Imperial College London

Department of Computing

Individual Project: Final Report

Java Algorithms for Computer Performance Analysis

Student:
Benjamin Homer
(bmh10@ic.ac.uk)

Supervisor:
Giuliano Casale
Second Marker:
Peter Harrison

June 16, 2014

Abstract

Performance analysis techniques which are capable of accurately modelling and predicting system performance measures are widely used for capacity planning, system tuning and workload analysis. For this purpose, networks are typically modelled mathematically as closed queueing networks from which metrics such as throughput, resource utilisation and response time can be derived. As computer systems continue to increase in both scale and complexity, the application of well-established performance analysis methods which provide exact results, such as Convolution and Mean Value Analysis (MVA), become inefficient and, in many cases, infeasible.

In the past, non-iterative bounding techniques such as Asymptotic Bounds (ABs) and Balanced Job Bound (BJBs) were established as an efficient, approximate alternative; capable of finding upper and lower limits on performance measures at a fraction of the computational cost. More recently, Geometric Bounds (GBs) were put forward as an new technique capable of obtaining much tighter bounds. In the first part of this report we document the development and integration of all three of these bounding techniques within JMVA, an open-source queueing network analysis tool. We then evaluate their performance by comparing the speed and accuracy of these methods over several network models. The results show that GBs significantly reduce the maximum bounding error in comparison with the other methods.

An alternative approach is to try and optimise the exact methods themselves. With this in mind, Tree Convolution (TC) and Tree MVA (TMVA) were suggested as practical extensions of their sequential counterparts. These algorithms are theoretically much more efficient when analysing sparse networks, which are common in commercial computer systems. Significant space and time savings are possible by arranging queueing networks as tree data structures in a way which allows the exploitation of network routing information. These algorithms also lend themselves to a parallel implementation, further increasing their potential for fast and accurate analysis. The main part of this report presents the development of a library implementing these techniques, and their subsequent integration into JMVA. We then evaluate the effectiveness of these algorithms by running several experiments and discuss their applicability for practical analysis purposes. For sparse networks, our evaluation shows that both tree algorithms significantly outperform existing sequential techniques, both in terms of runtime and memory usage.

Acknowledgements

I would like to thank my supervisor Giuliano Casale for his support, encouragement and technical guidance throughout the year. In addition, I would like to thank my friends and family for their inspiration and understanding.

Contents

1 Introduction **5**
1.1 Motivation . . . 5
1.2 Contributions . . . 6

2 Background **7**
2.1 General Performance Analysis Techniques . . . 7
2.2 Queueing Network Models . . . 8
2.2.1 Model Description . . . 8
2.2.1.1 Customer Description . . . 9
2.2.1.2 Station Description . . . 10
2.2.1.3 Service Demands . . . 11
2.2.2 Fundamental Laws . . . 11
2.2.2.1 Common Measures . . . 11
2.2.2.2 Little's Law . . . 12
2.2.2.3 Forced Flow Law . . . 12
2.2.2.4 Flow Balance Assumption . . . 12
2.2.3 Model Outputs . . . 13
2.2.3.1 Throughput . . . 13
2.2.3.2 Utilisation . . . 13
2.2.3.3 Residence and Response Time . . . 13
2.2.3.4 Queue Length . . . 13
2.2.4 Limitations of Queueing Models . . . 14
2.3 Introduction to JMT . . . 15
2.4 Introduction to JCoMoM package . . . 16

3 Queueing Network Analysis Techniques **18**
3.1 Performance Bounding Techniques . . . 18
3.1.1 Asymptotic Bounds . . . 19
3.1.2 Balanced Job Bounds . . . 21
3.1.3 Geometric Bounds . . . 22
3.2 Iterative Solution Techniques . . . 24
3.2.1 Solution of Open Models . . . 24
3.2.2 Solution of Closed Models . . . 25
3.2.3 Convolution . . . 26
3.2.4 Multi-class sequential Convolution . . . 27
3.2.5 Mean Value Analysis (MVA) . . . 29
3.2.6 Approximate MVA (AMVA) . . . 33
3.2.7 Related Work . . . 34

4 Tree Algorithms for Sparse Networks 35
4.1 Tree Convolution (TC) . . . 35
4.1.1 Theoretical Concepts and Notation . . . 35
4.1.2 Algorithm Overview . . . 36
4.1.3 Normalization Constant Calculation . . . 37
4.1.4 Feedback Filtering . . . 38
4.1.5 Performance Measure Computation . . . 39
4.1.5.1 Throughputs . . . 39
4.1.5.2 Mean Queue Lengths . . . 39
4.1.6 Space-Time Trade-offs . . . 41
4.1.7 Complexity Analysis . . . 41
4.1.8 Algorithm Pseudocode . . . 41
4.2 Tree MVA (TMVA) . . . 42
4.2.1 Algorithm Overview . . . 42
4.2.2 Potential Optimisations . . . 44
4.2.3 Complexity Analysis . . . 45
4.2.4 Algorithm Pseudocode . . . 45
4.3 Tree Planting . . . 46

5 Design & Implementation 48
5.1 Existing JMT Architecture . . . 48
5.2 Bounding Solvers Implementation . . . 50
5.2.1 Single-class Bounding Solvers Architecture . . . 50
5.2.2 AB Solver . . . 51
5.2.3 BJB Solver . . . 51
5.2.4 GB Solver . . . 52
5.2.5 Multi-class Bounding Solvers Architecture . . . 53
5.2.6 Integration with JMVA . . . 53
5.2.7 Testing and Validation . . . 55
5.3 Tree Algorithms Implementation . . . 56
5.3.1 Solution Architecture . . . 56
5.3.1.1 Existing JCoMoM Architecture . . . 56
5.3.1.2 Overview of new `TreeAlgorithms` package . . . 56
5.3.2 Algorithm Configuration . . . 58
5.3.3 Preprocessor Stage . . . 58
5.3.3.1 `TreePlanter` . . . 58
5.3.3.2 `ComplexityEvaluator` . . . 60
5.3.4 Tree Traversal . . . 60
5.3.4.1 Single-threaded Traversal . . . 60
5.3.4.2 Multi-threaded Traversal . . . 61
5.3.5 Tree Convolution Implementation . . . 61
5.3.5.1 `ConvolutionLayer` . . . 62
5.3.5.2 `ConvolutionCore` . . . 62
5.3.5.3 Memory Management . . . 63
5.3.5.4 Optimisation Log . . . 63
5.3.6 Tree MVA Implementation . . . 64
5.3.6.1 `TMVACore` . . . 64
5.3.6.2 Ported sequential MVA solver . . . 67
5.3.6.3 Optimisation Log . . . 67
5.3.7 Integration with JMVA . . . 67

5.3.8 Measuring Sparsity . . . 69
5.3.9 Generating Random Networks . . . 69
5.3.10 Commandline Tool . . . 70
5.3.11 Testing and Validation . . . 70

6 Evaluation **71**
6.1 Geometric Bounds Evaluation . . . 71
6.2 Tree Algorithms Evaluation . . . 74
6.2.1 Runtime comparison . . . 74
6.2.2 Memory usage comparison . . . 75
6.2.3 Stress Testing . . . 75
6.2.4 Non-sparse networks . . . 77
6.2.5 Single vs multi threading . . . 78
6.2.6 Tree Convolution Breakdown . . . 79
6.2.7 Key Observations . . . 79
6.2.8 Real World Example . . . 80
6.2.9 Comparison with MoM, CoMoM and RECAL . . . 81
6.3 Qualitative Aspects . . . 81
6.4 Strengths and Weaknesses . . . 82

7 Conclusion **83**
7.1 Future Work . . . 84

A Real-world Example Service Demands **88**

Chapter 1

Introduction

1.1 Motivation

In the current era of computing, distributed systems have continued to become more prevalent and complex, with many companies now choosing to run their applications across globally dispersed data centres. As the software world continues to adapt to these new scales of operation there has become an increased focus on analysing these systems in order to optimise performance. Often techniques such as benchmarking, profiling and metrics collection can be employed to get an overall idea of how a system is performing over a period of time. Of course the downside to these approaches is that the system has to be built first. In order to circumvent this need, queueing network models can be used, allowing engineers to analyse a system's expected performance during the design phase.

Queueing network models are a widely researched performance analysis tool which can be used to describe a variety of networks. Each model describes a system as a network of stations (also known as queues), with each station representing a system resource. Various classes of customers, representing jobs in the system, travel around this network being serviced by stations. This kind of model is useful for analysing expected system performance over varying workloads, assessing how system architecture changes will affect performance, and identifying network bottlenecks [23].

In this project we focus on a particular subset of queueing networks called closed product-form queueing networks [1] since they are amenable to a variety of solution techniques [2]. There are several techniques which can be used to evaluate these models in order to obtain various per server and whole system performance measures, such as mean throughput, utilisation and response time. Two of the most well-known algorithms for this purpose are Mean Value Analysis (MVA) [28] and Convolution [27]. The problem with these algorithms is that, while they provide exact solutions, they quickly become computationally infeasible as the system scales to larger numbers of customer classes or service stations, thus making the evaluation of complex modern systems impossible.

This problem led to the development of several approximate MVA algorithms which were the focus of an earlier project [13]. These methods make various assumptions in order to dramatically reduce the cost of computation and hence allow fairly accurate solutions to be found for large models. In this project we begin by investigating another type of approximate analysis which focusses on obtaining upper and lower bounds on performance. These methods are typically of a non-iterative nature, making them extremely fast and useful during the initial stages of analysis [23]. Some modern applications such as self-optimising systems require that thousands of possible configurations be evaluated in real-time, meaning that the underlying evaluation mechanism must be extremely efficient and accurate [6]. For such systems, iterative techniques are too expensive and so accurate bounding methods are preferred. In particular we focus on the implementation of Geometric Bounds (GBs) [6], a recently established technique, since GBs are more accurate than other methods such as Asymptotic Bounds (ABs) [24], Balanced Job Bounds (BJBs) [20, 37] and Proportional Bounds (PBs) [22] and require a similar computational cost. In order to justify this claim ABs and BJBs are also implemented for comparison.

After experimenting with bounding techniques, we then move onto the main part of this project where we develop more advanced versions of both the MVA and Convolution algorithms which make use of a tree structure in order to reduce the cost of computation. The Tree Convolution [21] and Tree MVA [34, 17] algorithms exploit network routing information and make use of tree traversal techniques in order to significantly reduce the computational cost of computing an exact solution. This allows models with large numbers of stations and customer classes to be evaluated in a reasonable amount of time, provided that each customer class only visits a small fraction of the stations in the network. This condition, referred to as the *sparseness property*, is exhibited in many commercial computer systems. Both of these tree algorithms provide exact, rather than approximate, solutions.

There are various software packages that use queueing network models to evaluate systems, however the software we will be working with closely is Java Modelling Tools (JMT) [18]. JMT is a suite of several tools for system performance analysis and planning which receives thousands of downloads every year. In this project we are interested specifically in a tool called JMVA which can currently be used to analyse product-form queueing models using various exact and approximate algorithms. We will be attempting to extend this software by implementing the aforementioned techniques in order to increase the range of networks the tool is able to solve.

1.2 Contributions

Below are listed the main contributions which this project makes, along with the references to the related sections in this report:

- A Java implementation of Asymptotic Bounds, Balanced Job Bounds and Geometric Bounds (as proposed in [6]), providing bounds on throughput, mean queue length, residence times, utilisation and system power (section 5.2).

- An efficient implementation of the Tree Convolution algorithm as proposed in [21] and the Tree MVA algorithm as proposed in [34, 17] (section 5.3).

- Extension of the tree algorithms to a multi-core approach in which subnetworks within the tree can be evaluated in parallel (section 5.3.4.2).

- Implementation of tree planting and complexity evaluation methods, allowing the tree planting algorithms to run efficiently (sections 5.3.3.1 and 5.3.3.2). These implementations include new methods which were introduced to determine whether a given queueing network is suitable for solution by one of the tree algorithms (section 5.3.8).

- Integration of the above algorithms into JMVA, allowing them to be used intuitively alongside the existing algorithms. This required an extension of the JMVA user interface, including graphical visualization of the results for each algorithm and a new configuration options menu for the tree algorithms (sections 5.2.6 and 5.3.7).

- Evaluation and analysis of each implemented technique and comparison of the results with the claims made in relevant papers (chapter 6). In particular, for the tree algorithms there was a focus on experimenting with different implementations and tree planting procedures in order to draw some conclusions about how best to optimise the algorithms for practical network evaluation.

Chapter 2

Background

2.1 General Performance Analysis Techniques

The performance of any system can be defined empirically by looking at the amount of useful work done in proportion to the time taken and the resources used to complete the work. Performance analysis is a discipline which uses such measurements in an attempt to evaluate *why* systems perform in a certain manner and, often more importantly, provide insights into how a system's performance can be improved. While this project focuses mainly on the performance analysis of computer systems, a lot of the ideas discussed, particularly in relation to queueing networks, can be extrapolated to the analysis of more general systems, such as in the spheres of business or manufacturing.

Of course in the modern landscape of software engineering, performance analysis is often a necessity, allowing software and hardware configurations to be optimised for the specific goals of a company. This is sometimes referred to as *performance tuning* [32]. Performance analysis techniques can also be used for estimating the impact of modification to a system, and are particularly useful for studying system bottlenecks. The results allow systems to be compared against each other successfully and to be described in absolute terms, which is often necessary in order to prove that Service Level Agreements and quality-of-service targets are being met.

One method of approaching analysis is by focusing on the collection of quantifiable metrics. For example, robust computer system performance may be characterised by attributes such as high throughput, high availability of resources, high bandwidth, low response time, low resource utilisation and low packet drop rate. The metrics used in practice will be highly dependent on the specific circumstances under which the system is running and what the purpose of the analysis is.

Another common way of analysing performance is through the use of benchmarks [25]. The purpose of benchmarking is to assess relative performance of hardware or software by running a number of trials. As with metrics collection, the type of benchmark employed is likely to be highly application specific. Notable industry standard benchmarks include SPECint and SPECfp, though there are also many open-source benchmarking tools available. Problems with this approach include its complexity (often the tests must be run multiple times to gain useful conclusions) and the notoriously difficult task of interpreting the results (which marketers often abuse to sell their products). The design of benchmarking suites is also hard to get right, for example many benchmarks focus purely on computational performance whilst neglecting other important aspects such as scalability, reliability and security.

In terms of software, profiling is another technique which can be employed to perform dynamic program analysis, typically with the goal of optimisation. Profiling tools usually measure features such as space and time complexity of a program, and metrics related to instruction use or function calls. This is achieved by instrumenting the binary executable or source code using a profiler tool.

Over recent years, performance analysis has also played a part in the introduction of systems which are capable of self-optimising [36]. The systems which make use of such technologies are typically highly complex, with multi-layered architectures and unpredictable traffic flow [6]. Such systems can adapt to changes in real-time in order to meet requirements. This can be achieved by the evaluation of all possible configurations using nonlinear programming techniques [15]. In situations like this, where thousands of configurations may need to be searched, efficient and accurate analysis techniques are mandatory. We discuss and develop such techniques as part of this project.

Although all of the techniques outlined above are valid ways of approaching performance analysis, we will be focussing our attention on the use of queueing network models which we now discuss in more detail.

2.2 Queueing Network Models

Before we can begin to carry out performance analysis on a system we must first understand how the system can be represented effectively as a model and what the fundamental laws of this model are. In particular we focus on a subset of queueing network models known as product-form (or separable) queueing networks [1] which allow a balance to be struck between accuracy of representation and computational efficiency. Product-form queueing networks are stochastic models which have been widely studied in the arena of performance analysis due to the existence of several simple and relatively efficient solution techniques [2], for example the Convolution algorithm [27] and the Mean Value Analysis (MVA) algorithm [28]. Conveniently the parameters which describe product-form models have a direct correspondence with high-level descriptions of computer systems [23]. In addition they have the useful property that each resource within the model can be evaluated in isolation, with the solution for the whole system being a combination of the individual resource solutions. This is in fact why they are sometimes referred to as *separable* queueing networks [23]. Other types of model are possible, however a lack of exact solution methods makes these representations less desirable in practice. In the following subsections we define the components of product-form queueing models and discuss some of the underlying laws.

2.2.1 Model Description

Queueing network models consist of stations, which represent system resources, and customers, which represent users or jobs in the system. These models can be defined as single-class, in which there is only one type of customer, or multi-class, in which their are several types of customer. While single-class models are sufficient in some circumstances, multi-class models allow the representation of different types of customer or request in a system and so can offer a more fine-grained description of a real-world system. However, there are also some downsides to multi-class models. From a user's perspective, the use of multi-class models means that there are more input parameters to fill in before the model can be evaluated. Additionally, the outputs from multi-class models are often less accurate than single-class models, due to the currently available measurement tools. There is also the concern of evaluation complexity, as the techniques available for the solution of multi-class models are significantly more involved than the methods used to solve single-class models and hence tend to use more system resources. [13]

As a concrete example of the kind of system we wish to model, consider a typical bank in which customers must queue in order to interact with a bank teller. The bank tellers represent stations in the system as they 'process' customers from the queue, while the customers are essentially jobs in the system which must be processed. If the customers were to be classified by their intentions, for example ‘bill-paying customers’, ‘cash withdrawal customers’ etc., then the system would be multi-class. When considering this model it is clear that there are certain characteristics of the system which will impact how it changes over time, for example whether queue length is constant or there is a fixed arrival rate of new customers into the queue. Multi-class systems are inherently more complex to analyse as each type of customer may require a different amount of time to be serviced by the bank tellers. In addition, the tellers themselves may be more adept at handling a certain class of customer and hence the differences between the tellers must also be incorporated into the model. Careful analysis of such a system would enable it to be optimised to meet desired criteria, for example maximized throughput. We now consider each of the system components in turn.

2.2.1.1 Customer Description

There are three different ways the workload intensity for a specific type of customer can be described. These different descriptions, outlined below, allow various types of system workloads to be successfully modelled: [23]

- *Transaction workloads* describe the workload intensity by specifying an arrival rate λ at which customers arrive into the system. As a result of this the number of customers in the system varies over time. Once customers have been serviced they leave the model. Note that if the arrival rate is high and the throughput of the system is relatively low there will be a threshold at which point the number of customers in the system will continue to increase infinitely. This phenomenon is known as system *saturation.*

- *Batch workloads* describe the workload intensity by specifying the average number of active customers in the system, N. Since N is constant the average number of customers in the system is also constant. Customers that have been serviced leave the model and are instantly replaced from a backlog of waiting customers.

- *Terminal workloads* describe the workload intensity by specifying N, the number of active customers, and Z, the *think time* describing the average length of time between a customer finishing an interaction with a station and starting a new interaction. If Z is set to zero then this is the same as a batch workload.

Models consisting of only batch and terminal workloads are often described as *closed models* since the customers within these systems recirculate and there is no influx of new customers into the system. On the other hand, models containing only transaction workloads are referred to a *open models* since there is an infinite stream of customers both entering and exiting the system. With multi-class models there is also the possibility there may be a mixture of batch, terminal and transaction workloads, in which case the model is described as *mixed.* The techniques that can be employed to evaluate a model vary depending on the categorisation of the model as open, closed or mixed. [23]

It should be noted that in multi-class models the workload intensity is defined on a per class basis which can be represented in a vector. So, for example, in an open model we represent the workload intensity as $\lambda = (\lambda_1, \ldots, \lambda_c)$ where λ_i is the arrival rate for customer class i and c is the total number of customer classes. Similarly for a closed model the workload intensity is represented by N = $(N_1, \ldots, N_c)$ where N_i is the average number of active customers of class i (a Z_c parameter must be added for terminal classes). Finally a mixed workload is represented as $I = (N_1 or \lambda_1, \ldots, N_c or \lambda_c)$. [23]

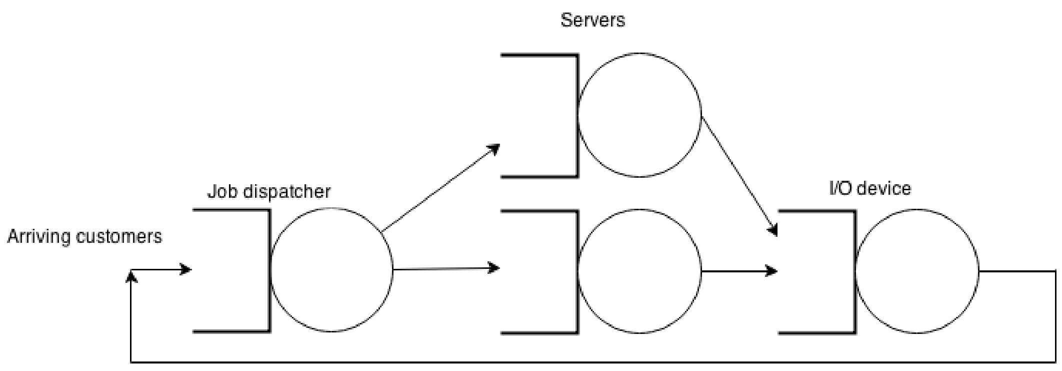


Figure 2.1: An closed model example.

