised algorithm produce the same errors as the standard algorithm, thus indicating that the estimated state sequences from the two algorithms are the same.

G.2 Theory

The theory behind the state sequence estimator is found in section 4.1 and section 2.2.2. The estimator consists of a Matlab function implementation of the Viterbi algorithm, which can be used to determine the most likely state sequence from the output from a hidden Markov process, such as the one used to describe the dynamics of the system.

G.3 Setup

The test has been conducted on output from the Simulink simulation model, such that it has been possible to verify the estimated state sequence against the actual sequence. In order to estimate the state sequence for the assets, both Viterbi algorithms have been used. The test has been performed on simulated data, from different simulations using different numbers of locations in the simulations. Different sequence lengths in the estimations are obtained by truncating the output sequences at the last non-empty output. This is done in order to verify that the estimation error does not depend on these parameters. The system parameters used in the Viterbi algorithm has been the same as those used in the simulation of the system.

G.4 Equipment

The test is run on an Intel T2330@1.6GHz Core 2 Duo based laptop with 2GB system memory, running Mathworks Matlab 7.5.0.338 on Ubuntu 7.10.

G.5 Results

The results from the test is shown in Figure G.1 and Figure G.2, which show the results from the standard Viterbi algorithm and the customised Viterbi algorithm respectively.

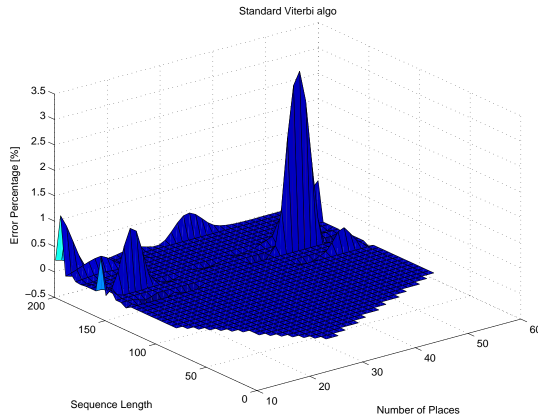


Figure G.1: *Plot of estimation error percentage for the standard Viterbi algorithm*

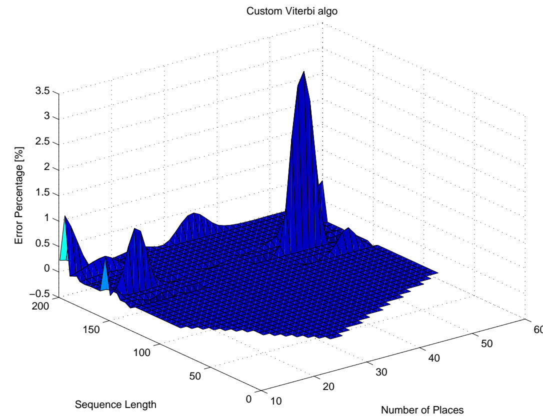


Figure G.2: *Plot of estimation error percentage for the customised Viterbi algorithm*

G.6 Discussion

As it can be seen in the two figures the two different algorithms produce exactly the same results on the same output sequences which is a strong indication that the customised algorithm produces the same estimations as the standard algorithm for the system in question. The results show a mean percentage of erroneous states in the estimated sequences of approximately 0.05 with a standard deviation of 1.89. These results indicate that the estimator currently is able to fulfil its requirement when considering the mean value. But taking the relatively high standard deviation into account it is concluded that it is not able to fulfil its requirement.

G.7 Conclusion

From the results plotted in Figure G.1 and G.2 it is concluded that the two algorithms has the same error percentage profiles which could indicate that the two algorithms produce the same sequence estimations given the same observation sequences, and using the system model in question. This means that the customised may be used as a substitute for the standard Viterbi algorithm, when the system is symmetric in its parameters as is the case for the distribution system in question. It is furthermore concluded that the estimator in its current form cannot fulfil the requirement of estimating correctly in 98.75 % of the cases.

Appendix H

Test of Time Consumption of Custom Viterbi Algorithm

This chapter deals with a test of the time consumption of the two Viterbi algorithms used in the estimation of the state sequences of the assets in the system.

H.1 Purpose

The purpose of the test is to confirm that the custom Viterbi algorithm designed for symmetric hidden Markov models as the one modelling the distribution system in question has the time consumption as expected. Furthermore, the test will show the ratio in calculation time between the standard Viterbi algorithm and the custom one.

H.2 Theory

The theory behind the state sequence estimator is found in section 4.1 and section 2.2.2. The estimator consists of a Matlab function implementation of the Viterbi algorithm, which can be used to determine the most likely state sequence from the output from a hidden Markov process, such as the one used to describe the dynamics of the system. As described in section 4.1 the calculation complexity of the standard Viterbi algorithm is $\mathcal{O}(n^2k)$, where n is the number of states in the system and k is the length of the output sequence. The customised Viterbi algorithm is expected to have a calculation complexity of $\mathcal{O}(k)$.

H.3 Setup

The test has been run on the data set produced by the Simulink simulation model, which also has been used in the test of the error percentage of the estimator described in appendix G. The simulation has been run using different number of locations in the system. Furthermore, the outputs from the individual assets in the simulated data has been truncated at the last non-empty output in order to obtain calculation times for different sequence lengths. The estimators used in the tests are Matlab function implementations and the calculation time for the algorithms is calculated using the tic/toc function in Matlab.

H.4 Equipment

The test is run on an Intel T2330@1.6GHz Core 2 Duo based laptop with 2GB system memory, running Mathworks Matlab 7.5.0.338 on Ubuntu 7.10.

H.5 Results

The results of the test is shown in Figure H.1 and Figure H.2 which shows the calculation time for the standard and custom algorithm respectively. Figure H.1 is a repetition of Figure F.1 on page 106. Figure H.2 shows that the custom algorithm is linear in the sequence length but not entirely constant in the number of locations in the system. This can be seen more clearly in the cross section plots in Figure H.3 and Figure H.4. The ratio in calculation time of the two algorithms can be seen in