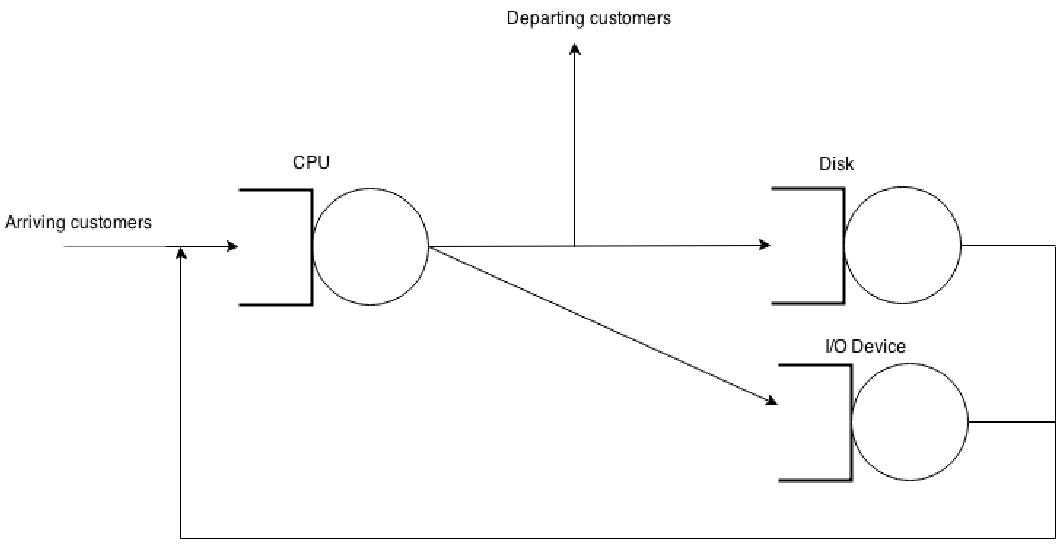


Figure 2.2: An open model example.

2.2.1.2 Station Description

Stations, also referred to as service centres or queues, represent system resources such as CPUs, I/O devices or types of server, which can process customers/jobs. Stations can be categorised as either *queueing* or *delay* stations. A queueing station comprises of a processing component, for example a CPU, and a queue of waiting customers. As such, these queueing stations can be used to represent resources for which users must compete. The total time spent by a customer at a queueing station is therefore the sum of the queue waiting time and the time spent receiving service. It is assumed that one customer is in service whenever there are customers at a station. In single-class models there is no need to specify a scheduling plan seen as all the customers at any given station will be identical. In multi-class models the assumption is made that the scheduling plan is class independent. Coupled with the earlier assumption this means that the performance measures obtained will not be dependent on the scheduling plan used and hence we do not need to specify a scheduling plan for multi-class models either. Queueing stations can further be categorized as *load dependent* or *load independent.* If a queueing station is load dependent then the service time at the station depends on the number of customers in the station's queue. [23]

Delay stations can be used to model resources for which there is no competition, as each customer at a delay station is logically allocated their own server. Since there is no queueing involved, the residence time of a customer at a delay centre is equal to the customer's service demand there. Delay stations are useful for modelling thinking times or delays which occur in a system, for example the average time it takes a user to interact with a system or the time taken for some data to be transferred over a network. [23]

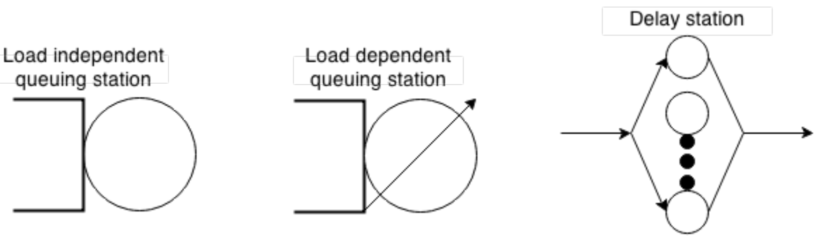

Load independent queuing station
Load dependent queuing station
Delay station

Figure 2.3: Station types.

2.2.1.3 Service Demands

Once the customers and stations have been defined, the final required inputs for the model are the service demands, denoted D_k for the service demand at station k in a single-class model. Intuitively D_k is the total amount of time a customer requires to be serviced by a station which can be calculated as B_k/C, where B_k is the busy time of station k and C is the number of system completions, or equivalently by U_kT/C, where U_k is the utilisation of resource k and T is the observation time. Alternatively the service demand D_k can be characterised as $V_k \times S_k$, where V_k is the number of customer visits to station k and S_k is the service time required per visit. So a model can be defined either by specifying D_k directly or by specifying V_k and S_k for each station k, although the solution depends only on D_k. This if fortunate seen as D_k is much more easily obtained from measurements than V_k or S_k. The total service demand of a customer is defined as $D = \sum_{k=1}^{K} D_k$. [23]

For multi-class models the service demands must be defined on a per class basis and are denoted $D_{c,k}$, which represents the service demand of a customer in class c at station k. The total service demand of a customer of class c can then be defined, $D_c = \sum_{k=1}^{k} D_{c,k}$. [23]

2.2.2 Fundamental Laws

We now take a look at some of common measures used to determine system performance and discuss the fundamental laws of product-form queueing systems which link these measures together.

2.2.2.1 Common Measures

Considering an abstract system where T is the length of time we observe the system, A is the number of arrivals into the system, and C is the number of completions (customers leaving the system) we can define the commonly used measures: [23]

- Arrival rate, $\lambda = A/T$
- Throughput, $X = C/T$ (simply the rate of request completions)

In a system with a single resource we can measure B, the time the resource was observed to be busy. We can then define two additional measures: [23]

- Utilisation, $U = B/T$ (this is normally given as a percentage, e.g. 30% server utilisation)
- Service requirement per request, $S = B/C$

From these measures we can derive the Utilisation Law as $U = XS$ [23].

2.2.2.2 Little's Law

One of the most useful fundamental laws is Little's Law which states that N, the average number of customers in a system, is equal to the product of X, the throughput of the system, and R, the average time a customer stays in the system. Formally this gives: [23]

$$N = XR \tag{1}$$

Although this may appear obvious it is actually quite an exceptional result seen as the relationship between the quantities is "not influenced by the arrival process distribution, the service distribution, the service order, or practically anything else" [33]. Little's Law is important because it can be applied in many scenarios. In particular it is common to face situations in which two of the three quantities which the law relates are known and we wish to calculate the third. Little's Law can also be applied to any subsystem within a model, from a single resource up to the whole system, as long as the law is used consistently. Note that the previously defined Utilisation Law is a special case of Little's Law. [23]

For terminal workloads in which a think time Z is used, the law can be rewritten as $N = X(R + Z)$. The application of this formula is so pervasive that is is often referred to as the Response Time Law when rearranged so that R is expressed in terms of the other quantities: [23]

$$R = N/X - Z \tag{2}$$

2.2.2.3 Forced Flow Law

Another fundamental law of queueing systems is the Forced Flow Law which, informally, states that the throughputs in all parts of the system must be proportional to one another. Formally the Forced Flow Law is given by: [23]

$$X_k = V_k X \tag{3}$$

where X_k is the throughput at station k, V_k is the visit count of resource k defined by $V_k = C_k/C$ (where C_k is the number of completions at station k and C is the total number of system completions), and X is the throughput of the whole system.

Combining both Little's Law and the Forced Flow Law allows for a wide range of scenarios to be analysed and solved for a particular desired quantity.

2.2.2.4 Flow Balance Assumption

Often it is convenient to assume that a model satisfies the *flow balance* property which states that the number of customers arriving into a system is equal to the number of customers leaving the system over a period of time. Formally this can be stated as $A = C$ which also means that $\lambda = X$ i.e. the arrival rate is equal to the throughput of the system. If we assume the flow balance property holds and make use of Little's Law and the Forced Flow Law then we can calculate per station device utilisation for all transaction workloads. All product-form queueing network models follow the flow balance assumption. [23]

2.2.3 Model Outputs

Having looked at how queueing models are described and also some of the fundamental laws which govern the characteristics of these models, we now discuss the outputs which we wish to obtain in order for useful performance analysis to be performed. The output values depend on all of the model input values and as such represent *averages* over the system. Sometimes we will specify that an output value corresponds to a specific workload intensity, for example $X(\lambda)$ is the throughput for a transaction workload with arrival rate λ. [23]

For each performance measure we first give its definition in a single-class model and then extend the definition to multi-class models. For the latter, performance measures can be obtained either on a per-class basis or on an aggregate basis [23]. Note that additional outputs can be calculated however they will usually incur some extra computational cost over the measures presented below.

2.2.3.1 Throughput

As already mentioned the throughput is the rate of request completions. If the model is parameterised in terms of service demands, D_k, then we can calculate the system throughput, X. However, in order to calculate individual station throughputs the model must instead be parameterised in terms of V_k and S_k, since then the Forced Flow Law can be applied to calculate $X_k = V_k X$. [23]

For multi-class models the throughput at a station k is given by the sum of the per-class throughputs as k, $X_k = \sum_{c=1}^{C} X_{c,k}$. The throughput of the whole system could then be calculated by rearranging the Forced Flow Law as $X = X_k/V_k$ and calculating for some station k. Note that X_k and $X_{c,k}$ are only meaningful if the model is defined in terms of V_k and S_k, since the Forced Flow Law must be used to derive further information. [23]

2.2.3.2 Utilisation

Utilisation, denoted U_k, can be defined as the percentage of time station k is busy. Or for delay stations, the average number of customers being serviced there. The previously defined Utilisation Law, given by $U_k = X_k S_k$, can be used to calculate the utilisation. Note that this output is only defined on a per station basis. For multi-class models the per class measure $U_{c,k}$ can be obtained and the aggregate measure U_k is given by the sum of the per-class utilisations, $U_k = \sum_{c=1}^{C} U_{c,k}$. [23]

2.2.3.3 Residence and Response Time

The residence time at a station k, denoted R_k, is the total time a customer spends at station k, including both queue waiting and servicing time. If the model is defined in terms of V_k and S_k then we can say that S_k, the time a customer spends at station k in a single visit, is equal to R_k/V_k (i.e. the total time spent at station k divided by the total number of visits to station k). The average response time of a system, i.e. the average time between the arrival and departure of a customer, is equal to the sum of the residence times at each station, $R = \sum_{k=1}^{K} R_k$. For multi-class models the average residence time at station k can be computed by $R_k = (\sum_{c=1}^{C} R_{c,k} X_c)/X$ and the average response time can be calculated as $R = (\sum_{c=1}^{C} R_c X_c)/X$. Note that the per-class measures have been normalised by the relative throughput. [23]

2.2.3.4 Queue Length

The average queue length at a station k, denoted Q_k, includes all customers either waiting for or receiving service from the station. The number of waiting customers can be calculated by $Q_k - U_k$, as U_k essentially represents the average number of customers receiving service at station k. The average number of customers in a system, Q, can be calculated depending on the workload type: [23]

- *Batch workload:* $Q = N$
- *Transaction workload:* $Q = XR$ (via Little's Law)
- *Terminal workload:* $Q = N - XZ$ (via Little's Law and Response Time Law)

The average number of customers in any subsystem can be calculated as the product of the throughput and residence time of the subsystem, or alternatively by summing the queue lengths at the stations within the subsystem. For multi-class models, Q_k can be calculated as the sum of the per-class queue lengths, $Q_k = \sum_{c=1}^{C} Q_{c,k}$. A system-wide value for Q can then be calculated as $Q = \sum_{k=1}^{K} Q_k$. [23]

2.2.4 Limitations of Queueing Models

As with most attempts to model real-world systems, there are complexities and special circumstances inherent in computer systems which cannot be captured by queueing network models. That is to say, the inputs and outputs we have defined are not always sufficient to describe a system. Below we summarise some of the characteristics of computer systems which cannot be represented: [23]

- *Concurrent or parallel behaviours* - various scenarios, such as if a customer needs to be serviced by multiple resources simultaneously, cannot be expressed directly within queueing network models. Similarly, synchronisation points at which two customers must be processed in parallel cannot be modelled.
- *Non-deterministic behaviours* - in systems such as store-and-forward networks, in which data is sent to an intermediate node which may hold on to the data before forwarding it at a later time, the state of one station will affect how customers are processed at another station. This cannot be represented directly in our model. Another behaviour which cannot be expressed is so-called 'adaptive behaviour', for example when a dynamic routing protocol is employed so that customer routing decisions are made at run-time. Similarly priority scheduling which takes into account class dependent information cannot be modelled directly.
- *Memory constraints* - for each workload type we make implicit assumptions about the number of customers that can be held in memory. For example, in transaction workloads there is an implied assumption that any number of customers can be stored in memory, no matter how large. In batch workloads we assume there is a constant level of concurrency. In reality the number of concurrent jobs varies over time and is limited by memory constraints determined by the hardware. In particular, if there is an acute variability in the number of concurrent jobs then the system's performance may be degraded, but this cannot be represented in the queueing model.
- *Response time distributions* - the distribution of response times of a system cannot be calculated directly from the system queueing model at a reasonable cost.
- *Process forking* - due to the fact that in closed classes the number of customers must remain constant and in open classes the number of customers must be unbounded, it is not possible to explicitly model forking whereby a process spawns a sub-process, as is common within a Unix context.

Despite these inadequacies, product-form queueing network models are still often successful at representing the behaviour of complex computer systems since a lot of the relevant complexity is captured implicitly in the measurement data used as input for the model. In many cases queueing network models are also adequate for projecting the impact of architecture changes on a system, provided that the modification is representable by adjusting model inputs and that certain secondary characteristics of the modification can be ignored. However, in situations where detailed analysis is required to study how a modification will affect the implicit characteristics of a system, the standard queueing network model is not adequate. In these scenarios the product-form queueing network model can be augmented to include procedures that can compute revised estimates for the underlying characteristics we are interested in, allowing the desired accuracy to be achieved. So regardless of the apparent disparities which can arise between a system and its model representation, a sound and useful analysis can often still occur. [23]

2.3 Introduction to JMT

Everything we have described so far has been theoretical in nature, however the software suite Java Modelling Tools (JMT) is a free open-source toolset which implements some of the ideas we have discussed. The tools are intended to be used for system tuning, capacity planning, performance evaluation and workload characterization of distributed computer systems or any system that can be represented by a queueing model [18]. In this project the main tool we will be focussing on is called JMVA, in which network models can be described and evaluated using a variety of techniques. Queueing network models can be input via a wizard, which involves defining the stations, customer classes and service demands manually, or can be imported from another tool called JSIMengine, which allows models to be created graphically. Models can also be loaded and saved from XML files.

Once the model is specified it can be solved using a range of existing algorithms, including exact and approximate MVA methods, RECAL (Recursion by Chain), MoM (Method of Moments) and CoMoM (Class-Oriented Method of Moments). Once the model has been solved the results can be output both graphically and numerically. A useful feature of JMVA is the ability to perform 'what-if analysis' in which a control parameter, for example the network population, is swept over a specified range and the model solved for each value. This allows graphs to be drawn showing how the control parameter affects certain performance characteristics such as throughput, response time and utilisation. It also allows the model to be solved with several algorithms so that their accuracy can be compared. This feature was particularly useful for comparing the accuracy of the newly implemented bounding techniques.

In this project we will be contributing to JMT by integrating all solutions directly into the JMVA tool. Should this be successful, the primary developers of JMT will incorporate these updates into the next publicly available release.

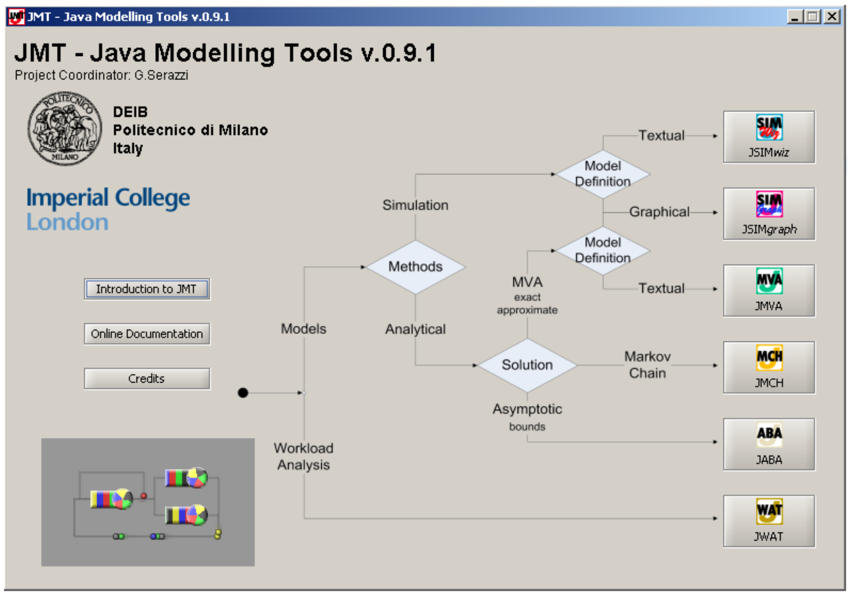
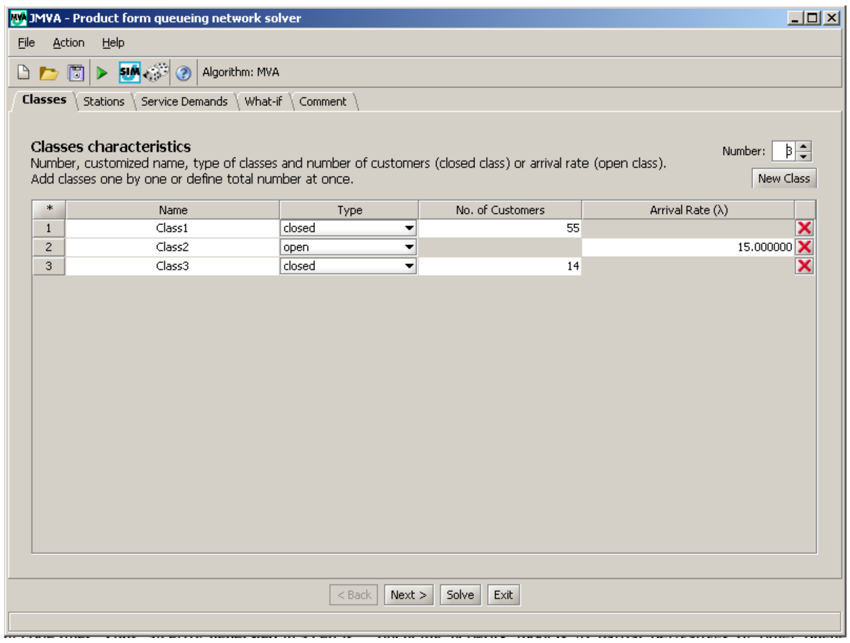


Figure 2.4: JMT tool selection screen.



Classes characteristics
Number, customized name, type of classes and number of customers (closed class) or arrival rate (open class).
Add classes one by one or define total number at once.

Number: 3

*	Name	Type	No. of Customers	Arrival Rate (λ)
1	Class1	closed	55	
2	Class2	open		15.000000
3	Class3	closed	14	

Figure 2.5: JMVA model input wizard.

2.4 Introduction to JCoMoM package

Some of the previous work completed by Imperial students, aimed at implementing more advanced, research-based algorithms, were placed in a separate project known as the JCoMoM package. For example, Bradshaw implemented a version of the CoMoM algorithm as discussed in [3]. The JCoMoM package was created in order to protect the rights to the code, and also separate these more experimental algorithms from the main JMT codebase. This package currently contains implementations for MoM, CoMoM, RECAL and basic Convolution methods. While separate from the JMT codebase, some of the algorithms in the JCoMoM package are still accessible directly from the JMVA interface, though the source code is not publicly available (only a JAR file for the JCoMoM package is downloadable). The JCoMoM package also has its own commandline interface, so the algorithms within the package can be used in isolation from the rest of JMT. In this project we will be adding the Tree Convolution and Tree MVA algorithms to this package while also allowing these algorithms to be accessed from the JMVA interface.

Chapter 3

Queueing Network Analysis Techniques

Now that we have fully defined the queueing models we will be working with and studied their limitations, we take a look at some of the analysis algorithms which can be used to evaluate these models.

3.1 Performance Bounding Techniques

A simple but often useful technique for evaluating queueing networks is bounding analysis [23, 6]. Bounding techniques aim to calculate the limiting bounds of system performance measures, such as throughput and response time, as a function of workload intensity. The main advantage of these techniques is that they are highly computationally efficient and are therefore ideal for preliminary analysis of a system, before more complex and expensive approaches are used. For certain applications, for example systems using real-time self-optimisation techniques, it must be possible to analyse many thousands of configuration options quickly and accurately. For this purpose exact methods are much too slow. Local iterative approximations [11] are substantially more efficient than exact methods but still cannot match the speed of single-step bounding techniques [12, 5]. Having said this, the accuracy of such bounding techniques may be significantly worse [6]. One of the goals of this project is to tackle this problem by implementing a bounding technique which is both fast and accurate, namely *Geometric Bounds* as described in [6].

Another useful application of bounding techniques is for studying the influence of bottleneck stations. Secondary bottlenecks can be identified and used to analyse how the system may be modified in order to reduce the load placed on the primary bottleneck. Bounding methods can also be used for studying the effects of possible system modifications. Often a number of possible modifications can be grouped together and a single bounding analysis pass done to provide feedback about the possibilities, providing a very low cost analysis. An additional benefit of these methods is that the actual development and implementation of bounding techniques often provides useful insights about the core elements of a system which influence performance. [23]

In the following subsections we discuss various single-step bounding techniques, with a focus on defining bounds for system throughput and response time. For batch and terminal workloads the bounds reflect the maximum and minimum possible throughputs and response times in terms of N, the number of customers in the system. For transaction workloads the maximum system throughput specifies the maximum customer arrival rate that the system is able to deal with, while the bounds on response time indicate the range of possible response times in terms of λ, the customer arrival rate. *Optimistic bounds* are often used to refer to the bounds providing the best case performance of a system, for example maximal throughput (upper bound) and minimal response time (lower bound). *Pessimistic bounds* are the inverse, indicating the worst case performance. Closely related to the concept of optimistic bounds is the notion of system power [19] which is defined as X/R i.e. the ratio between throughput and response time, either as a system-wide measure or per customer class. Simply put the optimal performance of a system occurs when throughput is maximized and response time is minimised, which corresponds to when the system power is maximal [9].

Bounds for other measures can be derived by using the aforementioned system laws. Per station bounds on utilisation and throughput can also be calculated by applying the methods described in the following sections and then applying either the Utilisation Law or the Forced Flow Law respectively. It is also worth noting that the application of bounding methods is usually confined to single-class models. Multi-class variations exist but are not widely used since these methods often encroach on the main advantage of bounding analysis, which is its simplicity. Having said this, we implement multi-class ABAs and BJBs as part of this project for completeness. The theory for multi-class GBs has not yet been developed.

3.1.1 Asymptotic Bounds

For single-class queueing networks, asymptotic bounding analysis [24] outputs both optimistic and pessimistic bounds on system throughput and response time. The bounds are obtained by examining the asymptotic extremes when the system is under light and heavy loads. In order for these bounds to be viable it must be assumed that stations are load independent, that is the service demand for a customer at each station does not depend on the number of customers in the system, and also that the service demand for a particular customer does not depend on the locations of other customers. [23]

Transaction Workloads
As briefly suggested earlier, in transaction workloads there is a possibility of system saturation whereby the arrival rate of new customers is too great for the system to be able to process incoming customers in a timely fashion, leading to an ever increasing backlog of waiting jobs. The upper throughput bound represents the minimum arrival rate, λ_{sat}, at which the system becomes saturated. In order to acquire this bound we note that as long as none of the individual stations in the system are saturated, then an increased arrival rate can be accommodated. A station is said to be saturated if it has a utilisation of 100%. However, as soon as a single station becomes saturated, the entire system cannot accommodate an increase in arrival rate and is therefore also saturated. The station which becomes saturated at the lowest λ value is the system's bottleneck i.e. the station with highest service demand. If we let max be the index of the bottleneck station then mathematically we can derive: [23]

$$\lambda_{sat} = \frac{1}{D_{max}} \tag{3.1}$$

Proof.
$U_k = X_k S_k$ (Utilisation Law, for each station k)
Since $X_k = \lambda * V_k$ and $D_k = V_k S_k$ we can derive:
$U_k = \lambda D_k$ (D_k is service demand at station k)
So, $U_{max}(\lambda) = \lambda D_{max} \leq 1$ (since U is always less than or equal to 1, i.e. 100% utilisation)
In this case we know that U = 1 since the station is a bottleneck, therefore: $\lambda_{sat} = 1/D_{max}$
□

So if λ is greater than or equal to $1/D_{max}$ the system is saturated.

The response time bounds for transaction workloads can be computed by considering the two possible extremes. Since the system becomes unstable if $\lambda \geq \lambda_{sat}$, we consider only the instances where the arrival rate is less than λ_{sat}. In the best case no customer ever has to wait for a system resource to become free and so the response time per customer is the sum of its service demands. In the worst case n customers arrive simultaneously every n/λ time units (so the arrival rate $= n/(n/\lambda) = \lambda$). For any upper bound on response times we chose, we can always pick an n such that the bound will be exceeded. Therefore, there is no upper (pessimistic) bound on response times for transaction workloads. [23]

Batch and Terminal Workloads
For batch and terminal workloads the results are more promising. Considering first the heavy load case for terminal workloads, we note that as N increases, the station utilisations increase but never exceed 1. Therefore we can say $U_k = X(N)D_k \leq 1$ (by the Utilisation Law) and using the same line of reasoning as for transaction workloads we obtain: [23]

$$X(N) \leq \frac{1}{D_{max}} \text{ (where } max \text{ is the first station to become saturated)} \tag{3.2}$$

For the light load case, we consider first the case of one customer in the system (N=1). Since the time of each interaction with the system is equal the sum of the average length of service ($D = \sum_{k=1}^{K} D_k$) and the average think time Z, the system throughput is given by $1/(D+Z)$. As N increases we see there are two bounding cases. Firstly, when each added customer is not delayed at all by other customers the individual customer throughput is given by $1/(D+Z)$, since D time units are spent in service and Z time units are spent thinking, giving an overall system throughput of $N/(D+Z)$. Secondly, when each added customer has to queue behind all other customers in the system, the throughput of a single customer is given by $1/(ND+Z)$, since $(N-1)D$ time units are spent queueing, D time units are spent in service and Z time units are spent thinking, giving a system throughput of $N/(ND+Z)$. To summarise, for terminal workloads we have throughput asymptotic bounds: [23]

$$\frac{N}{ND+Z} \leq X(N) \leq \min(\frac{1}{D_{max}}, \frac{N}{D+Z}) \tag{3.3}$$

It can be observed that the optimistic bound consists of two elements. The first, $1/D_{max}$, holds when the system is under heavy load, while the second, $N/(D+Z)$, holds under light load. There is a crossover point given by $(D+Z)/D_{max}$ at which these formulas give the same value. Below this point the light load bound holds and above it the heavy load bound holds. [23]

The bounds on response time for terminal workloads can be obtained by applying Little's law to get: [23]

$$max(D, ND_{max} - Z) \leq R(N) \leq ND \tag{3.4}$$

Proof.
$\frac{N}{ND+Z} \leq X(N) \leq \min(\frac{1}{D_{max}}, \frac{N}{(D+Z)})$ (throughput bounds)
From Little's Law/Response Time Law we get $X(N) = \frac{N}{R(N)+Z}$, so
$\frac{N}{ND+Z} \leq \frac{N}{R(N)+Z} \leq \min(\frac{1}{D_{max}}, \frac{N}{D+Z})$
$max(D_{max}, \frac{D+Z}{N}) \leq \frac{R(N)+Z}{N} \leq \frac{ND+Z}{N}$ (by inverting)
$max(D, ND_{max} - Z) \leq R(N) \leq ND$ (by rearranging, note $D = \sum_{k=1}^{K} D_k$) $\square$

Note that the asymptotic bounds for batch workloads are obtained simply by setting Z = 0.

In summary, asymptotic bounds are derived by considering the extreme (best and worst) cases under which a system might operate. Their main advantage is their simplicity and low computational cost due to the fact that the calculations are independent of both the number of stations in the system and customer class information. The linearity of the bounding equations (except for the the lower throughput bound for terminal workloads) means that only a small number of arithmetic operations are needed to compute most of the bounds once D and D_{max} are known. [23]

3.1.2 Balanced Job Bounds

Though incurring slightly higher computational costs than their asymptotic counterparts, Balanced Job Bounds (BJBs) [20, 37] provide more precise outputs. The bounds provided by BJB analysis are based upon the study of systems in which the service demands are *balanced*, i.e. the demands at every station are identical. This is because balanced systems exhibit various properties which can be utilized in order to tighten the asymptotic bounds previously outlined. [23]

Considering a balanced batch workload, if we take D_{min}, D_{max} and D_{avg} to represent the minimum, maximum, and average service demands in the system, and K as the number of stations in the network, we can bound the throughput of the system as follows: [23]

$$\frac{N}{N+K-1} \times \frac{1}{D_{max}} \leq X(N) \leq \frac{N}{N+K-1} \times \frac{1}{D_{min}} \tag{3.5}$$

Proof.
$U_k(N) = \frac{N}{N+K-1}$ (utilisation at every station k for batch workloads)
$X(N) = \frac{U_k}{D_k} = \frac{N}{N+K-1} \times \frac{1}{D_k}$ (by Utilisation Law)
Equation (3.5) follows from this since:
- LHS is the throughput of the balanced system where every service demand is set to D_{max}, so the throughput will be the lowest possible.
- RHS is the throughput of the balanced system where every service demand is set to D_{min}, so the throughput will be the highest possible. □

In order to tighten the bounds we can restrict both D_{max} and total demand, $D = \sum_{k=1}^{K} D_k$. D can be restricted since out of every possible system with total service demand D, the one in which all service demands are equal is the one with highest throughput. Therefore we can set all station service demands to D_{ave} and derive the throughput optimistic bound as: [23]

$$X(N) \leq \frac{N}{N+K-1} \times \frac{1}{D_{ave}} = \frac{N}{D+(N-1)D_{ave}} \tag{3.6}$$

Now we consider the lower bound. For all systems with a total service demand D and maximum service demand D_{max}, the system which has D/D_{max} stations each with service demand D_{max} and the rest of the stations having service demand 0, is the system with the lowest throughput. Therefore the throughput pessimistic bound is given by: [23]

$$\frac{N}{N+\frac{D}{D_{max}}-1} \times \frac{1}{D_{max}} = \frac{N}{D+(N-1)D_{max}} \leq X(N) \tag{3.7}$$

Taking into account that the upper balanced job bound on throughput intersects the upper asymptotic bound for heavily loaded systems, the throughput bounds can be summarised as: [23]

$$\frac{N}{D+(N-1)D_{max}} \leq X(N) \leq \min\left(\frac{1}{D_{max}}, \frac{N}{D+(N-1)D_{ave}}\right) \tag{3.8}$$

As before, Little's Law can be applied to derive the response time bounds: [23]

$$\max(ND_{max}, D + (N-1)D_{ave}) \leq R(N) \leq D + (N-1)D_{max} \tag{3.9}$$

A approach similar to that used for ABs can be used to derive BJB bounds for terminal and transaction workloads.

3.1.3 Geometric Bounds

Geometric bounds (GBs) [6] are a much more recently developed method than any of the other bounds considered. Compared to BJBs, geometric bounds attain higher levels of accuracy but maintain similar computational cost. The worst case bounding error for BJBs is usually within the range 15-35%, however GBs lower this to 5-13%. Additionally, solutions are typically closer to the global optimum than other bounding techniques. In fact the accuracy of GB techniques approaches that of many iterative MVA-based estimation methods, but at a much lower computational cost. In particular GBs have been shown to give highly accurate results in unbalanced networks with bottlenecks, which is an important consideration when analysing real systems. For these reasons the implementation of GBs will form a part of this project. [6]

Geometric bounds are obtained by representing station queue lengths as a geometric sequence of terms related to station utilisations. In order to derive this sequence we start from one of the MVA equations: [6]

$$Q_i(N) = X(N)R_i(N) = U_i(N)(1 + Q_i(N-1)), 1 \leq i \leq K \tag{3.10}$$

where Q_i is the average queue length at station i, $X(N)$ is the mean system throughput, $R_i(N)$ is the mean response time of station i, and U_i is the utilisation at station i.

This equation can be recursively expanded to give: [6]

$$\begin{aligned} Q_i(N) = &\{U_i(N)\} \\ &+ \{U_i(N)U_i(N-1)\} \\ &+ \{U_i(N)U_i(N-1)U_i(N-2)\} \\ &+ \ldots + \{U_i(N)U_i(N-1)U_i(N-2)\ldots U_i(2)U_i(1)\} \end{aligned}$$

We notice that this has the same structure as a geometric sum which can be calculated non-iteratively as: [6]

$$y + y^2 + \ldots + y^N = \frac{y}{1-y} - \frac{y^{N+1}}{1-y} \tag{3.11}$$

This is a key step in removing the iterative nature of MVA, allowing us to compute bounds on queue length at a reasonable cost.

Queue Length Bounds

Using this idea bounds on queue length can be derived as: [6]

$$Q^-_{i,GB}(N) = \begin{cases} \dfrac{yi(N)}{1-y_i(N)} - \dfrac{y_i(N)^{N+1}}{1-y_i(N)} & if D_i < D_{max} \\ \dfrac{1}{m_{max}}(N - ZX^+ - \displaystyle\sum_{k=D_k<D_{max}} Q^+_{k,GB}(N)) & if D_i = D_{max} \end{cases} \tag{3.12}$$

$$Q^+_{i,GB}(N) = \begin{cases} \dfrac{Yi(N)}{1-Y_i(N)} - \dfrac{Y_i(N)^{N+1}}{1-Y_i(N)} & if L_i < L_{max} \\ \dfrac{1}{m_{max}}(N - ZX^- - \displaystyle\sum_{k=D_k}^{D_{max}} Q^-_{k,GB}(N)) & if L_i = L_{max} \end{cases} \tag{3.13}$$

where $Q^-_{i,GB}(N)$ and $Q^+_{i,GB}(N)$ define the the lower and upper queue length bounds respectively. These equations hold for any for any X^+ and X^- s.t. $X(N) \leq X^+ \leq X_{max}$ and $0 \leq X^- \leq X(N)$. m_{max} is the number of queues with service demand D_{max}. $y_i(N)$ and $Y_i(N)$ give the ratio of the geometric sum where $y_i(N) = D_iN/(Z + D + D_{max}N)$ and $Y_i(N) = D_iX^+$. Additionally it should be noted that $D = \sum_{i=1}^{K} D_i$. The full derivations for these bounds can be found in [6].

Throughput Bounds
In order to use the bounds on queue length to derive throughput bounds an exact formula for X(N) must be obtained. This time we consider a different MVA equation given by: [6]

$$X(N) = N/(Z + R(N)) \tag{3.14}$$

To obtain an accurate approximation for X(N) we also take into account the population constraint, which states the the first moments of queue lengths are conserved in closed models: [6, 20]

$$\sum_{i=1}^{K} Q_i(N-1) = N - 1 - ZX(N-1) \tag{3.15}$$

By considering only the queues with service demand $D_i = D_{max}$ and using the previous two equations, an exact formula for X(N) can be derived: [6]

$$X(N) = N/[Z + D + D_{max}(N - 1 - ZX(N-1)) - D(N)] \tag{3.16}$$

where $D(N) = \sum_{i=1}^{M}(D_{max} - D_i)Q_i(N-1)$. In fact by comparing this formula to the lower BJB bound on throughput, and setting $X(N-1)$ and $D(N)$ to their minimum values, we can see that we have derived a more general method for obtaining throughput bounds [6].

By replacing $X(N-1)$ and $D(N)$ by appropriate bounds, we obtain the bounds on throughput $X(N)$ as: [6]

$$X^-_{i,GB}(N) = N/[Z + D + D_{max}(N - 1 - ZX^-) - \sum_{i=1}^{K}(D_{max} - D_i)Q^-_{i,GB}(N-1)] \tag{3.17}$$

$$X^+_{i,GB}(N) = N/[Z + D + D_{max}(N - 1 - ZX^+) - \sum_{i=1}^{K}(D_{max} - D_i)Q^+_{i,GB}(N-1)] \tag{3.18}$$

for any X^+ and X^- s.t. $X(N-1) \leq X^+ \leq X_{max}$ and $0 \leq X^- \leq X(N-1)$.

Both bounds on throughput and queue length using the GB approach have computational costs which grow as O(K) in both time and space, so their calculation is independent of the number of customers in the system. The GB method can be generalised in order to obtain accurate results for closed fork-join networks [35] which are used frequently in practice, for example in the optimisation of parallel systems. Closed fork-join networks are amenable to approximate evaluation and hence the usefulness of GBs goes beyond the analysis of product-form queueing networks. [6]

One of the first focuses of this project is to implement Geometric Bounds and assess their performance over a range of problem instances. We also aim to show that GBs provide higher accuracy outputs than ABs and BJBs, particularly over unbalanced systems.

3.2 Iterative Solution Techniques

In this section we move away from looking at single-step performance bounding techniques and instead focus on iterative techniques which can be used to obtain accurate system performance measures given an input model. The specific methods used depend on whether a model is open or closed.

3.2.1 Solution of Open Models

For open single-class models the customer arrival rate is given as an input, λ. Similarly for open multi-class models each customer class has an arrival rate λ_c, so we represent all the arrival rates in a vector $\lambda = (\lambda_1, \ldots, \lambda_c)$. For both types of open model we are essentially given the throughput as an input (seen as the customer arrival rate λ is equivalent to system throughput by the Forced Flow Law). This makes the solution methods for open models fairly simple. The table below shows the various performance measures which can be computed:

Measure	**Single-class Formula**	**Multi-class Formula**
Processing Capacity	$\lambda_{sat} = \frac{1}{\max\limits_k(D_k)} = \frac{1}{D_{max}}$	$\max\limits_k(\sum\limits_{c=1}^{C} \lambda_c D_{c,k}) < 1$
Device Throughput	$X_k(\lambda) = \lambda V_k$	$X_{c,k}(\lambda) = \lambda_c V_{c,k}$
Device Utilisation	$U_k(\lambda) = X_k(\lambda) S_k = \lambda D_k$	$U_{c,k}(\lambda) = X_{c,k}(\lambda) S_{c,k} = \lambda_c D_{c,k}$
Residence Time	$R_k(\lambda) = \begin{cases} V_k S_k = D_k & \text{(Delay)} \\ \frac{D_k}{1-U_k(\lambda)} & \text{(Queueing)} \end{cases}$	$R_{c,k}(\lambda) = \begin{cases} D_{c,k} & \text{(Delay)} \\ \frac{D_{c,k}}{1-\sum\limits_{j=1}^{C} U_{j,k}(\lambda)} & \text{(Queueing)} \end{cases}$
Queue Length	$Q_k(\lambda) = \lambda R_k(\lambda)$ (Little's Law). Therefore, $Q_k(\lambda) = \begin{cases} U_k(\lambda) & \text{(Delay)} \\ \frac{U_k(\lambda)}{1-U_k(\lambda)} & \text{(Queueing)} \end{cases}$	$Q_k((\lambda) = \lambda_c R_{c,k}((\lambda)$ (Little's Law). Therefore, $Q_k(\lambda) = \begin{cases} U_{c,k}(\lambda) & \text{(Delay)} \\ \frac{U_k(\lambda)}{1-\sum\limits_{j=1}^{C} U_{j,k}(\lambda)} & \text{(Queueing)} \end{cases}$
System response time	$R(\lambda) = \sum\limits_{k=1}^{K} R_k(\lambda)$	$R_c(\lambda) = \sum\limits_{k=1}^{K} R_{c,k}(\lambda)$
Average number of customers in system	$Q(\lambda) = \lambda R(\lambda) = \sum\limits_{k=1}^{K} Q_k(\lambda)$	$Q_c(\lambda) = \lambda_c R_c(\lambda) = \sum\limits_{k=1}^{K} Q_{c,k}(\lambda)$

Table 3.1: Exact solutions for single and multi-class open models [23].

Note that any formula in the table can be expanded out by substituting in formulas higher in the table. The processing capacity is the maximum arrival rate the system can cope with before it becomes saturated, λ_{sat}. All other formulae in the table assume that $\lambda < \lambda_{sat}$. In the multi-class case we check that no station is saturated as a result of the combined load placed upon the system by all customer classes. The formulas are derived using the laws we have already discussed, for detailed derivations [23, Chapter 6-7] can be consulted.

Since these equations are all fairly easy to compute and no real improvement can be achieved through optimisation, the exact solution of open models is not the primary focus of this project, however we mention it for completeness.