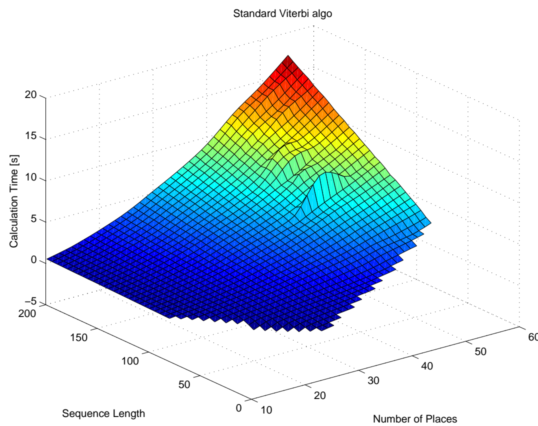


Figure H.1: *Plot of calculation time for the standard Viterbi algorithm*

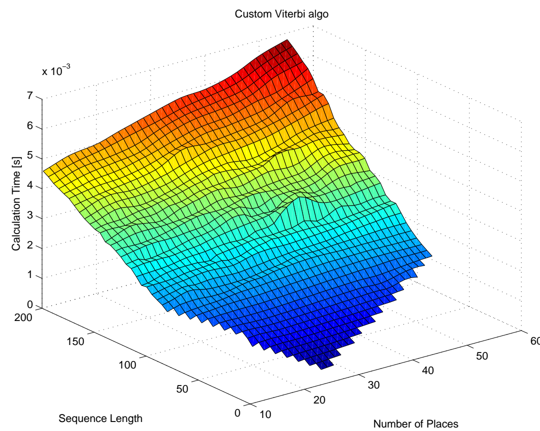


Figure H.2: *Plot of calculation time for the customised Viterbi algorithm*

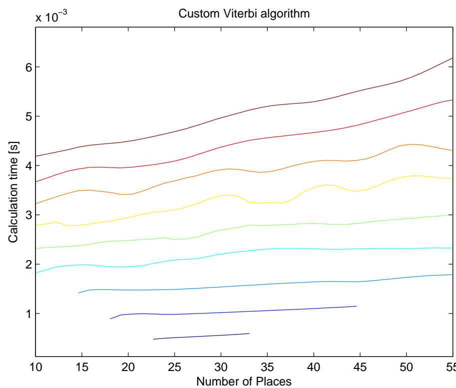


Figure H.3: *Plot of cross sections at different sequence lengths of Figure H.2*

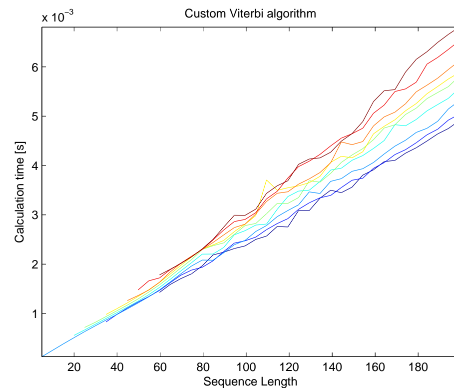


Figure H.4: *Plot of cross sections at different numbers of locations of Figure H.2*

Figure H.5. As it is apparent from the figure the ratio in calculation time grows with the number of locations in the system as expected. The plot in Figure H.6 shows the measured data along with a Matlab polyfit of the topmost line in Figure H.3. The plot in Figure H.7 shows the calculation time extrapolated to 20e3 locations using the polynomial found with polyfit. The value estimated at 20e3 locations is with a mean (μ) of 0.822 seconds and a standard variation (σ) of 0.016 seconds.

H.6 Discussion

The reason for the calculation time of the customised Viterbi algorithm not being entirely constant in the number of locations in the system is expected to be caused by the choice of implementation. As described in section 2.2.2, the algorithm treats the states corresponding to the last non-empty measurement as special cases and calculates special probabilities for those states, all other states of

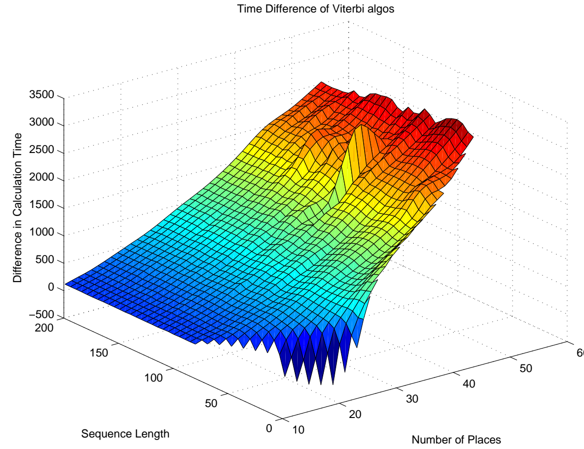


Figure H.5: *Plot of ratio in calculation time between the two algorithms*

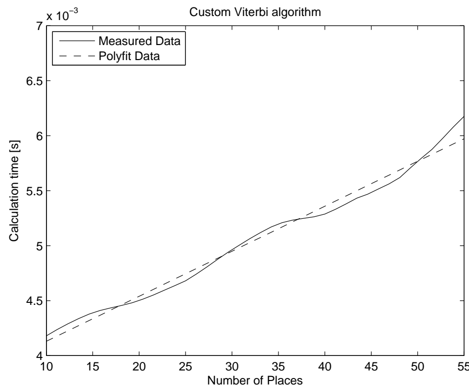


Figure H.6: *Illustration of measured data along with the polyfit at a sequence length of 200*

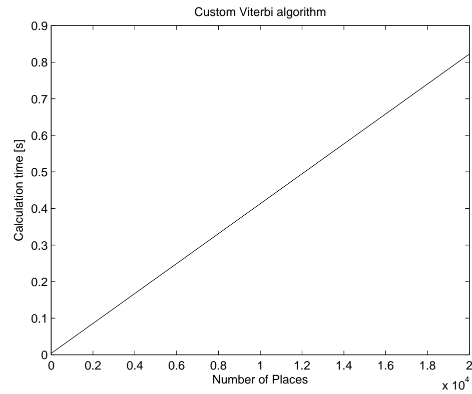


Figure H.7: *Extrapolation of calculation time using polyfit at a sequence length of 200*

the same type are assigned the same probabilities due to the symmetry of the system. Because of this the algorithm is supposed to have complexity $\mathcal{O}(k)$. In the implementation however the latter probabilities are first calculated and then assigned to the states using array assignments in Matlab. The time consumption of the array assignment is expected to be proportional to the number of states and thus the number of locations in the system, and because of this the time consumption of the algorithm is not constant in the number of locations in the system.
From Figure H.7 it is estimated that it will take the algorithm between 0.790 and 0.854 ($\mu \pm 2\sigma$) seconds to calculate the state sequence of a single asset at a 2σ level of confidence if there is 20e3 locations in the system and the sequence length is 200. If this number is scaled to a system consisting of 4e6 assets it would take between 36.5 and 39.5 days to calculate the state sequence of all assets on a single computer. If it is assumed that a computer is present at each location in the system, such that the computational burden can be distributed to each of these, it would take between 158 and 171 seconds to calculate the state sequences. This is well within the four days that a sequence length of 200 corresponds if a day is split into intervals of half an hour.

H.7 Conclusion

From the extrapolation of the test data to a system in the scale of the distribution system in question, it is concluded that it is feasible to use the customised version of the Viterbi algorithm as state sequence estimator. This is under the assumption that it will be possible to distribute the computational burden to a number of computers corresponding the number of participants in the system.

Appendix I

Test of C++ Implementation of Viterbi Algorithm

This chapter deals with the test of the C++ implementation of both the standard and the customised Viterbi algorithm. The two algorithms are tested regarding both the error rate of the algorithm as well as the time consumption.

I.1 Purpose

The C++ implementation of the algorithms is tested in order to confirm that they are implemented properly and performs similar to the Matlab implementations. Furthermore, the time consumption of the algorithms is measured and the results are compared to the Matlab implementations.

I.2 Theory

The theory behind the state sequence estimator is found in section 4.1 and section 2.2.2. The estimator consists of a C++ implementation of the Viterbi algorithm, which calculates the most likely state sequence.

I.3 Setup

The test of the two algorithms is run on the same simulated data set as the tests of the estimation error and time consumption of the Matlab implementations, which is described in appendix G and appendix H. The data are produced by the SimEvent model of the system, and the output sequences are truncated at the last non-empty output in order to obtain different sequence lengths.

I.4 Equipment

The test is run on an Intel T2330@1.6GHz Core 2 Duo based laptop with 2GB system memory, running Ubuntu 7.10.

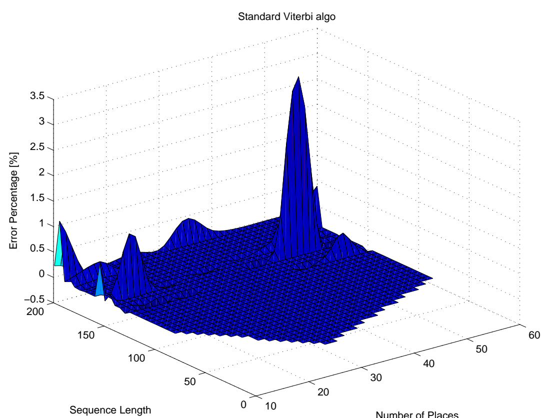


Figure I.1: *Plot of estimation error percentage for the standard Viterbi algorithm in C++*

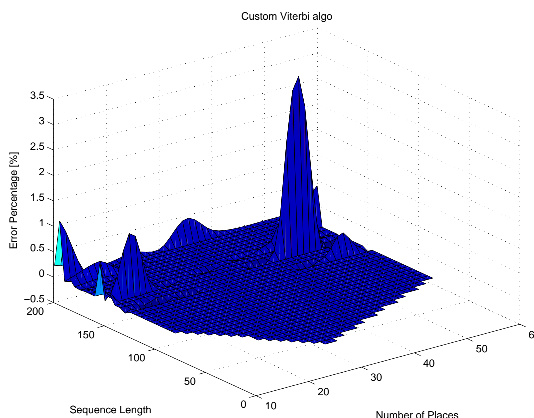


Figure I.2: *Plot of estimation error percentage for the customised Viterbi algorithm in C++*

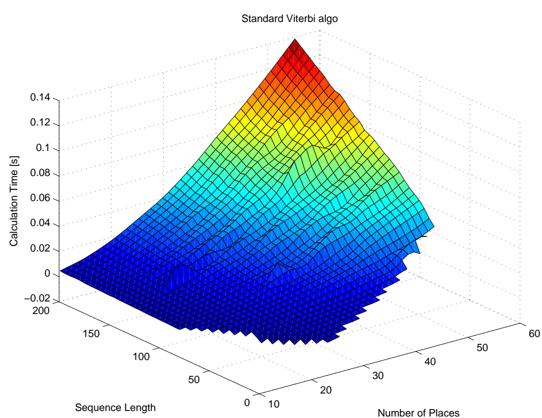


Figure I.3: *Plot of calculation time for the standard Viterbi algorithm in C++*

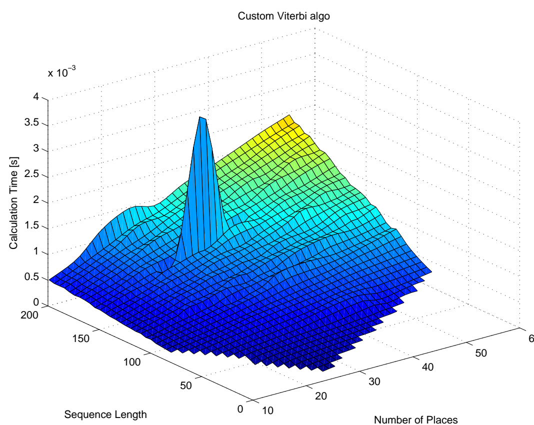


Figure I.4: *Plot of calculation time for the customised Viterbi algorithm in C++*

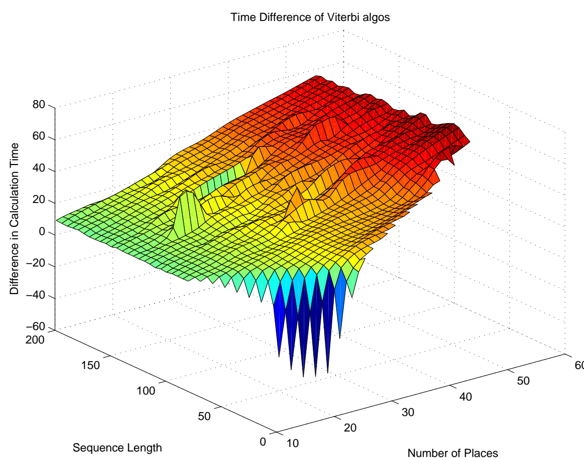


Figure I.5: *Plot of ratio in calculation time between the two algorithms in C++*

I.5 Results

The results of the test of the error percentage is shown in Figure I.1 for the standard algorithm and in Figure I.2 for the customised algorithm. The results of the time consumption test is shown in Figure I.3, Figure I.4 and Figure I.5. The plots in Figure I.6 and Figure I.7 shows a number of cross sections of the plot in Figure I.3, as it can be seen, the plots confirm that the computational complexity of the Viterbi algorithm is quadratic in the number of locations in the system and linear in the sequence length.