3.2.2 Solution of Closed Models

The more interesting case is the solution of closed models since the throughput is not known in advance, making the calculation of performance metrics considerably more challenging. In the single-class case, if a product-form queueing network has K stations and a population of N customers then the network can be described as $\mathbf{n} = (n_1, n_2, \ldots, n_K)$ where n_i is the number of customers at station i (i.e. summing the terms inside the vector gives N) [4]. If we additionally assume that the service time at all stations is given by a distributed random variable $1/u_i$, then: [16]

$$P(n_1, \ldots, n_K) = \frac{\prod_{i=1}^{K} (X_i)^{ni}}{G(N)} \tag{3.19}$$

where the set of X_i are the solutions to the equations $u_j X_j = \sum_{i=1}^{M} u_i X_i p_{i,j}, 1 \leq j \leq M$ given that $p_{i,j}$ is the probability that a customer completing service at station i will proceed to station j [4]. G(N) is a normalisation constant which ensures all of the $P(n_1, \ldots, n_K)$ sum to 1. This notation can be extended for the multi-class case which we discuss in section 3.2.4.

Various approaches have been tried in order to calculate the solution to this product-form equation. One approach is to compute G(N) by adding together the product terms over the entire state space. The first practical method for achieving this, known as the Convolution algorithm [4], was first published in 1973 by Buzen. Later developments saw this approach extended to work for multi-class models [10, 26]. At the time one of the problems with the Convolution algorithm was that it did not guard against numerical instability [34], however due to the rapid advancement of computing hardware these problems are no longer of great concern. A few years later a technique called Mean Value Analysis (MVA), which avoids the need to compute G(N), was developed and refined by Reiser and Lavenberg [28, 29]. It was also shown that MVA is as general as Buzen's Convolution algorithm [30].

The main problem with these two techniques is that they become computationally infeasible when large numbers of customer classes are introduced, which is often required for the analysis of real-world systems. In order to avoid this problem, several approximate versions of the MVA algorithm were proposed, each using various heuristics to reduce the complexity of the calculations. The implementation of these techniques was the focus of an earlier project [13]. However, after the development of the Convolution and MVA algorithms, various advanced versions were also proposed. These more advanced versions of the original algorithms still provide exact solutions, however they also allow the efficient solution of sparse networks which involve a large number of customer classes. The two we focus on in this project are the Tree Convolution [21] and Tree MVA [34] algorithms.

3.2.3 Convolution

The Convolution algorithm [4] computes the normalisation constant G(N) in order to obtain useful performance measures from a system. We first discuss Convolution for single-class models as it was first outlined by Buzen, before moving onto more complex multi-class Convolution techniques. Single-class sequential Convolution takes an iterative approach in order to compute the set of values G(1), G(2), . . . , G(N) using NK multiplications and NK additions, where N is the model population and K is the number of stations [4].

The algorithm is based on an auxiliary function: [4]

$$g(n,k) = \sum_{n \in S(n,k)} \prod_{i=1}^{k} (X_i)^{ni} \tag{3.20}$$

$$\text{where } S(N,K) = (n_1, n_2, \dots, n_K) \text{ s.t. } \sum_{i=1}^{K} n_i = N \text{ and } \forall_i n_i \geq 0 \tag{3.21}$$

Since G(N) is defined so that all the $P(n_1, \dots, n_K)$ sum to 1 (see equation (3.19)), we have that $G(N) \equiv g(N,K)$. In fact generally we can say, $g(n,K) \equiv G(n)$ for all $n = 0, 1, \dots, N$. [4]

In order to make $g(n,k)$ usable within an algorithm, some initial conditions and an iterative relationship can be derived from (3.20): [4]

Initial conditions

$$g(n,1) = (X_1)^n \qquad \text{for } n = 0, 1, \dots, N$$
$$g(0,k) = 1 \qquad \text{for } k = 1, 2, \dots, K$$

Iterative Relationship

$$\begin{aligned} g(n,k) &= \sum_{\substack{n \in S(n,k) \\ \& n_k = 0}} \prod_{i=1}^{k} (X_i)^{n_i} + \sum_{\substack{n \in S(n,k) \\ \& n_k > 0}} \prod_{i=1}^{k} (X_i)^{n_i} \\ &= g(n, k-1) + X_k g(n-1, k) \end{aligned}$$

Using these relationships a simple iterative algorithm can be defined in order to compute G(1), G(2), . . . , G(N), assuming that the X_k values are calculated in advance and passed in as an array. The pseudocode for this algorithm is shown in Algorithm 1 below.

Algorithm 1 Pseudocode showing simple iterative Convolution algorithm [4].

```
/** Computes the normalisation constants for the last column in the table.
  * @param X Per station throughputs.
  * @return Array containing normalisation constants g(1)...g(N).
  */
function ComputeNormalisationConstants(X) {
  C[0] = 1;
  C[n] = 0 forall 1 <= n <= N;

  for (k=1:K) {
    for (n=1:N) {
      C[n] = X[k] * C[n-1];
    }
  }

  return C;
}
```

The algorithm can also be visualised in a tabular format:

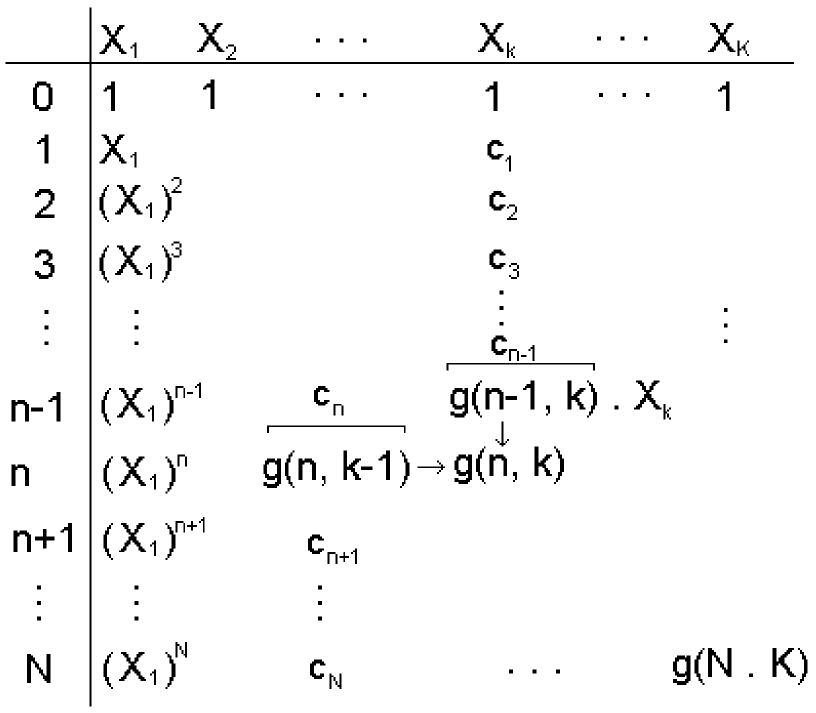


Figure 3.1: Visualisation of Convolution algorithm. [4]

This diagram attempts to show that any $g(n, k)$ can be computed by summing the value to the left ($g(n, k-1)$) with the value immediately above multiplied by the column variable ($g(n-1, k) \times X_k$). The algorithm therefore proceeds in a column-wise fashion, computing all the values in the X_1 column first, before computing values for the other columns in sequence. Note that when computing the first column it is assumed that there is a column of zeros immediately to the left (i.e. for the first column $g(n, k-1) = g(n, 0) = 0$ for all $0 \leq n \leq N$). The final goal of the algorithm is to compute $g(N, K)$ in the bottom-right corner since, as we already established, this is equal to G(N).

The variables $C_1 \dots C_N$ correspond to the values in the storage array presented in the algorithm code. Once the algorithm has completed the values in C will be equal to the final computed values for the right-most column in the table, which correspond to G(1), G(2), ..., G(N). Since the computation of each value in the table requires one addition and one multiplication and the table is of size NK, we can conclude that $2NK$ arithmetic operations are required for the computation of G(1), G(2), ..., G(N). In terms of space complexity we only need to store N values at any time, so long as the algorithm progresses in a column-wise fashion as described. [4]

3.2.4 Multi-class sequential Convolution

The Convolution algorithm presented so far is limited to single-class queueing systems. In this section we introduce a mutli-class approach which is required before we can discuss the Tree Convolution algorithm. In order to do this we must redefine the product-form equation (3.19) in order to take into account different customer classes.

If we consider a product-form queueing network with K stations and C customer classes, we can define a population vector $\mathbf{N} = (N_1, N_2, \dots, N_C)$ where N_c is the number of customers of class c. The normalisation constant can then be redefined as $G(\mathbf{N})$. If we let n_{kc} be the number of class c customers at station k, we can also define the network state as a vector: [21]

$$\mathbf{n} = (\mathbf{n_1}, \mathbf{n_2}, \ldots, \mathbf{n_k})$$
$$\text{where } \mathbf{n}_k = (n_{k1}, n_{k2}, \ldots, n_{kC}), \text{ for all } 1 \leq k \leq K$$

The product-form equation can now be redefined as: [21]

$$P(\mathbf{n}) = \frac{\prod_{k=1}^{K} p_k(\mathbf{n_k})}{G(\mathbf{N})} \tag{3.22}$$

where

$$p_k(\mathbf{n_k}) = \left(\prod_{i=1}^{n_k} \frac{1}{u_k(i)} \right) n_k! \prod_{c=1}^{C} \frac{D_{kc}^{n_{kc}}}{n_{kc}!} \tag{3.23}$$

where
- n_k is the summation of all terms in the vector $\mathbf{n_k}$ (i.e. $n_k = \sum_{c=1}^{C} n_{kc}$),
- $D_{kc} = S_{kc}V_{kc}$ is the service demand at station k for class c as before, where S_{kc} is the mean service time of a class c customer at station k and V_{kc} is the arrival rate of class c customers to station k,
- $u_k(i)$ is the service rate of station k when there are i customers in its queue. If the station is load-independent then $\forall_i u_k(i) = 1$.

p_k can be thought of as a function for $k = 1, 2, \ldots, K$, where the input is a C-dimensional array indexed between $\mathbf{0}$ and $\mathbf{N}$, where $\mathbf{0}$ is a vector of size C with each element set to 0. The convolution of two of these functions, for example p_1 and p_2, is defined as: [21]

$$\begin{aligned} g_2(\mathbf{i}) &= \sum_{j_1=0}^{i_1} \cdots \sum_{j_C=0}^{i_C} p_1(\mathbf{j}) p_2(\mathbf{i} - \mathbf{j}) \text{ for } \mathbf{0} \leq \mathbf{i} \leq \mathbf{N} \\ &= p_1 * p_2 \equiv p_2 * p_1 \text{ (shorthand notation)} \end{aligned} \tag{3.24}$$

Using this notation we can define: [21]

$$\begin{aligned} g_1 &= p_1 \\ g_k &= g_{k-1} * p_k \text{ for } 2 \leq k \leq K \end{aligned} \tag{3.25}$$

The values stored in the array g_k are the normalisation constants for all population vectors between $\mathbf{0}$ and $\mathbf{N}$, therefore $g_K(\mathbf{N}) = G(\mathbf{N})$. In fact the equations (3.24) and (3.25) define the Convolution algorithm for multi-class models as was first derived in [10, 26] in 1975.

For a model consisting of load independent stations the equations can be combined to get: [21]

$$g_k(\mathbf{i}) = g_{k-1}(\mathbf{i}) + \sum_{c=1}^{C} D_{kc} g_k(\mathbf{i} - \mathbf{1_c}) \text{ for } \mathbf{0} \leq \mathbf{i} \leq \mathbf{N} \tag{3.26}$$

where $\mathbf{1_c}$ is a vector of size c, with all values set to 0 apart from the cth value which is set to 1.

The complexity of the multi-class sequential Convolution algorithm is outlined in the following table:

	General case	**Load independent case**
Time	$O((K-1)\prod_{c=1}^{C}(N_c+1)(N_c+2)/2)$	$O(KC\prod_{c=1}^{C}(N_c+1))$
Space	$O(2\prod_{c=1}^{C}(N_c+1))$	$O(\prod_{c=1}^{C}(N_c+1))$

Table 3.2: Time and space complexity of the multi-class sequential Convolution algorithm for the general and load independent case. In reality, each unit of space is one array location and each unit in time is about the execution time of one addition and one multiplication [21].

Practical Implementation
In practice the multi-class sequential algorithm can use the following expression to compute the normalization constants: [26, 7]

$$G(\mathbf{1_k}, \mathbf{N}) = G(\mathbf{N}) + \sum_{c=1}^{C} D_{kc} G(\mathbf{1_k}, \mathbf{N} - \mathbf{1_c}) \tag{3.27}$$

where $1 \leq k \leq K$, $\mathbf{1_i}$ is a vector containing all zeros apart from a one in the ith position, and $G(\mathbf{1_k}, \mathbf{N} - \mathbf{1_c})$ is the normalisation constant for the model with a class c customer removed and a replica of station k added. $G(\mathbf{N})$ is given by $G(\mathbf{0}, \mathbf{N})$.

3.2.5 Mean Value Analysis (MVA)

Mean Value Analysis [28, 29] is a recursive technique for calculating exact outputs from product-form queueing models which describe the long-term behaviour of the system. In terms of mathematical queueing theory it relies upon the *arrival theorem* which states that when a new customer arrives into the system, that customer observes the system as being in an equilibrium state (for the system without the new customer). For example, in a closed system with N customers, the new customer observes the system to be in a steady state for $N-1$ customers.

The MVA algorithm can be generalised to work for both single and multi-class models, though the latter is significantly more computationally expensive as we shall see. For multi-class systems we recall that the model contains C customer classes and the workload intensity can be defined by the vector $\mathbf{N} = (N_1, \ldots, N_C)$ where N_c is the number of customers of class c. In the following sections we present MVA for both types of models in parallel so that the two approaches can be compared directly.

The MVA algorithm for both single and multi-class models is based on three equations which can be derived from the laws we have already defined: [23]

1. *Station residence time equations:*

Single-class:
$$R_k(N) = \begin{cases} D_k & \text{(delay stations)} \\ D_k(1 + A_k(N)) & \text{(queueing stations)} \end{cases} \tag{3.28}$$

Multi-class:
$$R_{c,k}(\mathbf{N}) = \begin{cases} D_{c,k} & \text{(delay stations)} \\ D_{c,k}(1 + A_{c,k}(\mathbf{N})) & \text{(queueing stations)} \end{cases} \tag{3.29}$$

(when a new customer arrives $A_k(N)$ is the average number of customers seen at station k and, similarly, $A_{c,k}(\mathbf{N})$ is the average queue length at station k seen by an arriving customer of class c).

2. *Throughput equations* - obtained via Little's Law applied to the whole queueing network:

Single-class:
$$X(N) = N/(Z + \sum_{k=1}^{K} R_k(N)) \tag{3.30}$$

Multi-class:
$$X_c(\mathbf{N}) = N_c/(Z_c + \sum_{k=1}^{K} R_{c,k}(\mathbf{N})) \tag{3.31}$$

   (When there are N customers in the network, $X(N)$ is system throughput and $R_k(N)$ is residence time at station k. Similar definitions apply for the multi-class case).

3. *Per station queue length equations* - via Little's Law applied to individual service stations:

$$\text{Single-class:} \qquad Q_k(N) = X(N)R_k(N) \tag{3.32}$$
$$\text{Multi-class:} \qquad Q_{c,k}(\mathbf{N}) = X_c(\mathbf{N})R_{c,k}(\mathbf{N}) \tag{3.33}$$

(Note that (2) and (3) can be applied directly if $R_k(N)$ or $R_{c,k}(\mathbf{N})$ is known from inputs, without having to use (1).)

In order to use these algorithms in practice, we must first obtain $A_k(N)$ for all k in single-class models and $A_{c,k}(\mathbf{N})$ for all c and k in multi-class models. Considering the single-class case for simplicity, once we have this set of values we can apply (1) (using input values for D_k) to find the set of R_k values. We can then substitute the R_k values into (2) to get $X(N)$. Finally we substitute the $X(N)$ and R_k values into (3) to compute the Q_k values.

For open models we can make the assumption that $A_k(N)$, the arrival instant queue lengths, are equal to $Q_k(N)$, the time average queue lengths (in fact this is how the open model residence time formula we presented in Figure 3.1 is derived) [23]. For closed models this assumption does not hold since the computation of A_k takes into account that a newly arriving customer is not in the queue at station k, while Q_k is computed over random snapshots in time, meaning that all customers could be in the queue [23]. So for closed models we must compute $A_k(N)$ ourselves.

The solution method works by calculating the values of A_k (single-class) or $A_{c,k}$ (multi-class) exactly and then applying the three MVA equations. In fact one of the reasons closed product-form queueing networks are so widely used in performance analysis is precisely because $A_k(N)$ or $A_{c,k}(\mathbf{N})$ can be calculated simply by looking at time averaged queue lengths, Q_k. $A_k(N)$, the length of a queue seen when a new customer arrives at station k when there are N customers in the system, is equal to the time averaged queue length at the same station with one less customer in the system. Mathematically this is: [23]

$$A_k(N) = Q_k(N-1) \tag{3.34}$$
$$\text{or } A_{c,k}(\mathbf{N}) = Q_k(\mathbf{N}-\mathbf{1_c}) \tag{3.35}$$

The second equation is for multi-class models, where $\mathbf{N} - \mathbf{1}_c$ is the system with $\mathbf{N}$ customers with one class c customer removed. Note that for multi-class models: $Q_k(\mathbf{N}) = \sum_{c=1}^{C} Q_{c,k}(\mathbf{N})$.

Intuitively, for a system with N customers, when a new customer arrives at a station, the new customer cannot already be in the queue which suggests that only the other $N-1$ customers could have an impact on the new customer. Out of these $N-1$ customers the average number of them which are actually in the queue is given by the average queue length at the specified station for $N-1$ customers, denoted $Q_k(N-1)$. [23]

So equation (3.34) can be used in conjunction with the previous three equations to calculate exact values for the system throughput, station queue lengths and station residence times for a system with N customers. For single-class models an iterative approach is required since in order to solve the model with N customers, the model must first be solved for $N-1$ customers. For example, if we wanted to solve a model with 50 customers, we must first calculate the $A_k(50)$ values using (3.34). However this requires that we know the $Q_k(49)$ values which can only be obtained by first solving the model for 49 customers. Thus in order to obtain exact solutions for any N we must start by solving the equations for $N = 0$ and then apply them iteratively until we reach the desired N value. For the base case ($N = 0$) we can trivially see that all station queue lengths must be 0 since there are no customers in the system. Consecutive application of the equations yields solutions for customer populations $1, 2, \ldots, N-1, N$. [23]

For multi-class models the situation is more complex as the evaluation of a model for a specific $\mathbf{N}$ value requires C (the number of customer classes) inputs, one for each possible model with population $(\mathbf{N} - \mathbf{1_c})$ i.e. we must consider every possible model in which the population is $N-1$ but the customer removed is of each class type. So if we have two customer classes X and Y, we start with the base case $(0X, 0Y)$. From this we can then compute solutions for models containing one customer, $(1X, 0Y)$ and $(0X, 1Y)$. These two results can then be used to calculate solutions for models with a two customer population and so on. Looking at it another way, in order to compute a solution for (MX, NY) we must first have computed results for $((M-1)X, NY)$ and $(MX, (N-1)Y)$. These dependencies can be visualised as a 2D mesh, as shown in the diagram below. Note that in general, for a class with C customers, the dependencies can be visualised as a C-dimensional mesh.

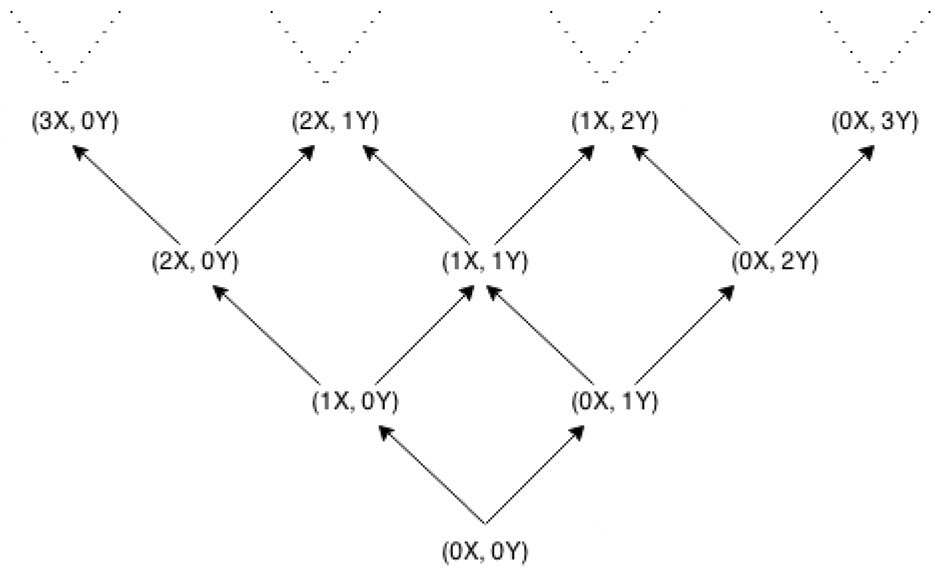


Figure 3.2: Mutli-class MVA intermediate solutions.

Below we present pseudocode for the MVA algorithm in both single and multi-class form:

Algorithm 2 Pseudocode for single-class MVA algorithm [23].

```
/** Computes residence times, throughput and mean queue lengths for single-class
  * model using MVA. */
function SingleClassMVA() {
  // Initialise station queue lengths to 0.
  for (k=1:K) Q_k = 0;

  for (n=1:N) {
    // Compute residence times.
    sumR_k = 0;
    for (k=1:K) {
      if (delayStation(k)) R_k = D_k;
      else if (queueingStation(k)) R_k = D_k(1+Q_k);
      sumR_k += R_k;
    }

    // Compute system throughput.
    X = n/(Z+sumR_k);

    // Compute mean queue lengths.
    for (k=1:K) Q_k = X*R_k;
  }
}
```

Algorithm 3 Pseudocode for multi-class MVA algorithm [23].

```
/** Computes residence times, throughput and mean queue lengths for multi-class
  * model using MVA. */
function MultiClassMVA() {
  // Initialise station queue lengths to 0.
  for (k=1:K) Q_k(0) = 0;
  for (n=1:sumOver(c, N_c)) {
    for (each population in (n1...nC) with n total customers) {
      // Compute residence times.
      for (c=1:C) {
        for (k=1:K) {
          if (delayStation(k)) R_ck = D_k;
          else if (queueingStation(k)) R_ck = D_ck(1+Q_k(n-1_c));
        }
      }

      // Compute per class throughputs.
      for (c=1:C) X_c = n_c/(Z_c + sumOver(k, R_ck))

      // Compute mean queue lengths.
      for (k=1:K) Q_k(n) = sumOver(c, X_c*R_ck)
    }
  }
}
```

Where `sumOver(k, R_ck)` gives $\sum_{k=1}^{K} R_{ck} = R_c$

Once the single-class algorithm has run we have X, the system throughput, R_k, station k residence time, and Q_k, the average queue length at station k. The multi-class algorithm gives corresponding results for X_c, R_{ck} and Q_k. We can then apply Little's Law to derive the other performance measures outlined in section 2.2.3.

The time complexity of the single-class algorithm grows linearly with $N \times K$, since to solve for population N the algorithm must loop N times, and for each loop all K stations in the system must be considered. The space complexity is K, since only the results from the directly preceding iteration need to be stored. For the multi-class algorithm the space and time requirements are both significantly greater than for the single-class algorithm due to the composite nature of the solution dependencies. The time complexity is proportional to $CK\prod_{c=1}^{C}(N_c+1)$ and the space complexity is given by $K\prod_{c=1,c\neq c_{max}}^{C}(N_c+1)$ where c_{max} is the index of the class with the greatest population. [23]

For single-class networks MVA is therefore fairly efficient however, for multi-class networks the number of arithmetic operations required grows exponentially with the number of customer classes. In practice the algorithm often starts to become infeasible with only 3-4 customer classes [8], making it unsuitable for exact evaluation of many real-world system models. For this reason various approximate versions of MVA have been developed.

3.2.6 Approximate MVA (AMVA)

The iterative nature of the exact method is completely derived from equation (3.34), which requires that we compute Q_k for the model with population $N-1$ before we can obtain A_k for the model with population N. For approximate techniques, known as Approximate MVA (AMVA), we attempt to amend this characteristic in order to significantly reduce the complexity of the algorithm. To do this we replace equation (3.34) by: [23]

$$A_k(N) \approx h(Q_k(N)) \tag{4.1}$$

where h is a heuristic function. By doing this we can directly solve a model of population N without having to solve the model for all populations less than N. However, the accuracy of the result strongly depends on the accuracy of the heuristic h used. [23]

The major advantage of this approximate approach is the increase in computational efficiency over exact MVA, particularly for multi-class models. The space complexity of AMVA for multi-class models is proportional to $C \times K$ which is a significant saving over MVA, since we now only have to store the model solution for one population, $\mathbf{N}$. The time complexity is impossible to define exactly since the number of iterations required will depend on the user defined termination criteria and also on the specific model being solved, though in practical cases the time savings over MVA are significant. The number of arithmetic operations per iteration is proportional to $C \times K$ and so the number of customers in each class are not a concern. In terms of accuracy the results are typically within a few percent of the exact solution for utilisations and throughputs, and within 10% for residence times and queue lengths. For very large multi-class models AMVA is often the only feasible approach of reaching a solution. [23]

A previous project looked at studying, implementing and evaluating some of these approximate algorithms. In particular the approximate algorithms studied were Chow, Bard-Schweitzer, Linearizer, De Souza-Muntz Linearizer and Aggregate Queue Length (AQL). Since approximate techniques are not the focus of this project we do not discuss these algorithms in detail, though further information can be found in [13]. It should be noted that a downside to the approximate method is that there are no restrictions on the accuracy of the result [23]. So while in most cases the result may be contained within a small error margin, there may be cases in which the result is much further away from the exact solution. This is one of the motivations for developing an exact technique which is more computationally efficient than MVA.

3.2.7 Related Work

In addition to the algorithms we have discussed there are several other approaches for obtaining performance measures from closed queueing networks. Below we give a brief overview of these techniques for completeness:

- **RECAL** - The Recursion by Chain Algorithm (RECAL) [14] can be used to calculate mean performance measures in multi-class networks. Like the Convolution algorithm it is based on the calculation of normalisation constants in a recursive fashion. It relies on an expression which relates the normalization constant of a network with c classes to the normalization constants for a set of networks with $c - 1$ classes. This strategy works by decomposing each class into subclasses which only contain one customer. The time and space requirements of RECAL are polynomial in the number of customer classes meaning that RECAL is more efficient than both sequential MVA and Convolution when the network contains many classes. However, RECAL still does not scale particularly well for models with many stations, as was noted in [3].

- **MoM** - The Method of Moments (MoM) algorithm [8] manages to avoid the combinatorial growth of recursions which the Convolution and RECAL algorithms suffer from, by recursively evaluating a set of linear systems that relate bases of normalising constants [8]. The basis is essentially a set of higher-order moments which is defined so as to allow bases for larger populations to be computed.

- **CoMoM** - Class-Oriented Method of Moments (CoMoM) [7] is a variation on MoM which is suited to the exact evaluation of networks containing large numbers of customer classes. The difference is in how the bases of normalising constants are defined. For CoMoM the basis is formed from a wider set of different populations but never adds more than one copy of a queue [7]. This method allows CoMoM to scale more efficiently as the number of customer classes increases.

RECAL, MoM and CoMoM were all implemented within JMVA as part of earlier projects so it will be interesting to compare their performance with the tree algorithms which form the backbone of this project.

Chapter 4

Tree Algorithms for Sparse Networks

4.1 Tree Convolution (TC)

In 1983, Lam and Lien proposed a modified version of the Convolution algorithm which employs a tree data structure and allows the solution of certain models which would have been intractable using the sequential Convolution algorithm [21]. Specifically the algorithm is highly efficient in models which contain many stations and customer classes, and which exhibit the *sparseness property*, which says that on average a certain class of customer is serviced by only a small fraction of the stations in the model. If the customer classes are also clustered into specific subregions of the network (*locality property*) then the complexity of the algorithm can be further reduced. [21]

One of the primary goals of this project is to implement a version of the Tree Convolution algorithm which will allow the exact solution of large sparse multi-class models.

4.1.1 Theoretical Concepts and Notation

Before we can discuss the algorithm it is necessary to understand the following concepts and notation: [21]

1. **Class Coverage** - If we consider the set of stations which are visited by a certain class of customers c and call this set STATIONS(c), then the class of customers c is said to be *fully covered* with respect to a subnetwork $\mathcal{A}$ of stations if we have STATIONS(c) $\subseteq \mathcal{A}$. Conversely if STATIONS(c) $\cap \mathcal{A} = \varnothing$, class c is *noncovered* with respect to $\mathcal{A}$. If neither of these is true (i.e. only some of the stations in STATIONS(c) are in $\mathcal{A}$) then class c is *partially covered* with respect to $\mathcal{A}$.

2. **Partially covered arrays for reducing space requirements of Convolution** - If we again consider a subnetwork $\mathcal{A}$ which consists of stations $(k_1, k_2, \ldots, k_x) \subseteq (1, 2, \ldots, K)$ then by equation (3.25) from the sequential Convolution algorithm, we define $g_{\mathcal{A}} = p_{k_1} * p_{k_2} * \ldots * p_{k_x}$, which is an intermediate step towards calculating $g_K(\mathbf{N}) \equiv G(\mathbf{N})$. The principle realisation for reducing space requirements is that some classes of customer will be fully covered or noncovered with respect to $\mathcal{A}$, which means that only some of the values within array $g_{\mathcal{A}}$ are required in order to compute $G(\mathbf{N})$. In particular, if the set of all customer classes is split into three sets such that σ_{pc} contains all classes which are partially covered by $\mathcal{A}$, σ_{nc} contains classes which are noncovered by $\mathcal{A}$, and σ_{fc} contains all fully covered classes with respect to $\mathcal{A}$, then it is possible to store $g_{\mathcal{A}}$ as a $|\sigma_{pc}|$-dimensional array and still be able to successfully calculate $G(\mathbf{N})$. Such an array is

called a *partially covered array* or *g-array* in [21]. By using partially covered arrays the space required for the Convolution algorithm can be reduced to $\prod_{c\in\sigma_{pc}}(N_c+1)$ memory locations, since we no longer need to use C-dimensional arrays. In queueing network models which exhibit sparseness and locality properties the space savings from using this approach can be considerable. (Note that since the size of the partially covered arrays varies from model to model, a dynamic allocation scheme is desirable).

3. **Reducing time requirements of Convolution** - Once again we consider a subnetwork $\mathcal{A}$, however we now partition the set into two subsets, $\mathcal{A}_1$ and $\mathcal{A}_2$, giving:

$$g_{\mathcal{A}} = g_{\mathcal{A}_1} * g_{\mathcal{A}_2} \tag{4.1}$$

   Now in addition to the class coverage properties we have already discussed, we say that a customer class c is *overlapped* if it is partially covered by both $\mathcal{A}_1$ and $\mathcal{A}_2$. Thus we can classify each customer class c based on two properties; (1) whether or not c is partially covered with respect to $\mathcal{A}$ and (2) whether or not c is overlapped with respect to $\mathcal{A}_1$ and $\mathcal{A}_2$. Based on this categorisation we can split the set of customer classes into four sets, $\sigma_{no,np}$ (not overlapped, not partially covered), $\sigma_{no,p}$ (not overlapped, partially covered), $\sigma_{o,np}$ (overlapped, not partially covered), and $\sigma_{o,p}$ (overlapped, partially covered). If partially covered arrays are used for the computation of $g_{\mathcal{A}} = g_{\mathcal{A}_1} * g_{\mathcal{A}_2}$ then the time requirement can be reduced to the order of:

$$\prod_{c\in\sigma_{o,np}\cup\sigma_{no,p}} (N_c+1) \prod_{c\in\sigma_{o,p}} (N_c+2)(N_c+1)/2 \tag{4.2}$$

   Therefore by using partially covered arrays we can reduce both the time and space complexity of the Convolution operation considerably, particularly if there are only a few partially covered classes in $\mathcal{A}$, $\mathcal{A}_1$ and $\mathcal{A}_2$, which is likely to be the case in sparse networks.

4. **Exploiting routing information** - The techniques described so far can be applied to the sequential algorithm in order to reduce computational costs. However, significantly more time and space savings can be made by taking the network's routing information into account. Looking back at equation (3.25) we can see that the Convolution algorithm essentially starts with one station (smallest possible subnet) and then proceeds to merge this station with other stations, in a sequential fashion, until all stations have been merged. The key point is that since the Convolution operation is commutative we can modify the order in which the merging occurs so as to reduce the number of partially covered classes at intermediate subnetworks. By taking into account the network routing information, which specifies which customer classes are serviced by which stations, we can therefore obtain an optimal merging sequence to maximise time and space savings.

4.1.2 Algorithm Overview

The ideas discussed above form the basis for the Tree Convolution algorithm outlined in [21] which makes use of a binary tree data structure. In practice the algorithm consists of three main stages:

1. *Preprocessor stage* - this is the initial setup stage which can be further separated into two phases:

   - *Tree planting phase* - which involves converting the network into a tree which can be solved efficiently.

- *Complexity evaluation phase* - where the complexity of the Tree Convolution algorithm, given the planted tree, is calculated.

2. *Normalisation constant calculation* - in which the convolution operation is applied to the nodes in the tree as it is traversed, in order to calculate the normalisation constant at the root.

3. *Performance measure calculation* - which involves further partial traversals (often of subtrees) in order to calculate additional normalisation constants required to compute performance measures.

During the tree planting stage, stations are placed at the bottom-most level of the tree as leaves, and all non-leaf nodes represent subnetworks containing all the stations from both child nodes. So the tree's root node represents the whole network and for a network with K stations there will be $K-1$ non-leaf nodes representing subnetworks. This setup is shown below for $K = 9$:

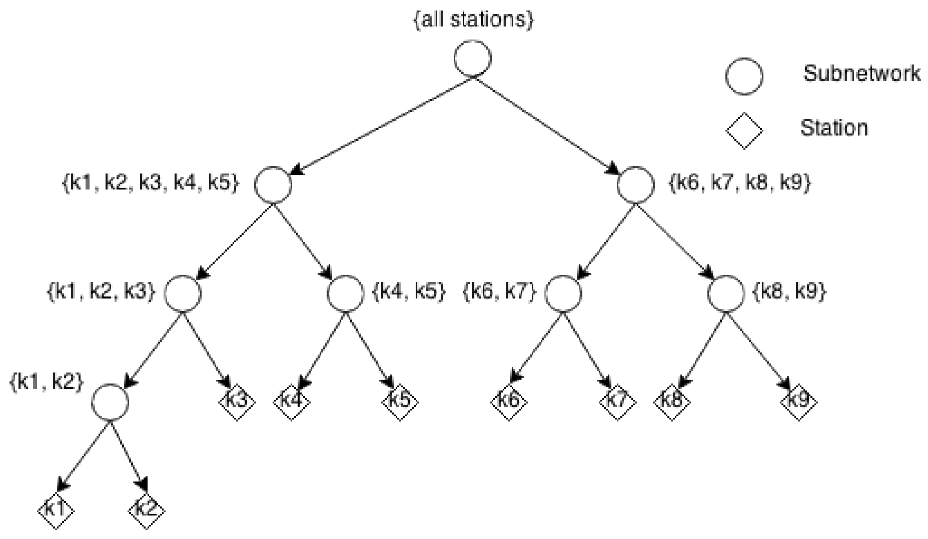


Figure 4.1: Binary tree setup for Tree Convolution algorithm.

In practice this binary tree will be set up in such as way as to optimise the algorithm's performance by exploiting routing information (see section 4.3).

4.1.3 Normalization Constant Calculation

Once the tree has been planted the normalisation constant is calculated by traversing the tree in a postorder fashion, with the root node being visited last. Any non-leaf node can only be visited if both of its children have already been visited. The calculation performed at each node is as follows: [21]

- **Station/leaf node** - At a leaf node, representing a single station, the g-array can be obtained from a modified version of equation (3.23) given by:

$$g_{\{k\}}(\mathbf{i}_{pc}) = \left(\prod_{j=1}^{n_k} \frac{1}{u_k(j)}\right) n_k! \prod_{c\in\sigma_{pc}} \frac{D_{kc}^{i_c}}{i_c!} \prod_{c\in\sigma_{fc}} \frac{D_{kc}^{N_c}}{N_c!} \tag{4.3}$$
$$\text{for } \mathbf{i}_{pc}, \text{ where } 0 \le i_c \le N_c, c \in \sigma_{pc}$$

where {k} is a subnetwork consisting of only the station k represented by the leaf node, σ_{pc} is the set of partially covered classes at the leaf node and σ_{fc} is the set of fully covered classes. Additionally, $n_k = \sum_{c\in\sigma_{pc}} i_c + \sum_{c\in\sigma_{fc}} N_c$. In terms of time complexity, equation (4.3) requires the following number of multiplications:

$$4 \times \prod_{c\in\sigma_{pc}\cup\sigma_{fc}} (N_c + 1) \tag{4.4}$$

- **Subnetwork/non-leaf node** - At a non-leaf node, representing a subnetwork, the g-array is computed from the node's two children using equation (4.1) and the approach outlined in item (3) from section 4.1.1 above. If one of the children is a leaf node representing a load independent station then equation (3.26) can be used.
- **Root node** - The same technique is applied as for all other non-leaf nodes, however the output is the normalisation constant $G(\mathbf{N})$ for the whole network.

It is worth noting that the sequential Convolution algorithm we defined previously is a special case of the Tree Convolution algorithm. For both the sequential and tree versions of the algorithm the total number of convolution operations required is $K - 1$. However, Tree Convolution allows the order of the operations to be modified so as to minimize the size of the partially covered arrays stored at each node, which can lead to significant savings in time and space. [21]

4.1.4 Feedback Filtering

An alternative approach known as *feedback filtering* can be used instead of the standard convolution equation in some cases. The equation (3.26) used for the sequential version of the algorithm can be modified so that it can be applied to two leaf nodes, so long as at least one of the leaf nodes is a fixed-rate station.

Given two leaf nodes representing stations x and y, where x is a fixed-rate station. We define: [21]

$$\mathbf{i}_{xy} = \{i_c, c \in \sigma_x \cup \sigma_y\} \tag{4.5}$$

where σ_x is the set of classes partially or fully covered by station x and σ_y is the set of classes partially covered by y. Using this notation, equation (3.26) can be rewritten as: [21]

$$g_{\{x,y\}}(\mathbf{i}_{xy}) = g_{\{y\}}(i_c, c \in \sigma_y)\delta(\mathbf{i}_{xy}) + \sum_{c\in\sigma_x} D_{xc} g_{\{x,y\}}(\mathbf{i}_{xy} - \mathbf{1}_c) \tag{4.6}$$

for $\mathbf{i}_{xy}$ where $\mathbf{i}_c = 0, 1, \ldots, N_c$ for $c \in \sigma_x \cup \sigma_y$, $\mathbf{1}_c$ is a vector with the cth element set to 1 and the remaining elements set to 0, and the delta function is defined as: [21]

$$\delta(\mathbf{i}_{xy}) = \begin{cases} 0 & \text{if } i_c > 0 \text{ for any } c \in (\sigma_x - \sigma_y) \\ 1 & \text{otherwise} \end{cases} \tag{4.7}$$

Once equation (4.6) has been applied at a node, the following equation is used to obtain the partially covered array for the parent node which covers the subnetwork $\{x, y\}$: [21]

$$g_{\{x,y\}}(\mathbf{i}_{pc}) = g_{\{x,y\}}(i_c \text{ for } c \in \sigma_{pc}; N_c \text{ for } c \in \sigma_{fc}) \tag{4.8}$$

Using this feedback filtering approach when appropriate can reduce the time required to compute the partially covered g-array at a node when compared to the standard convolution equation. However, this is only true under the condition that the population of the overlapped classes between the two child nodes are large.

4.1.5 Performance Measure Computation

Once the preprocessor and core algorithm stages have run we are left with a value for $G(\mathbf{N})$ at the root node. From this point we aim to calculate meaningful performance measures for the system, however, as we shall see, this often requires the computation of additional normalization constants.

4.1.5.1 Throughputs

For both load independent and load dependent stations, the throughput of class c customers at station k is given by: [21]

$$X_{kc}(\mathbf{N}) = \lambda_{kc}\frac{G(\mathbf{N}-\mathbf{1}_c)}{G(\mathbf{N})} \qquad \text{for } 1 \le c \le C, 1 \le k \le K, \mathbf{N} \ge \mathbf{1_c} \tag{4.9}$$

where λ_{kc} is the arrival rate of class c customers at station k and $G(\mathbf{N}-\mathbf{1}_c)$ is the normalisation constant of the network with one class c customer removed.