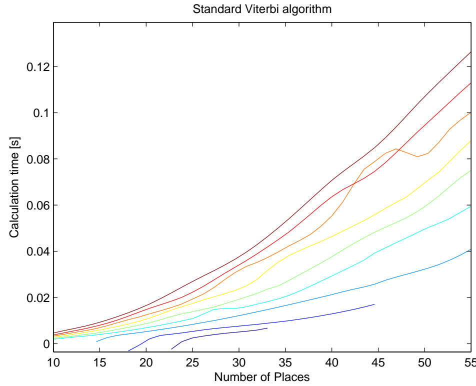


Figure I.6: *Plot of cross sections at different sequence lengths of Figure I.3*

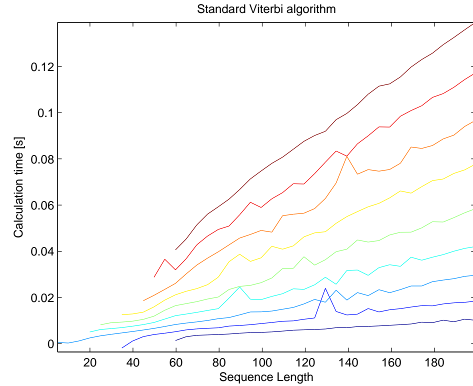


Figure I.7: *Plot of cross sections at different numbers of locations of Figure I.3*

The plots in Figure I.8 and Figure I.9 shows a number of cross sections of the plot in Figure I.4, as it can be seen, the plots confirm that the computational complexity of the custom Viterbi algorithm is linear in the sequence length, but that it is also linear in the number of locations opposed to being constant as described in section 2.2.2.

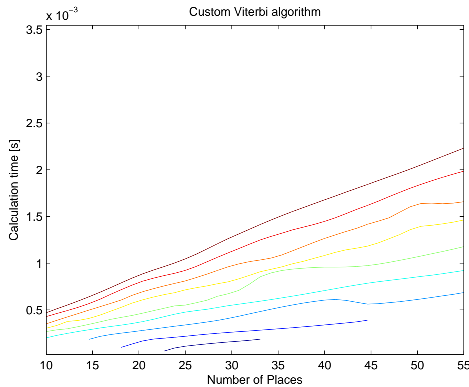


Figure I.8: *Plot of cross sections at different sequence lengths of Figure I.4*

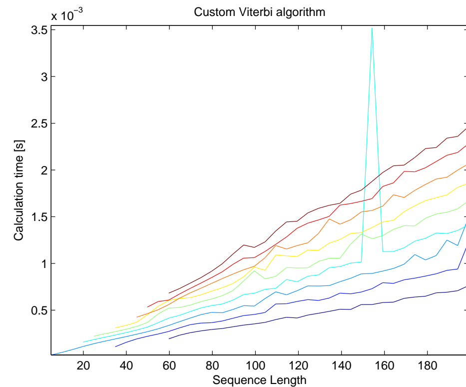


Figure I.9: *Plot of cross sections at different numbers of locations of Figure I.4*

The plot in Figure I.10 shows the measured data along with a Matlab polyfit of the topmost line in Figure I.6. The plot in Figure I.11 shows the calculation time extrapolated to 20e3 locations using the polynomial found with polyfit. The value estimated at 20e3 locations is with a mean (μ) of 1.62e4 seconds and a standard variation (σ) of 320.26 seconds.

The plot in Figure I.12 shows the measured data along with a Matlab polyfit of the topmost line in Figure I.8. The plot in Figure I.13 shows the calculation time extrapolated to 20e3 locations using the polynomial found with polyfit. The value estimated at 20e3 locations is with a mean (μ) of 0.79 seconds and a standard variation (σ) of 0.004 seconds.

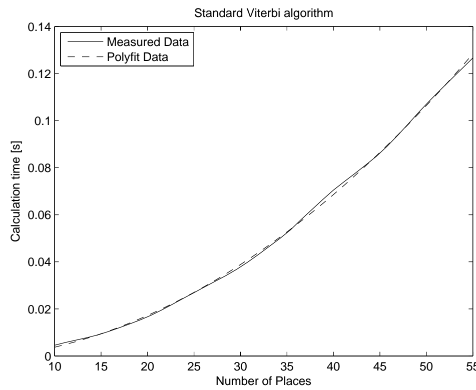


Figure I.10: *Illustration of measured data along with the polyfit at a sequence length of 200*

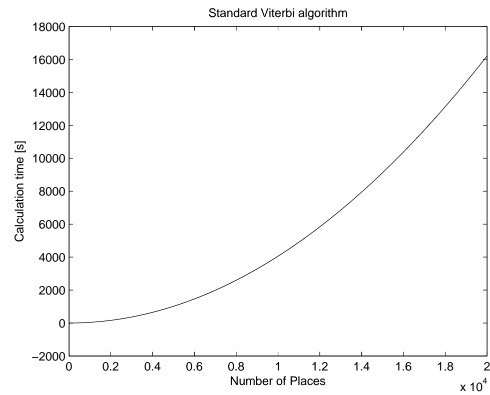


Figure I.11: *Extrapolation of calculation time using polyfit at a sequence length of 200*

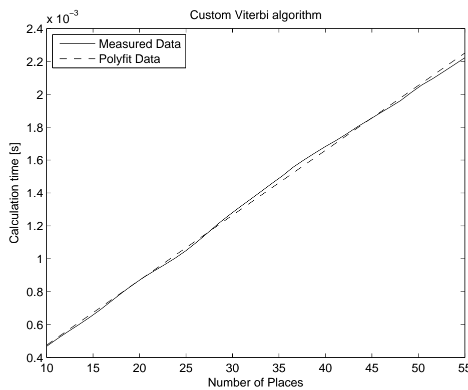


Figure I.12: *Illustration of measured data along with the polyfit at a sequence length of 200*

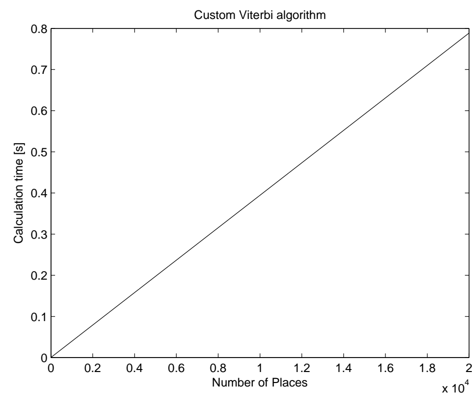


Figure I.13: *Extrapolation of calculation time using polyfit at a sequence length of 200*