Since we already have $G(\mathbf{N})$ from the previous stage of the algorithm, $G(\mathbf{N}-\mathbf{1_c})$ is the only constant left to calculate. In order to calculate $G(\mathbf{N}-\mathbf{1_c})$ for all possible classes c, two methods are suggested in [21]:

1. Considering a customer class c that is partially covered by a station k, we find the node higher up the tree at which class c becomes fully covered. Convolution is then performed at this node to compute the g-array for population $N_c - 1$. Additional convolutions are then performed sequentially along the path from the node to the root in order to calculate $G(\mathbf{N}-\mathbf{1_c})$. If class c becomes fully covered immediately at a leaf node, then the g-array of station k can be obtained from the previously stored g-array at this leaf node using the following relation:

$$g_{\{k\}}(\mathbf{i}_{pc}) \leftarrow \frac{u(n_k)N_c}{n_k D_{kc}} g_{\{k\}}(\mathbf{i}_{pc}) \qquad \text{for } \mathbf{i}_{pc},\ \text{where } 0 \le i_c \le N_c, c \in \sigma_{pc} \tag{4.10}$$

2. The second method suggested is to compute the normalisation constants $G(\mathbf{N}-\mathbf{1}_c)$ for c in the range [1, C] during the same tree traversal that we compute $G(\mathbf{N})$. In practice this means that at each node in the tree we compute a g-array for population $\mathbf{N}$ and several g-arrays for population $\mathbf{N}-\mathbf{1_c}$, one for each class c in the set of fully covered classes at the node. At the root node we will then be able to obtain $G(\mathbf{N})$ and $G(\mathbf{N}-\mathbf{1_c})$ for c in the range [1, C].

4.1.5.2 Mean Queue Lengths

The average number of class c customers at station k can be computed as: [21]

$$Q_{kc}(\mathbf{N}) = \begin{cases} 0 & \text{if } c \text{ is noncovered by k} \\ N_k & \text{if } c \text{ is fully covered by k} \\ D_{kc}\dfrac{G_{k+}(\mathbf{N}-\mathbf{1_c})}{G(\mathbf{N})} & \text{if } c \text{ is partially covered by load independent station } k \end{cases} \tag{4.11}$$

for any station k in the range [1, K], any class c in the range [1, C] and $\mathbf{N} \ge \mathbf{1_c}$. Where $G_{k+}(\mathbf{N}-\mathbf{1_c})$ is the normalisation constant of the network with one class c customer removed, computed from a tree in which station k occurs twice at two leaf nodes (i.e. a clone of station k is made).

For load dependent stations the average station population for a particular class of customer can be computed from the marginal distribution of queue lengths at a station k, which can be obtained from: [21]

$$p_k(\mathbf{n_k}) = \frac{p_k(\mathbf{n_k})G_{k-}(\mathbf{N} - \mathbf{n_k})}{G(\mathbf{N})} \tag{4.12}$$

for any station k in the range $[1, K]$ and $\mathbf{0} \leq \mathbf{n_m} \leq \mathbf{N}$. Where $G_{k-}(\mathbf{N} - \mathbf{n_k})$ is the normalisation constant obtained from the original tree with station k removed.

Below we outline strategies which can be used to compute $G_{k+}(\mathbf{N} - \mathbf{1_c})$ and $G_{k-}(\mathbf{N} - \mathbf{n_k})$, assuming for now that the g-arrays obtained during the computation of $G(\mathbf{N})$ are stored in memory: [21]

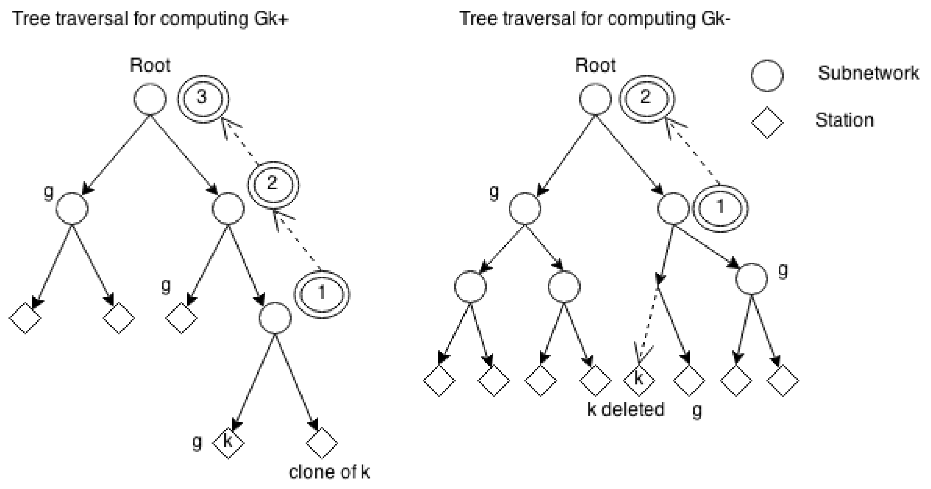

Tree traversal for computing Gk+
Root
3
2
1
g
g
g
k
clone of k
Tree traversal for computing Gk-
Root
2
1
g
g
k
k deleted
g
Subnetwork
Station

Figure 4.2: Summary of how the normalization constants $G_{k+}(\mathbf{N} - \mathbf{1_c})$ and array $G_{k-}(\mathbf{N} - \mathbf{n_k})$ can be computed. The dashed arrows indicate the route taken and the label ‘g’ indicates which nodes we need g-arrays for. Note that for G_{k+} we require $(log_2 4) + 1 \equiv 3$ convolution operations and for G_{k-} we require $(log_2 8) - 1 \equiv 2$ convolution operations.

1. $G_{k+}(\mathbf{N} - \mathbf{1_c})$ - is the normalization constant for a modified version of the initial tree in which station k is cloned and one customer of class c is removed from the population. It is required in order to compute the mean queue lengths for load independent stations. After station k is cloned, a customer class which was originally fully covered by k will now only be partially covered by k, but fully covered by k and k's clone together. $G_{k+}(\mathbf{N} - \mathbf{1_c})$ can be calculated by reevaluating the convolutions along the path in the tree from station k to the root node, which will require $(log_2 K) + 1$ convolution operations in a balanced tree. In fact $G_{k+}(\mathbf{N} - \mathbf{1_c})$ must be calculated for every class c which is partially covered by k. One way of achieving this, if we have some extra space, is to compute $G_{k+}(\mathbf{N} - \mathbf{1_c})$ for all required classes c at the same time by calculating and storing multiple g-arrays at each node.

2. $G_{k-}(\mathbf{N} - \mathbf{n_k})$ - is the normalization constant which can be computed from the initial tree with station k deleted and is required to compute the marginal distribution of queue lengths in load dependent stations. The side effect of removing k from the tree is that all customer classes which were covered by station k now remain partially covered at the root node and so G_{k-} can be considered to be an array indexed by $i_c = 0, 1, \ldots, N_c$ for all customer classes c that were partially covered by station k. G_{k-} is found by recalculating

the convolutions along the path between station k and the root node, which for a balanced tree requires $(log_2 K) - 1$ convolution operations.

4.1.6 Space-Time Trade-offs

When describing the methods we can use to calculate the normalisation constants above, we assumed that the g-arrays from the computation of $G(\mathbf{N})$ were stored in memory. However, the calculation of each extra normalisation constant for each customer class and station can be computed using about the same amount of space as the initial calculation of $G(\mathbf{N})$. Therefore if memory is limited, we could reuse the same space for each computation, allowing the time requirement to to increase to $(K + 1)C$ times the time requirement of computing $G(\mathbf{N})$. On the other hand, if extra space is available then we can make a space-time trade-off by using the additional space to partially or fully store the g-arrays from the computation of $G(\mathbf{N})$ and also compute multiple normalisation constants at the same time as suggested above. This would enable the constants to be calculated without having to fully traverse the tree each time, leading to savings in time. In particular [21] notes that in practical cases they found that modest investments in space often led to significant time savings. If only a subset of the g-arrays can be stored in memory then their storage should be prioritized at nodes closer to the top of the tree, since these are the most frequently accessed. [21]

4.1.7 Complexity Analysis

During the preprocessor stage the space and time requirements for the planted tree can be approximated as follows: [21]

- **Time complexity** - The total time required to compute $G(\mathbf{N})$ can be found by summing the time requirements to compute the g-arrays for the K leaf station nodes (given by equation (4.4)) and the $K-1$ non-leaf subnetwork nodes (for which the time requirement is given by (4.2)). The total time required to compute specific performance measures after calculating $G(\mathbf{N})$ may be significantly higher than the time required to compute $G(\mathbf{N})$, depending on the space-time trade-offs used. So this must also be taken in account.

- **Space complexity** - The total amount of space required to compute $G(\mathbf{N})$ is the maximum amount of space required to store the intermediate g-arrays at the same time. The number of g-arrays which need to be stored in parallel depends on the tree traversal order. However, the number of arrays itself is not enough to provide an accurate space estimate since the partially covered arrays at each node are of different sizes. So both of these factors should be taken into account to get an accurate estimate for the space requirements. The total amount of space required to compute specific performance measures after calculating $G(\mathbf{N})$ will be approximately the same as the space required for computing $G(\mathbf{N})$, so the space can be reused if memory is limited.

4.1.8 Algorithm Pseudocode

Below is the high-level pseudocode for the complete Tree Convolution algorithm we have described. For all customer classes c in range [1, C] and stations k in range [1, K], the 'inputs' include STATIONS(c) (the set of stations at which a class c is serviced), N_c (the population of class c), D_{kc} (the service demands matrix), and u_k (the service rate of station k).

Algorithm 4 High-level pseudocode for Tree Convolution algorithm [21].

```
/** Runs the Tree Convolution algorithm. */
function runTreeConvolution(inputs) {

  // 1: Preprocessor stage
  suitableTreeFound = false;
  int round = 0;
  while(!suitableTreeFound) {
    binaryTree = plantTree();
    suitableTreeFound = treeHasAcceptableComplexity(binaryTree);
    if (!suitableTreeFound && round > MAX_ROUND) terminate algorithm early;
    round++;
  }

  // 2: Normalisation constant calculation stage
  Define/decide time-space trade-offs;
  Postorder tree traversal on binaryTree to compute G(N);

  // 3: Performance measure calculation stage
  Compute additional normalisation constants;
  Compute performance measures using normalisation constants;

  return performance measures;
}
```

4.2 Tree MVA (TMVA)

Inspired by Lam and Lien's approach to developing the Tree Convolution algorithm a technique called Tree MVA (TMVA) was developed independently by Tucci in 1985 [34] and Hoyme et al. in 1986 [17]. The TMVA algorithm makes use of Lam and Lien's ideas summarised previously in section 4.1.1, but it is also influenced by the hierarchical MVA approach outlined by Sauer in [30]. TMVA, like Tree Convolution, exploits a network's routing information to substantially decrease the computational complexity (both in space and time) of computing performance measures in sparse multi-class networks, when compared to the sequential MVA algorithm [34].

4.2.1 Algorithm Overview

The algorithm has a similar structure to the Tree Convolution algorithm in that it can be separated into two stages; the preprocessor phase and the merging phase. The preprocessor phase is essentially the same as before except that after planting a tree the estimated time and space complexities will be calculated differently due to the fact that the main merging phase is different. However, the goal of the tree planting phase is still to find a binary tree which will maximise the efficiency of running the core TMVA algorithm.

The merging phase for TMVA is conceptually simpler than for Tree Convolution since it only requires one traversal of the tree in order to compute all performance measures (rather than having to compute the normalization constant and performance measures independently). It should also be noted that the approaches presented by Hoyme in [17] and Tucci in [34] are slightly different in that Hoyme starts by initializing the leaf nodes and then applying the TMVA equations for all subnetwork nodes, whereas Tucci first applies sequential MVA to the bottom-most node pairs before applying the TMVA equations to all higher levels of the tree. While implementing the algorithm, both variations were attempted and Tucci's approach was found to be the fastest for most practical cases. In view of this we present Tucci's version of the algorithm here, though some of the notation is borrowed from Hoyme due to its terseness.

Whilst the sequential MVA approach cannot exploit routing information due to the fact that the entire network is analysed in one pass within the inner level of the algorithm, TMVA introduces a commutative merge operation (in a similar fashion to the commutative convolution operator), which allows this exploitation to occur. At the heart of this operation is a function known as a *station service function* which is simply defined as the reciprocal of the throughput at a subnetwork node: [17]

$$S_{A,c}(\mathbf{n}) = \frac{1}{X_{A,c}(\mathbf{n})} \tag{4.13}$$

where A represents a subnetwork, c is the class and $\mathbf{n}$ is the population vector. By using this function in conjunction with the Forced Flow Law for two subnetworks, A and B, the flow-equivalent throughput can be computed without having to evaluate intermediate throughputs for individual stations. Note that the flow-equivalent throughput at a subnetwork is the throughput when the service demands for all stations outside of the subnetwork are set to zero [21]. Thus TMVA is able to use this approach to efficiently compute performance measures, a subnetwork at a time, until performance measures are acquired for the whole system. The merging of subnetworks in this fashion can be explained through the analogy of *Norton's Theorem* for DC circuits, which states that two terminals can be substituted for an equivalent current source. Chandy proves that this idea also holds for queueing network models in [10]. [34]

A detailed description of the merging phase is given below: [34, 17]

1. **Initial step** - The leaf nodes of the tree, representing individual stations, are considered in pairs. For each pair a merge step is done by applying the sequential MVA algorithm directly, using the equations outlined in section 3.2.5. The results of merging a single pair are stored at the pair's parent node, which represents an aggregate/composite station. So if initially the tree has K ($= 2^p$) stations as leaves, after the initial step there will be 2^{p-1} composite stations at the level above.

2. **Merging of aggregate stations** - Once the initial step is complete, the throughput and mean queue length results stored in the aggregate station nodes are used to perform a further merge step. Once again the aggregate stations are taken in pairs. For the sake of simplicity we consider a single pair of aggregate station nodes, A and B. Their parent node, which will eventually store the aggregate results after merging A and B, is referred to as AB.

   The residence time for class c at subnetwork A is given by:

   $$R_{A,c}(\mathbf{n}) = \sum_{\mathbf{k}=\mathbf{1_c}}^{\mathbf{n}} k_c S_{A,c}(\mathbf{k}) p_A(\mathbf{k}-\mathbf{1_c}, \mathbf{n}-\mathbf{1_c}) \tag{4.14}$$

   where k_c is the the number of class c jobs in vector $\mathbf{k}$, $S_{A,c}$ is the service function defined by equation (4.13) and the marginal probability function $p_A(\mathbf{k},\mathbf{n})$ gives the probability that $\mathbf{k}$ jobs reside in subnetwork A when there are $\mathbf{n}$ jobs in the whole system.

   The function $p_A(\mathbf{k},\mathbf{n})$ is defined as:

   $$p_A(\mathbf{k},\mathbf{n}) = x_{AB,c}(\mathbf{n}) S_{A,c}(\mathbf{k}) p_A(\mathbf{k}-\mathbf{1_c}, \mathbf{n}-\mathbf{1_c}) \tag{4.15}$$

   where $x_{AB,c}(\mathbf{n})$ is the throughput at the parent node AB for population $\mathbf{n}$ and c is chosen such that $k_c > 0$. In order to avoid numerical instability problems when $\mathbf{k}$ is a zero-vector is is also defined that:

   $$p_A(\mathbf{0},\mathbf{n}) = x_{AB,c}(\mathbf{n}) S_{B,c}(\mathbf{n}) p_A(\mathbf{0}, \mathbf{n}-\mathbf{1_c}) \tag{4.16}$$

and additionally, $p_A(\mathbf{0}, \mathbf{0}) = 1$. The above equations can also be applied to subnetwork B by simply switching A for B. Once the residence times at both A and B have been computed the per class throughput at the parent node AB can be obtained via Little's Law:

$$X_{AB,c}(\mathbf{n}) = \frac{n_c}{R_{A,c}(\mathbf{n}) + R_{B,c}(\mathbf{n})} \tag{4.17}$$

Equation (4.13) can then be applied on AB to get the service function for the parent node.

The calculation of per station mean queue lengths for AB is simpler and involves adjusting the mean queue lengths computed at A and B to take into account the presence of the other subnetwork using the marginal probability function we have already defined. The equation is:

$$Q_{m,AB,c}(\mathbf{n}) = \sum_{\mathbf{k}=\mathbf{0}}^{\mathbf{n}} Q_{m,A,c}(\mathbf{k}) \times p_A(\mathbf{k}, \mathbf{n}) \tag{4.18}$$

where $Q_{m,AB,c}(\mathbf{n})$ is the mean queue length at subnetwork AB for station $m \in A$, class c, and population vector $\mathbf{n}$. Again this can also be applied similarly for subnetwork B.

This process is repeated so that we obtain networks with $2^p, 2^{p-1}, \ldots, 2^0$ composite stations. The results at the root node can then be used to calculate performance metrics for the original network.

One important point to notice is that although the equations above appear to show summations over the entire state space of a network, in the actual implementation we only iterate over the partially covered classes which are common to both child subnetworks A and B. The fully covered classes are not iterated over since we know that all jobs in a fully covered class are restricted to the subnetwork represented by the node under consideration and hence can be set to the maximum population for that class. Thus for sparse networks, in which the number of partially covered classes at any given node will be small, the TMVA merge algorithm will be highly efficient.

4.2.2 Potential Optimisations

There are a few potential optimisations for the algorithm we have presented so far which are alluded to by both Tucci and Hoyme. First of all we can avoid having to compute the function $p(\mathbf{k}, \mathbf{n})$ for both subnetworks A and B by taking into account the relation: [17]

$$p_A(\mathbf{k}, \mathbf{n}) = p_B(\mathbf{n} - \mathbf{k}, \mathbf{n}) \tag{4.19}$$

Hence we only need to compute $p_B(\mathbf{k}, \mathbf{n})$ as this relation can be substituted into the main algorithm equations presented above so that all equations are in terms of $p_B(\mathbf{k}, \mathbf{n})$. Furthermore we can store the results of $p_B(\mathbf{k}, \mathbf{n})$ temporarily so that they can be reused, for example in equations (4.14), (4.15) and (4.18).

Another optimisation is possible when an aggregate station is to be merged with a single load-independent station. In this case the residence time for the leaf node station L can be simplified to: [34, 17]

$$R_{L,c}(\mathbf{n}) = D_{m,c}(1 + Q_{m,L}(\mathbf{n} - \mathbf{1_c})) \tag{4.20}$$

where $Q_{m,L}(\mathbf{n}) = \sum_{c=1}^{C} Q_{m,L,c}(\mathbf{n})$ and $D_{m,c}$ is the service demand for station m and class c.

4.2.3 Complexity Analysis

Considering a network with K stations and C customer classes, we have to apply the merging process outlined above $K-1$ times. For each of these applications, the pair of stations being combined must be solved X times where X is the number of combinations of populations in the set of common partially covered classes from the two child nodes. If we let the partially and fully covered classes at a node be represented by σ_{pc} and σ_{fc} as before, then for each pair of stations and each population combination, the time complexity is of the order $2(|\sigma_{fc}|)\prod_{c\in\sigma_{fc}}(N_c+1)$. If there are no fully covered classes at the node then we assume that this equation evaluates to 1. If the pair of stations is visited by partially covered classes contained in σ_{pc} then the pair has to be solved $\prod_{c\in\sigma_{pc}}(N_c+1)$ times in order to calculate all combinations. Therefore, if the TMVA equations are applied at $p-1$ levels in the tree and there are 2^{l-1} station pairs at each level l, then the total time requirement is given by: [34]

$$\sum_{i=1}^{p}\sum_{j=1}^{2^{p-i}}\left(\prod_{c\in\sigma_{pc,j}^{p-i}}(N_c+1)\right)\left(2(|\sigma_{fc,j}^{p-i}|)\prod_{c\in\sigma_{fc,j}^{p-i}}(N_c+1)\right) \tag{4.21}$$

where $\sigma_{pc,j}^{p}$ and $\sigma_{fc,j}^{p}$ represent the partially and fully covered classes respectively at station j, from level p in the tree.

A similar expression also gives the space complexity. In fact one advantage of using a tree data structure is that it makes various space-time trade-offs possible for different models, as we discussed for Tree Convolution, and allows the integration of storage management for very large models [21]. It should be noted that although TMVA has the potential to significantly increase the number and size of models that can be solved exactly, it cannot solve all models and so the study of approximate approaches (AMVA), as outlined in section 3.2.6, is still worth pursuing. This leads us to another benefit of TMVA which is that it allows the results provided by approximate techniques to be validated over a larger set of models than was possible with sequential algorithms, without having to resort to simulation techniques [21].