I.6 Discussion

Figure I.1 and Figure I.2 shows that the error percentages are distributed exactly like the error percentages for the Matlab implementations which are found in Figure G.1 and Figure G.2.
It is apparent from Figure I.3 and Figure I.4 that the time consumption in the C++ implementation is lower for both the algorithms compared to the Matlab implementation, which can be seen when comparing with Figure F.1 and Figure H.2. The spike in Figure I.4 is assumed to be caused by an outlier in the results, which may be caused by scheduling. The ratio in calculation time is lower than for the Matlab implementations, which can be seen when comparing Figure I.5 and Figure H.5. The reason why the ratio in calculation time of the two algorithms is lower than in the case of the Matlab implementations is expected to come from the choice of implementation. As described in section 2.2.2, the algorithm treats the states corresponding to the last non-empty measurement as special cases and calculates special probabilities for those states, all other states of the same type are assigned the same probabilities due to the symmetry of the system. The assignment of probabilities to states which produce the same probability, is done using for loops in the C++ implementation of the custom Viterbi algorithm, which becomes time consuming when the number of locations in the system increase. The efficiency of the algorithm does though still grow with the number of locations when comparing it with the standard algorithm which is apparent from Figure I.5.
Even though the C++ implementation of the Viterbi algorithm is faster than the Matlab implementation it is still infeasible for use in the system, which can be concluded from the plot of the extrapolation of the calculation time in Figure I.11. Based on the extrapolation it will take between 1.97e3 and 2.13e3 ($\mu \pm 2\sigma$) years to calculate the state sequences for 4e6 assets when there is 20e3 locations in the

system and the sequence length is 200. This can be reduced to between 36 to 39 days if it is assumed that the computational burden can be distributed to 20e3 computers.
From the extrapolation of the calculation time of the custom Viterbi algorithm to 20e3 locations, it can be calculated that it will take between 36 to 37 days for a single computer to calculate the state sequences of 4e6 assets if the sequence length is 200. This time can be reduced to between 156.4 to 159.6 seconds if the computational burden can be distributed to 20e3 computers. Since a sequence length of 200 corresponds four days if the time is split into intervals of half an hour, it is concluded that it is feasible to use a distributed version of the algorithm as state sequence estimator in the system.

I.7 Conclusion

Since the distributions of error percentages are the same as the ones for the Matlab implementations of the algorithms it is concluded that the C++ implementations produce the same state sequences when given the same output sequences as the Matlab implementations, and are thus implemented correctly.
The test furthermore shows that it is feasible to use a distributed version of the custom Viterbi algorithm as state sequence estimator in the system.

Appendix J

Test of Load Balancing Algorithm

This chapter describes the test of the load balancing algorithm, which is used in the distributed software. The load balancing algorithm is used to ensure equal amounts of assets managed by each peer in the system.

J.1 Purpose

The purpose of the test is to determine if algorithm is able to balance the assets between the peers in the system. Furthermore the purpose is to test that an equilibrium is reached in a finite number of time steps.

J.2 Theory

The theory behind the load balancing algorithm is found in section 5.2. It is expected that the system will reach an equilibrium

$$load = \frac{N}{m}, \qquad \text{for } N >> m \tag{J.1}$$

Where:
N is the number of assets in the system
m is the number of peers in the system

J.3 Setup

The setup for the test is as follows; two computers are equipped with six network interfaces each, thus together are able to represent 11 peers and one server. This is the number of locations in the data file from Post Danmark. The number of assets in the file is 22257, and the sequence length is 1357. Four different measurement files are created, using the pure HMM simulation model, with the following parameters; 11 locations, 100 assets and sequence lengths of 100, 80, 30 and 20. The A and B probabilities are set to 80% and 30% respectively. The reason for the less assets and shorter sequence length, is based on the excessive time it would take to run the test. More assets added, to the system results in misbehaviour of the system. This is expected to be caused by a tcp buffer size. Due to time constrains no further investigation of this issue has been done. Because of this the number of assets is limited to 100. The measurement files are used to simulate readings from the RFID ports.

The current load of each peer, is written to a file, each time a successful load balancing between two peers has been completed. These output files are compared graphically in the result section.

J.4 Equipment

The equipment used in the test of the load balancing algorithm is listed in Table J.1.

Equipment	AAU number	Description
Desktop PC	52548	Pentium III@800MHz, 256MB memory, running Linux 2.6.18-4
Desktop PC	46787	Pentium III@800MHz, 512MB memory, running Linux 2.6.18-4
Switch	–	Netgear FS108

Table J.1: *Equipment used for load balancing test*

J.5 Results

The results from the test is shown in figure J.1, J.2 J.3 and J.4, which show the number of managed assets at each of the eleven locations, the number of managed assets is plotted when load balancing between two peers is in progress.

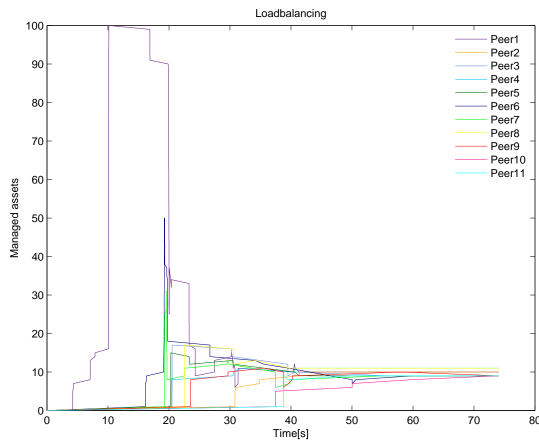


Figure J.1: *Load balancing run with a sequence length of 100*

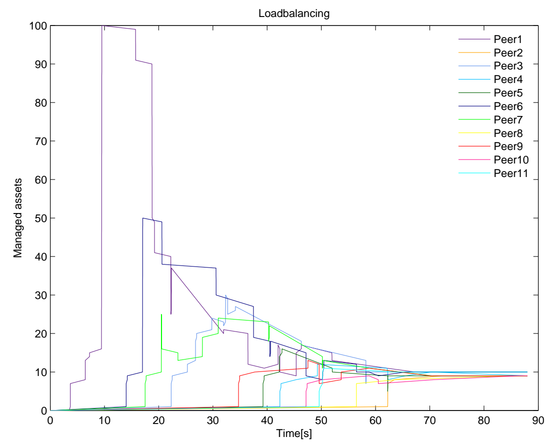


Figure J.2: *Load balancing run with a sequence length of 80*

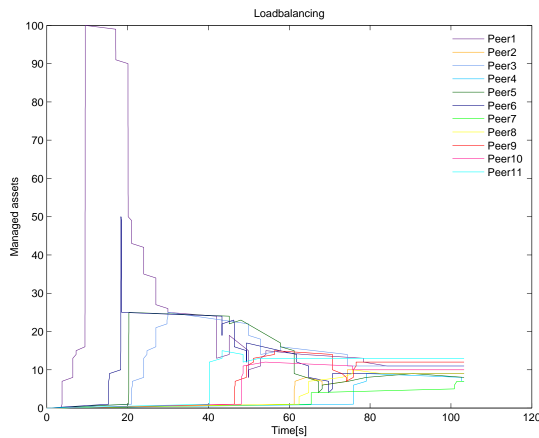


Figure J.3: *Load balancing run with a sequence length of 30*

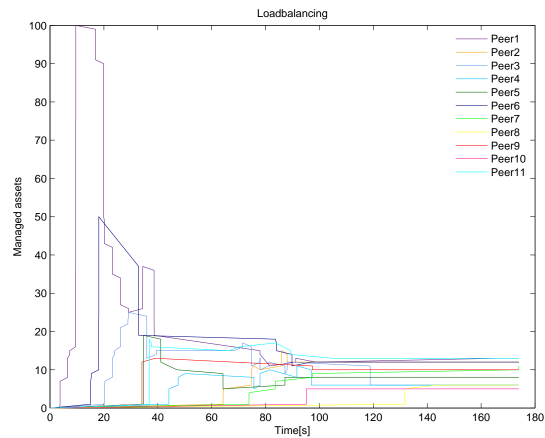


Figure J.4: *Load balancing run with a sequence length of 20*

J.6 Discussion

The result shows that with a long sequence length, shown in Figure J.1 and J.2, which means that more measurements for each asset are included in the measurement file, the load balance algorithm is able to equally balance the load between the peers in the system, and the number of assets managed by each peer converges toward

$$\frac{N}{m} = \frac{100}{11} = 9.09 \tag{J.2}$$

On the other hand with a short sequence length, the load balancing algorithm is not executed enough times to equally balance the system as seen in Figure J.3, and J.4, this is the expected behaviour of the algorithm.