4.2.4 Algorithm Pseudocode

Below is the high-level pseudocode for the Tree MVA merge operation we described between two subnetworks A and B, assuming that any values for $p(\mathbf{k},\mathbf{n})$ are reused where appropriate:

Algorithm 5 High-level pseudocode for TMVA merge operation [17].

```
function TMVAMerge(A, B) {
  for (each population vector between 1 and N) {
    compute residence times at A and B using equation (4.14);
    compute throughput of merged AB using equation (4.17);
    use throughput to compute service function at AB using equation (4.13);
    for (each station in AB) {
      Adjust mean queue length in AB using equation (4.18)
    }
  }
}
```

4.3 Tree Planting

A primary consideration when looking at both the TMVA and TC algorithms outlined so far is that the algorithms' time and space complexity depends very much on the model we are trying to solve and on how this model is laid out within a binary tree. For this reason we saw that the initial phase of the algorithm, referred to as the *preprocessor* stage, is concerned with finding a tree configuration which will minimize the computational cost of running the main phase of the algorithms. The 'tree configuration' refers to the overall tree structure, the positioning of stations at leaf nodes in the tree and the order in which the tree should be traversed for optimal results. The preprocessor uses information about the queueing network model we want to solve, such as service demands, the population vector $\mathbf{N}$, and routing data, in order to optimise this configuration. Unfortunately no efficient algorithm has been found to solve this optimisation problem exactly, however many effective heuristics exist. [21]

Regardless of the heuristics used, the tree planting procedures suggested by Lam and Lien always follow the same basic pattern. First of all, we start with our K stations which are set as the leaf nodes in the tree we are trying to build. For each station we calculate the set of partially covered customer classes. If we find two sets of partially covered classes such that one set is a subset of the other (this is referred to as a *superset relationship* in [21]) then the two stations corresponding to these sets are merged i.e. a new node is formed at the level above. If no more superset relationships can be found then the nodes are merged depending on a costing procedure, with the precise implementation depending on the heuristic we choose to use. This process is then repeated at each level of the tree until we reach a single root node. [21]

When there are no longer any superset relationships we must use a heuristic to select two candidate nodes for merging. The first candidate can be selected by finding the subnetwork with the greatest weight. Consider a subnetwork $\mathcal{A}$ and a set σ_{pc} which contains the customer classes partially covered by $\mathcal{A}$, then the weight of $\mathcal{A}$ can be defined: [21]

$$weight(\mathcal{A}) = \sum_{c \in \sigma_{pc}} |STATIONS(c) - \mathcal{A}| \tag{4.22}$$

Once the first candidate has been selected by weight, we then select a second candidate to minimise a costing function. If we let A be the selected first candidate and B be a potential second candidate then we can define a costing function as:

Algorithm 6 Example costing heuristic for tree planting [21].

```
/** Computes cost of merging two nodes A and B.
  * @param A The first candidate node.
  * @param B The potential second candidate node.
  * @return The cost of merging the nodes.
  */
function cost(A, B) {
  cost = 0;
  for (each c in Bs set of partially covered classes) {
    if (c is not covered by A) cost=cost+1;
    else if (c is partially covered by A && not fully covered by AUB) cost=cost-1;
    else if (c is partially covered by A && fully covered by AUB) cost=cost-2;
  }

  return cost;
}
```

Below we give the pseudocode for a tree planting algorithm suggested by Lam and Lien which can be used to plant a balanced binary tree. (This algorithm assumes that K, the number of stations, is a power of 2 for simplicity however the actual implementation works for any number of stations):

Algorithm 7 High-level pseudocode for tree planting [21].

```
function plantTree() {
  initialise tree with K stations as leaves;
  for (each level of the tree from leaves to root) {
    perform superset merges;
    sort unmerged subnetworks by weight in descending order;
    mark all unmerged subsets;
    while (there are marked subsets) {
      first merge candidate A
        = heaviest marked subset as defined by equation (4.22);
      second merge candidate B
        = marked subset B such that cost(A, B) is minimised;
      merge the two candidates into single unmarked subset at level above;
    }
  }
}
```

If there is more than one marked second candidate node which minimises `cost(A, B)` then the tie is broken first by weight and then by random choice. The space and time requirements of the preprocessor stage itself (i.e. the tree planting and complexity evaluation phases) are negligible compared to the main stages of both TMVA and TC algorithms.

Chapter 5

Design & Implementation

In this chapter we describe the main contributions of this project and discuss in detail the implementation and architectural decisions that were made. Since this project was integrated into an existing software solution, we also cover some of the software engineering aspects of the design. In particular, we focus on how the solution was designed to be extensible, maintainable and testable.

5.1 Existing JMT Architecture

In order to aid in understanding, we first give an overview of the existing software architectures in place. Since JMT is a large and ongoing project, we only focus on parts of the codebase which were modified during this project. JMT has been designed so that the core analytical and simulation engines are separated from the GUI by an XML abstraction layer. This makes the system very flexible and simple to test, since the underlying solvers can be called directly from the commandline by passing in an XML file representing the model to be solved. This modular design also allows for reusability, since the analytical and simulation engines are not strongly bound to the JMT front-end.

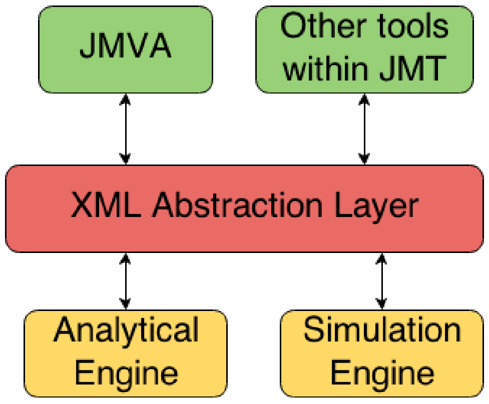


Figure 5.1: JMT basic architecture.

The main packages and classes within JMT that concern this project are described below:

- **gui.exact package** - This package contains all the code for the JMVA GUI interface. The key classes are:
  - **ExactWizard** - This class is the entry point to JMVA's GUI and essentially defines the actions that are run for each of the buttons within the JMVA model input wizard.

  - **ExactModel** - This class represents the queueing network model that is to be solved. In addition to the model inputs (such as station, class and service demand information) it can also store the results for a particular model for several algorithms over many iterations (if 'what-if' mode is used). Importantly the class has the functions `createDocument()` and `loadDocument()` which are capable of exporting or importing respectively to an XML representation of the model. By using this functionality models can be passed back and forth across the XML layer as required.
  - **SolverClient** - Once the user has input their model and clicked on 'solve', a `SolverClient` object, implemented within the `gui.exact.link` package, is invoked. An `ExactModel` object, representing the current model, is passed to the `SolverClient` which then calls `createDocument()` to create an XML representation of the model. This XML file is then saved as a temporary file, ready to be passed to a `SolverDispatcher` which is contained within the analytical engine itself (see below). On completion, the solver within the analytical engine updates the temporary XML file to contain the results. The `SolverClient` therefore waits for the solver to finish and then returns the temporary XML file to the `ExactModel` instance which can then show the results to the user in a solution window. Thus the `SolverClient` is essentially the link between the JMVA GUI and the underlying analytical engine.

- **analytical package** - This package contains all of the solver instances which can be used to solve queueing network models and also a dispatcher which is responsible for invoking the correct solver. The most important classes are:

  - **SolverDispatcher** - This class is called by the `SolverClient` when the user requests that a model within JMVA be solved. On receiving this call, the `SolverDispatcher` populates a new `ExactModel` instance using the temporary XML file containing the model information. Depending on the model type (open, closed, what-if, single or multi-class) and also on the algorithm selected by the user, the `SolverDispatcher` creates an appropriate solver instance and forwards the request to this solver. Once the solver has completed its calculation of performance measures, the `SolverDispatcher` populates the `ExactModel` instance with the results, ready to be converted back into XML format and sent across to the `SolverClient`.
  - **Solver Classes** - Each solver has its own class within the analytical package, with most of the solvers being further separated into multi and single class versions, corresponding respectively to the `MultiSolver` and `Solver` abstract classes in the package. For example, the MVA algorithm is implemented in a class called `SolverSingleClosedMVA`, which extends `Solver` and is capable of solving closed single-class models. There is also a `SolverMultiClosedMVA` class, which extends `MultiSolver` and is capable of solving closed multi-class models. Other solvers include implementations of AMVA methods such as AQL, Bard-Schweitzer, Chow and Linearizer as implemented in [13] by Chugh. Also included are classes which link to the MoM, CoMoM and RECAL implementations contained in the JCoMoM package, some of which were implemented in [3] by Bradshaw.

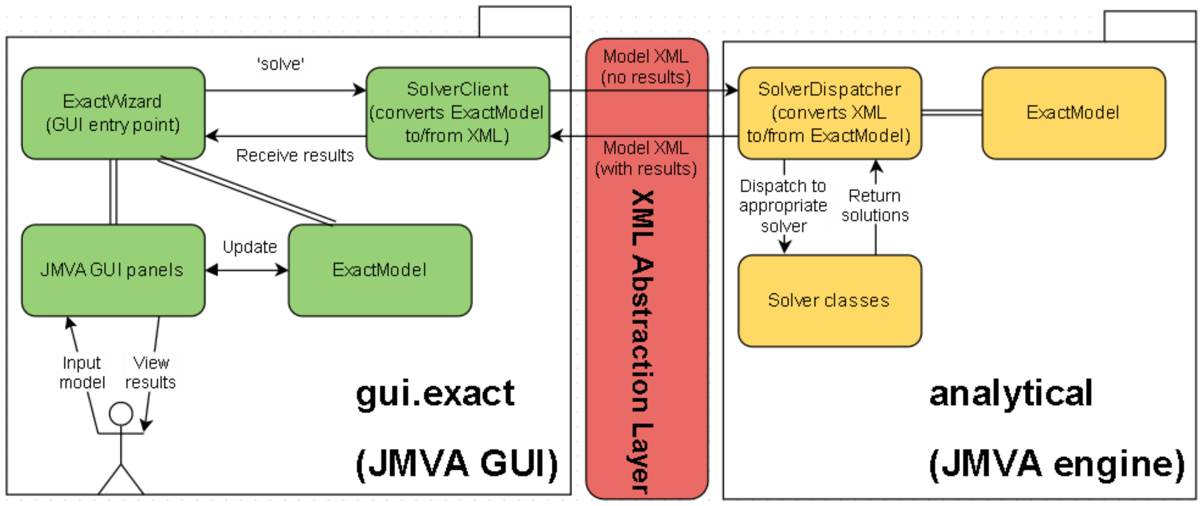


Figure 5.2: JMVA high-level lifecycle.

5.2 Bounding Solvers Implementation

In this project all of the bounding techniques discussed in section 3.1, namely Asymptotic Bounds (ABs), Balanced Job Bounds (BJBs) and Geometric Bounds (GBs), were implemented for closed, single-class models and integrated into JMVA. In addition ABs and BJBs were also implemented for multi-class models (GBs are currently only theoretically possible for the single-class case). In the following subsections we discuss the changes that were made to achieve this and highlight some of the implementation challenges.

5.2.1 Single-class Bounding Solvers Architecture

The single-class AB, BJB and GB solvers themselves each extend a new abstract class called `BoundsSolver` and were added to the `analytical` package with the other solvers. The class structure of the new bounds solvers is shown in Figure 5.3.

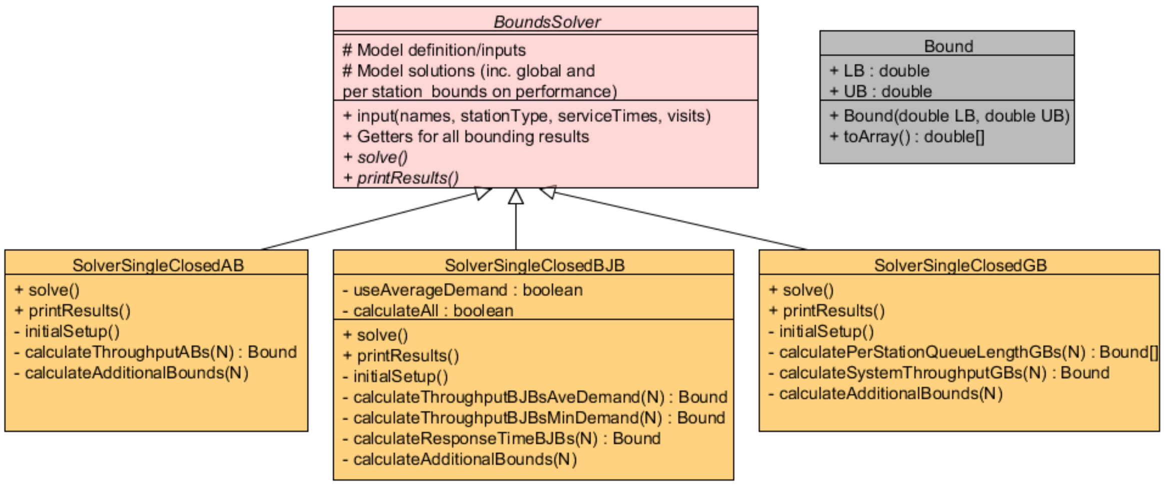

BoundsSolver
# Model definition/inputs
# Model solutions (inc. global and per station bounds on performance)
+ input(names, stationType, serviceTimes, visits)
+ Getters for all bounding results
+ solve()
+ printResults()
Bound
+ LB : double
+ UB : double
+ Bound(double LB, double UB)
+ toArray() : double[]
SolverSingleClosedAB
+ solve()
+ printResults()
- initialSetup()
- calculateThroughputABs(N) : Bound
- calculateAdditionalBounds(N)
SolverSingleClosedBJB
- useAverageDemand : boolean
- calculateAll : boolean
+ solve()
+ printResults()
- initialSetup()
- calculateThroughputBJBsAveDemand(N) : Bound
- calculateThroughputBJBsMinDemand(N) : Bound
- calculateResponseTimeBJBs(N) : Bound
- calculateAdditionalBounds(N)
SolverSingleClosedGB
+ solve()
+ printResults()
- initialSetup()
- calculatePerStationQueueLengthGBs(N) : Bound[]
- calculateSystemThroughputGBs(N) : Bound
- calculateAdditionalBounds(N)

Figure 5.3: Structure of single-class bounds solver classes.

The `BoundsSolver` parent class holds all model input data, for example the number of stations and their service demands, and also holds the outputs, including the various bounds on performance. As shown in the diagram, the bounds are represented by a simple `Bound` object which stores the lower and upper bounds.

For each bounds solver the `solve()` method first calls `initialSetup()`, which in each case carries out some preliminary calculation which is necessary for obtaining the bounds. `solve()` then calculates the throughput bounds (and for GBs also the queue length bounds) before calling `calculateAdditionalBounds()`, which applies simple laws (such as Little's Law and the Forced Flow Law) in order to obtain bounds on the other performance measures, such as residence times, queue lengths and utilisations.

5.2.2 AB Solver

In the AB solver `initialSetup()` calculates `Z`, the total delay of all delay stations, `Dsum`, the sum of the service demands, and `Dmax`, the maximum service demand. The subsequent calculation of throughput bounds is then a trivial implementation of equation (3.3). Once the system throughput bounds have been obtained, the remaining bounds on performance indices are obtained as follows:

- **Utilisation bounds** at each station k are obtained by multiplying the service demand at k, D_k, by the corresponding throughput bound, since $U_k = XD_k$ (obtained from Utilisation Law and Forced Flow Law).
- **Throughput bounds** at each station k are obtained by multiplying the visits at k by the corresponding throughput bound, since $X_k = V_k X$ (Forced Flow Law).
- **Residence time bounds** at each station k are obtained by applying the MVA formula $R_k(N) = D_k(1 + Q_k(N-1))$ for the bounding queue lengths. For ABs the limiting queue lengths are considered to be 0 (lower bound) and N (upper bound). Therefore, the lower bound for $R_k(N)$ is given by $D_k(1+0) = D_k$, and the upper bound is given by $D_k(1 + N - 1) = D_k \times N$.
- **Queue length bounds** at each station k are obtained by multiplying the bounds on system throughput by the residence time bounds for station k, following the equation $Q_k = XR_k$.
- **System queue length bounds and response time bounds** are obtained simply by summing the per station queue length bounds and residence time bounds respectively.

5.2.3 BJB Solver

The BJB solver has two modes of operation, since there are two possible ways of obtaining single-class BJB bounds. If the local boolean `useAverageDemand` is set to true, then the BJBs are calculated based on the average service demand using equation (3.8) (but extended to take into account the total delay Z). On the other hand if `useAverageDemand` is false, then the BJBs are calculated based on the minimum and maximum service demands using an extension of equation (3.5). In general, the BJBs based on average demands will provide tighter bounds for single-class models and are therefore used as the default for this project. However, for multi-class models the minimum/maximum service demand approach must be used (see section 5.2.5).

In view of these two modes of operation, the BJB solver calculates `Dmin`, the minimum service demand, and `Davg`, the average service demand, during the initial setup, in addition to the standard values outlined for the AB solver. The bounds on other performance measures are calculated in a similar fashion to the AB solver, however BJBs cannot be obtained for per station queue lengths and residence times.

5.2.4 GB Solver

The GB solver, which was a main focus for the first part of this project, is slightly more complex than the other solvers due to the nature of the bounding equations. Unlike the other two solvers, both the system throughput and per station queue length bounds are calculated directly. The algorithm for calculating per station queue lengths is shown in algorithm 8 and is based on the equations (3.12) and (3.13).

Algorithm 8 Calculating of per station queue length GBs in Java

```java
/** Calculates per station queue length GBs.
 * @param N The population.
 * PRE: D is an array containing the service demands,
 *      Z is the total delay,
 *      Dsum is the sum of the demands,
 *      Dmax is the maximum demand,
 *      Xmin/Xmax are the minimum/maximum possible throughputs (from BJB solver),
 *      mmax is the number of stations with the maximum demand Dmax. */
private Bound[] calculatePerStationQueueLengthGBs(int N) {
  Bound[] Qtmp = new Bound[stations];
  double Qu_sum = 0.0;
  double Ql_sum = 0.0;
  ArrayList<Integer> maxStationIdxs = new ArrayList<Integer>();

  // Calculate queue length bounds for stations with submaximal service demands.
  for (int k = 0; k < stations; k++) {
    if (D[k] < Dmax) {
      double ykn = D[k]*N/(Z + Dsum + Dmax*N);
      double Ykn = D[k]*Xmax;
      Qtmp[k] = new Bound();
      Qtmp[k].LB = (ykn/(1-ykn)) - (Math.pow(ykn, N+1)/(1-ykn)); // Lower bound
      Qtmp[k].UB = (Ykn/(1-Ykn)) - (Math.pow(Ykn, N+1)/(1-Ykn)); // Upper bound
      Ql_sum += Qtmp[k].LB;
      Qu_sum += Qtmp[k].UB;
    }
    else {
      // Prestore indices for stations with max service demand.
      maxStationIdxs.add(k);
    }
  }

  // Calculate queue length bounds for stations with max service demand.
  if (!maxStationIdxs.isEmpty()) {
      double maxLB = (N - Z*xmax - Qu_sum)/mmax;
      double maxUB = (N - Z*xmin - Ql_sum)/mmax;
      if (maxLB < 0 ) maxLB = 0;
      for (int k : maxStationIdxs) {
        Qtmp[k] = new Bound(maxLB, maxUB);
      }
    }

  return Qtmp;
}
```

Note that some parts of the actual implementation, such as the calculation for delay stations, have been omitted from the above algorithm for clarity. The calculation of throughput bounds follows a similar format to implement equations (3.17) and (3.18).

In order to obtain tighter GBs, it should be noted that `Xmin` and `Xmax` are obtained from the BJB solver's throughput bounds. However, to reduce the cost of calculating `Xmin` and `Xmax`, a flag is set within the BJB solver so that it only calculates the throughput bounds but does not compute bounds for other performance measures, since they are not required in this case.

The calculation of additional GBs on the other performance measures are obtained using the principles outlined for the AB solver. However, when applying the MVA equation to calculate residence times, the per station mean queue length bounds, calculated using the above algorithm, can be substituted in rather than assuming the limiting queue lengths are 0 and N as we did for the ABs. This gives much tighter bounds on per station residence times and hence also on aggregate response time.

5.2.5 Multi-class Bounding Solvers Architecture

Versions of the AB and BJB solvers were also implemented to handle multi-class models. The class structure of these bounds solvers is shown in Figure 5.4.