J.7 Conclusion

The conclusion is that the load balancing algorithm is able to balance the number of asset between the peers in the system, as the number of managed assets converges towards an equilibrium given enough interaction between the peers. The interactions needed is dependent on the number of assets and the number of location in the system. As seen in Figure J.1 and Figure J.2 the equilibrium is reached within 100 s for 100 assets and eleven peers.

Appendix K

Graphical User Interface for Model and Algorithms

As a part of the modelling of the distribution system, a graphical user interface (GUI) has been constructed, in order to ease the interaction with the Simulink model, and to present the simulation results in a easy comparable, and readable format. The GUI is implemented in Java, using Matlab.

From within the GUI the parameters, used in the Simulink model, can be changed.

These parameters are; number of assets, number of places, probability of self loop in location state, probability of self loop in transport state, scan probability, probability of faulty scan, length of observation sequence, and the seed for the random number generator. The main window of the GUI, is shown in Figure K.1, which consist of the eight main input fields for the simulation parameters, and a number of buttons, the buttons have the following functions associated to them:

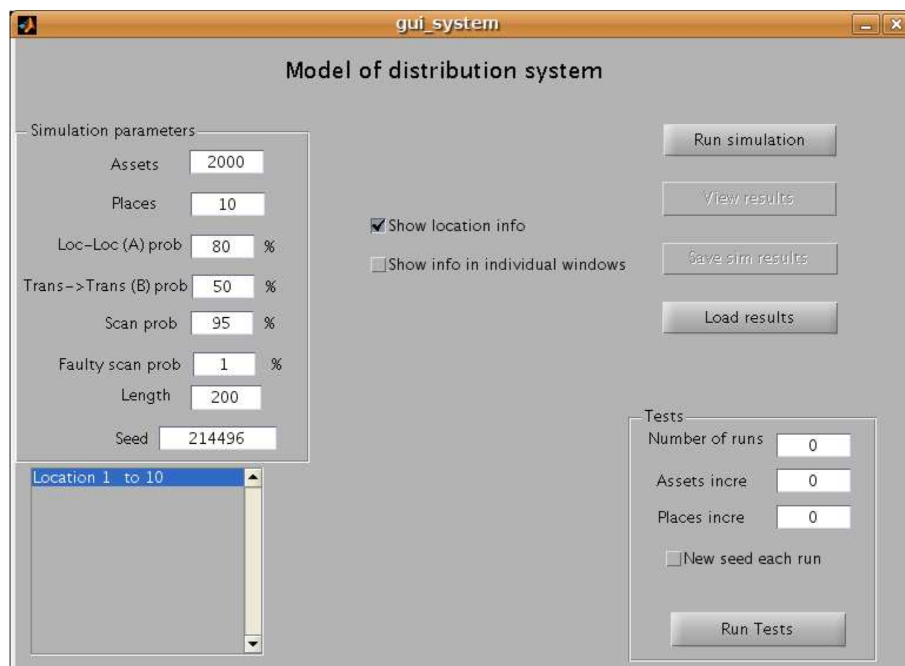


Figure K.1: *The main window of the graphical user interface*

1. *Run Simulation* Executes the Simulink model, with the entered parameters

2. *View results* Opens a new window to present the simulation results, this button is only available when results are present.
3. *Save sim results* Saves the simulation results, for later processing, this button is only available when results are present.
4. *Load results* Loads previously saved results.
5. *Run Tests* Runs the simulation, with the parameters specified.

In the lower right corner of the main window, a test section is present. The section is used to run the simulation a number of times, with increasing number of assets and places for each run. When the *Run Tests* button is pressed, a save to file dialog appears where the file name for the result should be entered. A struct for each run will be created in the file, where the simulated state and event sequences are present together with all the simulation parameters used in the simulation.

An activity diagram for the GUI, has been constructed, and can be seen in figure K.2, which shows the construction of the GUI.

A checkbox exists in the main window, called 'show location info'. When this is checked, and the view results button is pressed, the distribution of the assets in the system is shown at each time step in the simulation. If there exists more than ten locations in the simulation, the user is able to choose, for which ten locations the distribution of assets should be shown. An illustration of the distribution of the assets is shown in Figure K.3, where each colored line represent a location. Figure K.4 illustrates the number assets, which are in a transport state for each time step. When the show results button is pressed a new window appears, shown in Figure K.5, which consist of two panels, two buttons and an input field. The input field is used to enter the asset, for which the event, and state sequence should be shown. The state sequence can also be shown as a graphical state diagram, illustrated in Figure K.7. The state diagram should be interpreted in the following way; The number on the arcs indicate the time step, at which this transition has been taken. The 'T' state is a pseudo transport state, which includes transport states for all locations, to simplify and make the illustration more readable. An activity diagram for the 'view result' window is shown in figure K.6.

The last button, in Figure K.8 is the run algorithm, which makes another window appear. This window which is shown in Figure K.8, consist of a number of input fields, and three panels. The two first input fields is used to enter the parameters for the forward-backward algorithm, and the last input field is used by the Viterbi algorithm. The result is shown in the panels as the estimated state sequence and correct state sequence. The forward-backward panel shows the epsilon value, which is the value that shows convergence of the forward-backward algorithm, for the system matrix. The forward panel shows the probability calculated by the forward algorithm. An activity diagram for the 'run algorithm' window is show in in Figure K.9.

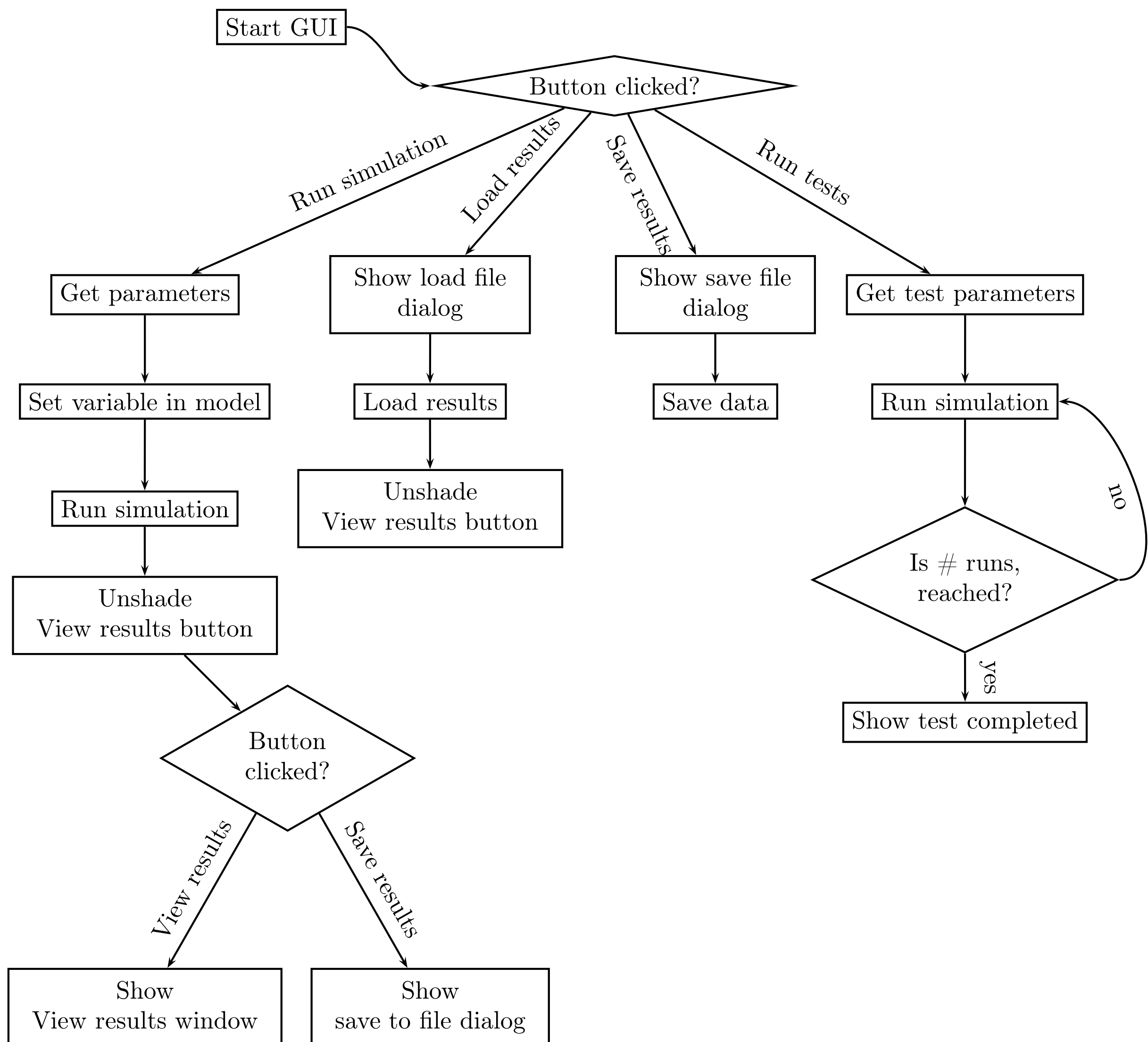


Figure K.2: *GUI activity diagram*

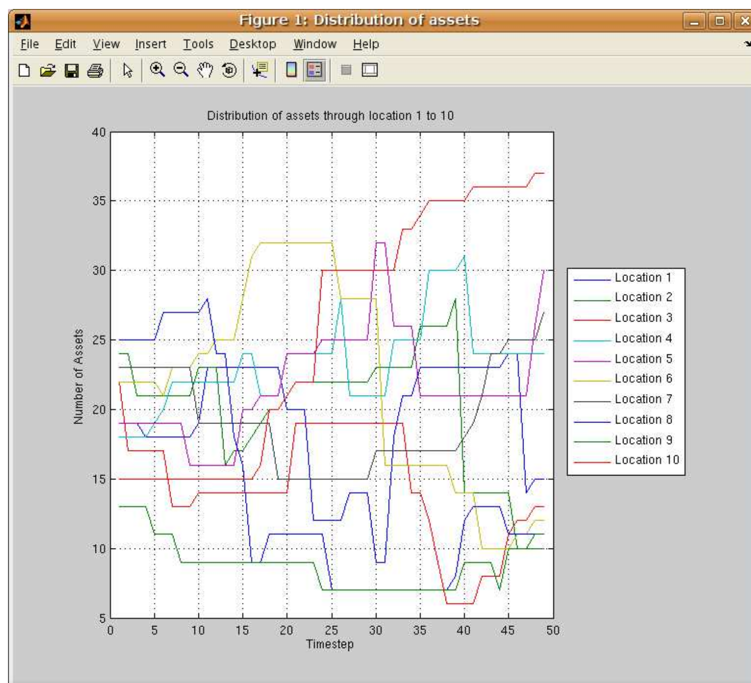


Figure K.3: *Illustration of asset distribution, each color represent a location in the system*

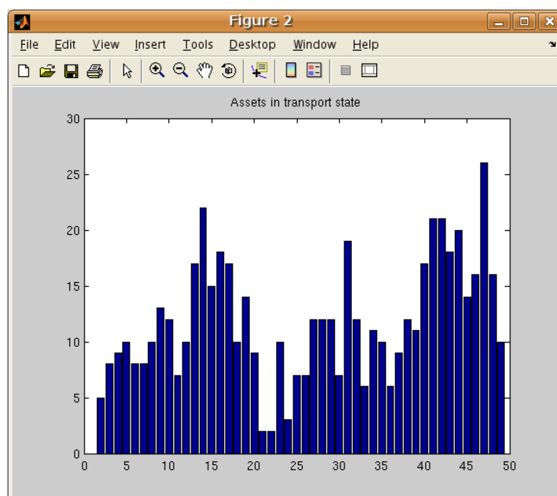


Figure K.4: *The number of assets, in transport as a function of time steps*

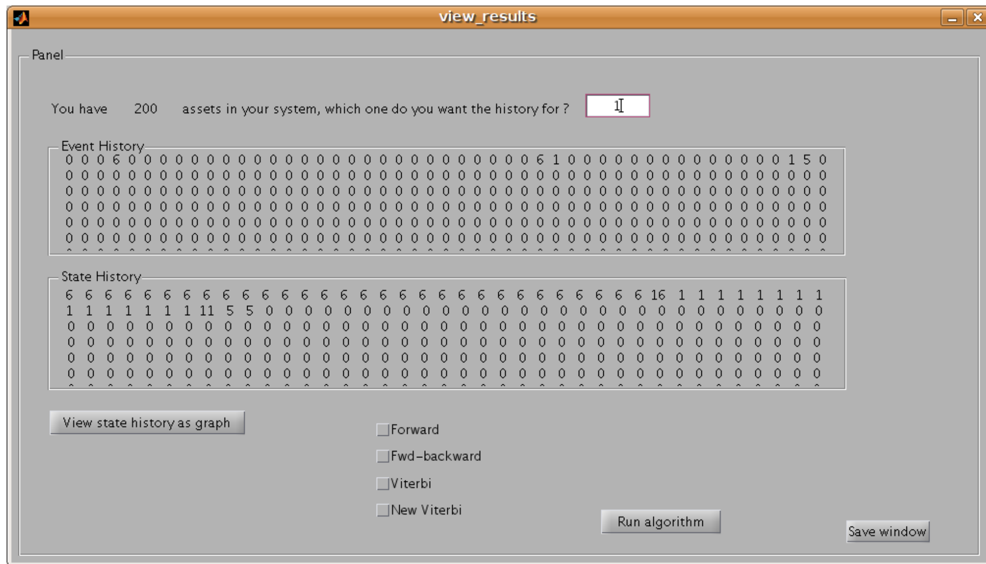


Figure K.5: *Illustration of the results window*

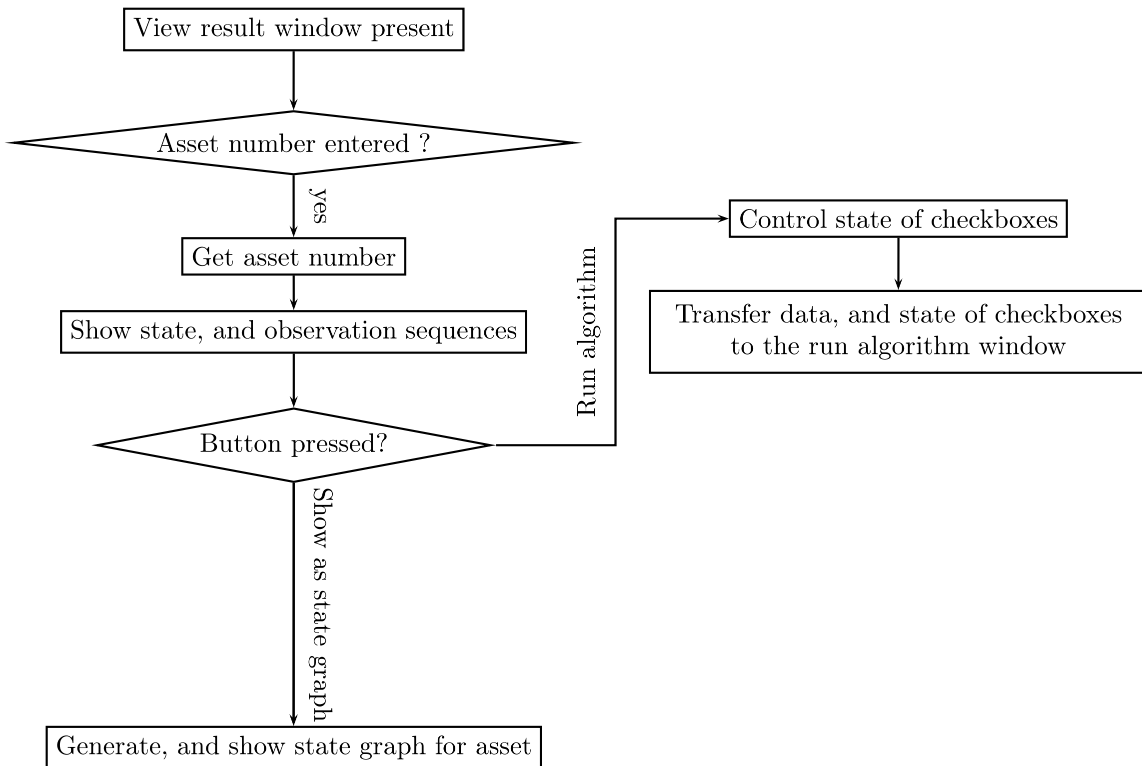


Figure K.6: *View result window activity diagram*

State sequence
6
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30
T
31 46
47
1
32 33 34 35 36 37 38 39 40 41 42 43 44 45
5
48
stop

Figure K.7: *Illustration of generated state diagram*

run_algorithm
Run algorithm
50 Enter maximum iterations
0.0001 Enter epsilon value
0.95 Enter scan prob
Run
Epsilon
Forward
Correct state sequence
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 13 13 13 8 8 8 8 18 6 6 6 6
Estimated state sequence
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 11 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 13 13 13 8 8 8 8 18 6 6 6 6
View as state diagrams
Save results

Figure K.8: *Illustration of the algorithm window*

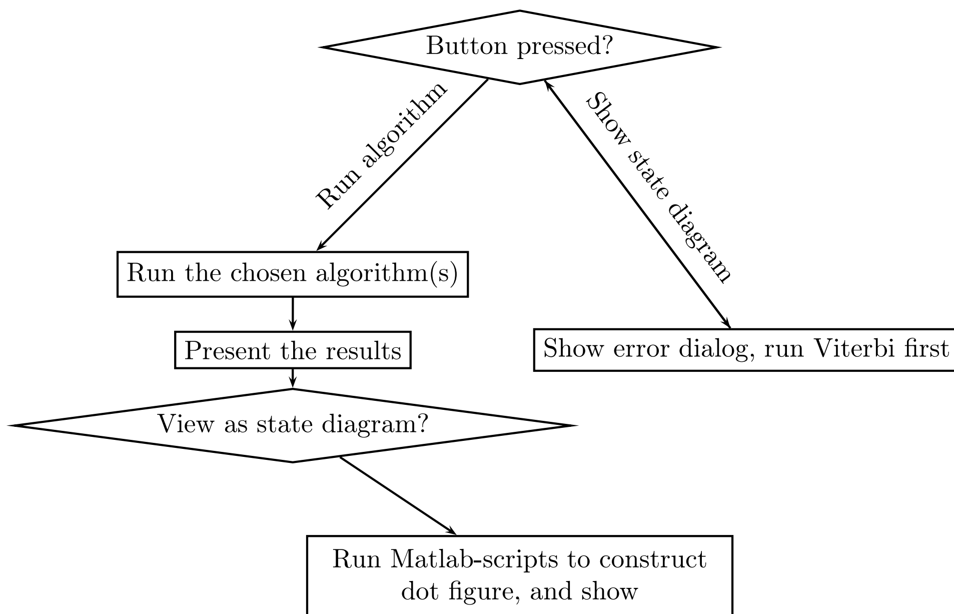


Figure K.9: *Run algorithm activity diagram*