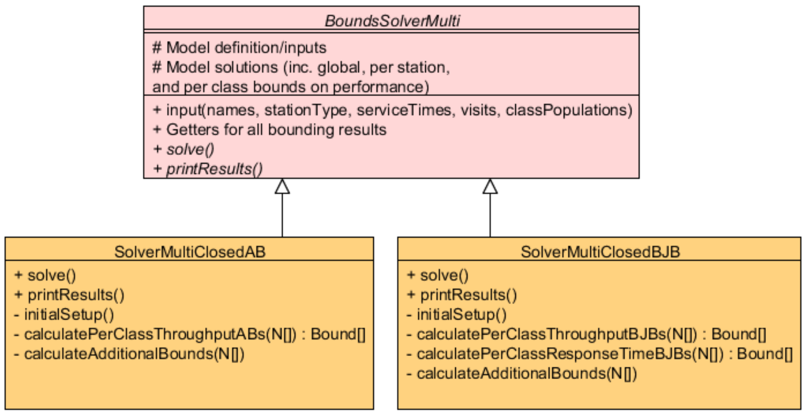

BoundsSolverMulti
# Model definition/inputs
# Model solutions (inc. global, per station, and per class bounds on performance)
+ input(names, stationType, serviceTimes, visits, classPopulations)
+ Getters for all bounding results
+ *solve()*
+ *printResults()*
SolverMultiClosedAB
+ solve()
+ printResults()
- initialSetup()
- calculatePerClassThroughputABs(N[]) : Bound[]
- calculateAdditionalBounds(N[])
SolverMultiClosedBJB
+ solve()
+ printResults()
- initialSetup()
- calculatePerClassThroughputBJBs(N[]) : Bound[]
- calculatePerClassResponseTimeBJBs(N[]) : Bound[]
- calculateAdditionalBounds(N[])

Figure 5.4: Structure of multi-class bounds solver classes.

The overall structure of these solvers is the same as for the single-class case. The only difference is that the bounding equations are expanded to give bounds on a per station per class basis. Aggregate results can then be obtained for per class, per station or whole system measures. We do not discuss the individual implementations further as they are very similar to the single-class bounding solvers already discussed.

5.2.6 Integration with JMVA

The most challenging part of implementing the bounds solvers was integrating them into the existing JMVA architecture. This was due to the fact that all of the preexisting data structures were designed to support algorithms which produce one result per performance measure, whereas bounding algorithms produce two results; the upper and lower bounds. In view of this, suitable data structures had to be introduced to facilitate the introduction of the new bounding class of algorithms. In addition, there were some non-trivial GUI modifications that had to be made, including the creation of new results panels capable of displaying bounding results and also the graphical representation of bounds in JMVA's 'what-if' mode.

In particular the following modifications were made:

analytical package

- The `SolverAlgorithm` enum class, which defines all of the algorithms used in JMVA, was modified to include the new bounding algorithms. This involved creating a completely new 'bounding' class of algorithms, which will also allow the addition of more bounding algorithms in the future.

- The `SolverDispatcher` was also modified to ensure that the new bounding solvers could be accessed from JMVA. If the `SolverDispatcher` receives a request to solve a model and the associated `ExactModel` object specifies that it should be solved using a bounding algorithm, then the correct solver is instantiated within the `SolverDispatcher`, initialised with the model values, and invoked.

gui.exact package

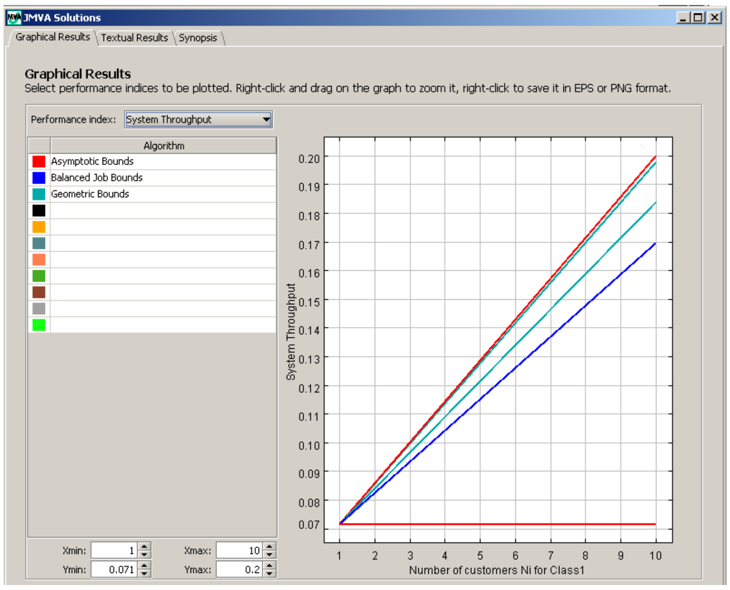
- The `ExactModel` class was modified so that it could store bounding results. This was done by adding a series of Java `Maps`, each mapping an algorithm type to the set of bounding results for a particular performance measure. This echoes the current data structures used for non-bounding results, and ensures that a single `ExactModel` object can store results for multiple algorithms.

- The XML schema, which specifies the structure of model XML files that are passed across the XML abstraction layer, had to be modified to allow the inclusion of bounding results.

- Several new GUI panel implementations were added to the `panels` subpackage. Since the bounding results could not be formatted to fit into any existing results panels, a new solution panel was added for each performance measure. These new panels are capable of displaying both the upper and lower bounds.

- The most complex GUI modification was carried out on the `GraphPanel`, which is responsible for displaying a results graph after 'what-if' analysis has completed in order to show the results of running an algorithm several times over a range of parameters. Again, due to the duality of bounding results, the graph was modified so that it is capable of displaying both upper and lower bounds for a single performance measure.

- Finally the logic which manages the creation of solution windows in the `ExactWizard` class was modified to ensure that the new bounds-specific panels are displayed when a model has been solved using bounding algorithms.



Graphical Results

Select performance indices to be plotted. Right-click and drag on the graph to zoom it, right-click to save it in EPS or PNG format.

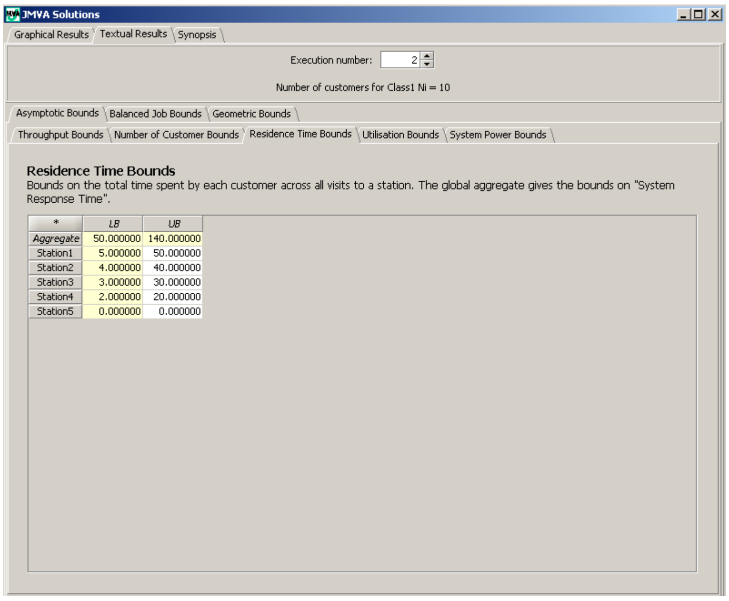
Performance index: System Throughput

	Algorithm
🟥	Asymptotic Bounds
🟦	Balanced Job Bounds
🟦	Geometric Bounds

System Throughput

Number of customers Ni for Class1

Figure 5.5: Graphical bounds comparison.



Execution number: 2

Number of customers for Class1 Ni = 10

Residence Time Bounds

Bounds on the total time spent by each customer across all visits to a station. The global aggregate gives the bounds on "System Response Time".

*	LB	UB
Aggregate	50.000000	140.000000
Station1	5.000000	50.000000
Station2	4.000000	40.000000
Station3	3.000000	30.000000
Station4	2.000000	20.000000
Station5	0.000000	0.000000

Figure 5.6: Textual bounds results in what-if mode.

5.2.7 Testing and Validation

A series of jUnit tests were added to JMT's `test.analytical` package. The tests were separated into two classes, `TestBoundingSolvers`, for testing the single-class bounds solvers, and `TestBoundingSolversMulti`, for testing the multi-class solvers. Each class contains tests over a variety of different networks with varying service demands and populations. Both testing classes can be configured to run the tests over all bounding solvers, or just a specified subset. For each test, the network is first solved using JMVA's existing MVA algorithm and then by the bounds solver under test. We then check that the results from the bounding solver bound the exact results obtained from the MVA algorithm.

5.3 Tree Algorithms Implementation

One of the main goals of this project was to implement both the Tree Convolution (TC) and Tree MVA (TMVA) algorithms. For an overview of the theoretical concepts underpinning these algorithms see section 4. In this section we focus on the design and implementation of the solutions. We also take an in-depth look at of some of the alternatives and optimisations that were tried in order to improve the performance of the algorithms.

5.3.1 Solution Architecture

Before we proceed it should be noted that the core implementations of both the TC and TMVA algorithms were implemented within the JCoMoM package, not directly within JMT, for the reasons mentioned in section 2.4. However, a JAR file containing the binaries for the JCoMoM library is included within the JMT open source distribution, so the solvers implemented within the JCoMoM library are still accessible from JMT, though the source code is not visible.

5.3.1.1 Existing JCoMoM Architecture

We first summarise the roles of the JCoMoM subpackages we are interested in for this project and describe the high-level changes that were made in each case:

- `Control` - This subpackage contains the main entry point class for the commandline interface. This was modified so that the TC and TMVA algorithms can also be accessed from the commandline. A few new options were added to the tool and these are discussed in more detail in section 5.3.10.
- `DataStructures` - This subpackage contains the various data structures used to implement the existing algorithms. This includes a `BigRational` class which uses two `BigIntegers` (Java built-in type) to represent any rational number. There are also several data structures for representing various types of vectors, such as population vectors. The most important class in this subpackage is `QNModel`, which is used to store all the information about a model (such as service demands, number of classes etc.) and also the results after a model has been solved, in particular the throughputs and per station queue lengths (since all other performance measures can be derived from these).
- `QueuingNet` - This subpackage contains all of the previously implemented algorithms, including solvers for standard Convolution, MoM, CoMoM and RECAL. The Tree Convolution and Tree MVA algorithms were implemented in a new `TreeAlgorithms` subpackage within `QueuingNet`.
- `Utilities` - This subpackage contains various useful utility classes, for example methods for computing factorials, printing matrices and copying arrays. A useful `Timer` class allows algorithms to be easily timed, providing a simple way of testing and comparing performance. In this project a new class for generating random networks was added to the `Utilities` package, see section 5.3.9 for more details.

5.3.1.2 Overview of new `TreeAlgorithms` package

As mentioned, the actual implementation of the TC and TMVA algorithms reside within a new `TreeConvolution` subpackage. While studying the two algorithms it became clear that certain aspects were similar and so an effort was made to design the code in such a way that the common elements were easily reusable in both algorithms. In terms of package organisation, all classes which are common to both algorithms reside at the top level, while algorithm-specific classes are placed within two further subpackages called `TMVA` and `TreeConvolution`.

Figure 5.7 below shows the overall structure of the `TreeAlgorithms` package. As can be seen, both algorithm implementations are split up into several smaller modules in order to increase the testability and extensibility of the solution, so each module can easily be swapped for another implementation or mocked for testing purposes. Additionally a consistent, almost symmetrical, structure is used for both algorithms to keep the design simple, maintainable and accessible to anyone who works on the codebase in the future.

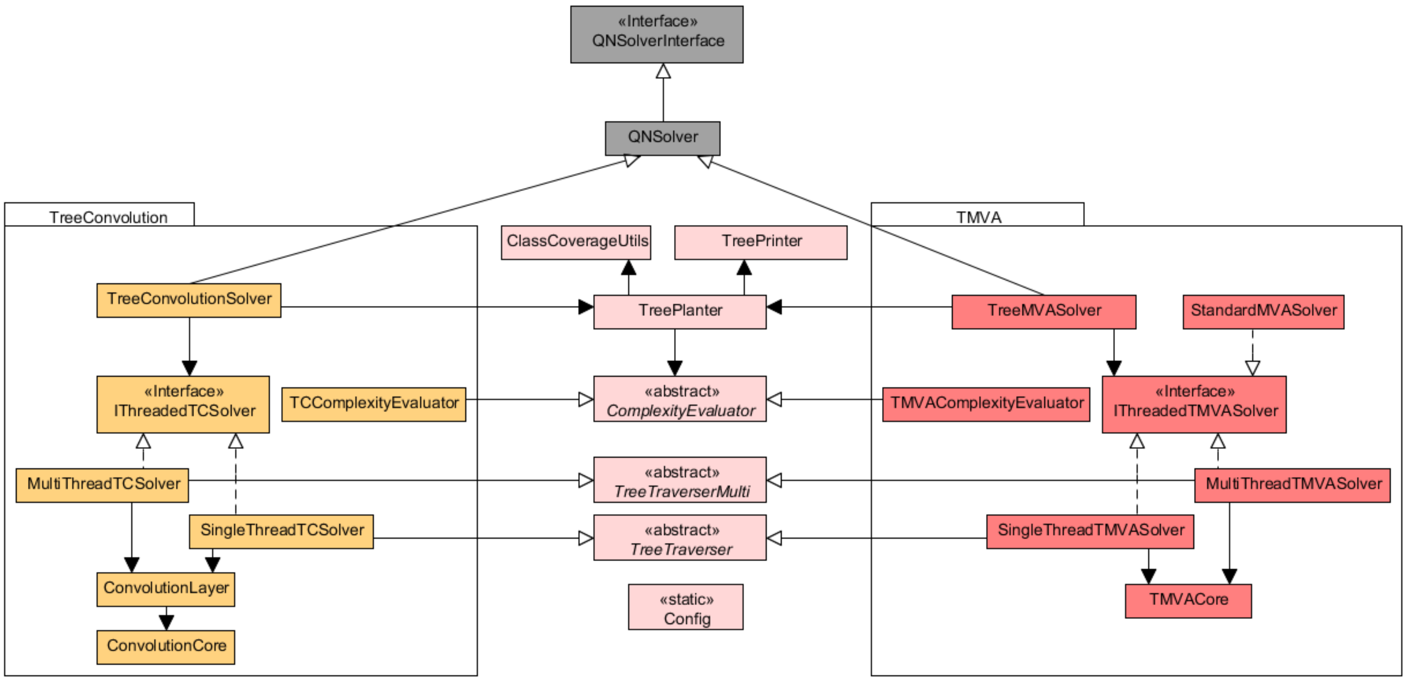


Figure 5.7: Structure of new `TreeAlgorithm` package (pink items are common to both algorithms, grey items are from the existing codebase).

Each subpackage follows a similar hierarchical pattern in which the classes gradually get more specific and fine grained as the dependency chain is followed. At the top of this chain are the `TreeConvolutionSolver` and `TreeMVASolver` classes which are the main entry points for the TC and TMVA solutions respectively. Both of these classes were deliberately kept simple and are mainly responsible for instantiating the required components, mainly the `TreePlanter` and relevant single or multi class solver (depending on a 'number of threads' parameter passed into the constructor). The main solver class then simply delegates to these classes as required in order to solve the model that is passed in.

At the lower levels of the dependency chains are the `ConvolutionLayer`, `ConvolutionCore`, and `TMVACore` classes. These are highly algorithm specific and implement the core elements of each algorithm. In the following sections we look at the responsibilities of each class and demonstrate how the solution fits together as a whole. We start with the aspects common to both TMVA and TC algorithms, including the preprocessor stage and tree traversal, before moving onto the technical details specific to each algorithm.

5.3.2 Algorithm Configuration

In order to centralise all of the user-definable options for the TC and TMVA algorithms a `Config` class was created which contains various static variables for defining how the algorithms should run. It contains boolean flags for general settings, such as for turning on/off console and file logging, in addition to algorithm specific settings, such as which tree planting mode to use and whether to use iterative or recursive tree traversal. Optional parts of the algorithms, such as feedback filtering for Tree Convolution, can be switched on or off easily. The intention behind this class is to allow different variants of the algorithms to be compared easily; simply by changing the required flags, rather than having to modify any of the algorithms' core logic.

5.3.3 Preprocessor Stage

As outlined in section 4 the preprocessor stage is the initial setup phase for both the TMVA and TC algorithms and has a relatively low complexity compared to the main performance measure calculation stages. The preprocessor stage can be further divided into the tree planting stage, implemented by the `TreePlanter` class, and the complexity evaluation phase, implemented by the abstract `ComplexityEvaluator` class and its algorithm-specific subclasses. As can be seen from figure 5.7 above, the classes concerned with preprocessing are shared by both algorithms, which is why we discuss them first.

5.3.3.1 `TreePlanter`

The `TreePlanter` class implements the initial tree planting phase for both TC and TMVA algorithms. The theoretical background for heuristic tree planting was discussed in section 4.3, so we focus here on the implementation details. The main method within the `TreePlanter` is `runTreePlantingPhase()` which is shown in Algorithm 9 below. It can be seen that the tree planting phase consists of two main stages; firstly a tree is planted using one of the implemented methods (lines 17-33), and secondly the tree planter delegates to the `ComplexityEvaluator` to check whether the tree has an acceptable complexity (line 37). If a suitable tree is found the method returns true and the main stage of the algorithm can proceed. If a suitable tree is not found after a certain number of planting attempts, defined by the `MAX_TREE_PLANT_ATTEMPTS` in the `Config` class, then the algorithm returns false and the parent solver throws a suitable error.

As the code suggests two modes of operation have been implemented which are determined by the flag `COMPARE_ALL_HEURISTICS`. If this flag is true then we plant a tree using all implemented methods and then let the `ComplexityEvaluator` determine which tree has the lowest complexity. If the flag is false then only one of the planting methods is run, again determined by a constant in the `Config` file. Interestingly it was found that the heuristic algorithm, as suggested by Lam and Lien, does not always produce a better tree than simpler planting methods, which is why the option to compare all planting methods was implemented.

Currently three planting approaches have been implemented:

1. `Heuristic` - This is an implementation of Lam and Lien's heuristic approach for which the pseudocode was outlined in section 4.3. Though a few modifications were required in order to adapt their approach for unbalanced trees.

2. `Simple` - This is a very naive planting approach which simply places the stations in order at the leaf nodes. Although this method was initially implemented for testing purposes, it surprisingly sometimes outperforms the more complex heuristic approach. Of course this is heavily dependent on the network being tested.

3. `Sequential` - This is another simple approach which plants the tree such that the Tree Convolution algorithm runs the exact same order of convolution operations as the sequen-

tial Convolution algorithm. This was mainly implemented for interest's sake and very rarely outperforms the heuristic approach.

Algorithm 9 High-Level Tree Planting and Complexity Evaluation Phase Algorithm

```
/**
 * Runs the main tree planting phase. Consists of two steps:
 * 1. Plant the tree.
 * 2. Check this tree has acceptable complexity characteristics.
 * N.B. 'ce' is the local ComplexityEvaluator object,
 * N is the model's population vector.
 * @return A boolean indicating whether an adequate tree was found.
 */
 public boolean runTreePlantingPhase() {
   boolean suitableTreeFound = false;
   int round = 0;
   while(!suitableTreeFound) {
     if (round >= Config.MAX_TREE_PLANT_ATTEMPTS) {
       return false;
     }

     if (Config.COMPARE_ALL_HEURISTICS) {
       Node[] trees = new Node[3];
       trees[0] = plantTreeHeuristicApproach();
       trees[1] = plantSimpleTree();
       trees[2] = plantSequentialTree();
       treeRoot = ce.getTreeWithBestComplexity(trees, N);
     } else {
       switch (Config.TREE_PLANT_MODE) {
         case HEURISTIC:
           treeRoot = plantTreeHeuristicApproach();
           break;
         case SIMPLE:
           treeRoot = plantSimpleTree();
           break;
         case SEQUENTIAL:
           treeRoot = plantSequentialTree();
           break;
       }
     }

     suitableTreeFound = ce.treeHasAcceptableComplexity(treeRoot, N);
     round++;
   }

   return true;
 }
```

Each node in a planted tree is represented by a `Node` object which contains links to its parent and child nodes, a set of stations (giving the subnetwork represented by the node), and algorithm-specific information which is calculated and stored as the algorithm progresses. Each node also stores the set of partially and fully covered classes at that node in order to keep recomputation to a minimum. The `Node` class also contains various useful functions such as for recalculating the partially covered classes and merging a node with another node.

As Figure 5.7 showed, the `TreePlanter` also links to a `TreePrinter` object which provides a simple algorithm that traverses the planted tree and prints each node to the console for debugging purposes. It also links to a `ClassCoverageUtils` class which contains various utility methods for calculating the coverage status of a certain class of jobs with respect to a subnetwork of nodes in the tree (see section 4.1.1 for an explanation of class coverage).

5.3.3.2 `ComplexityEvaluator`

The `ComplexityEvaluator`, which is used during the preprocessor stage to evaluate a planted tree's complexity, is an abstract class which provides a core implementation shared by both TC and TMVA algorithms. In order to compute the complexity of a planted tree, the tree is traversed and the complexities associated with each merging stage, either in the TMVA or TC algorithms, are added up. To make this simpler an internal `ComplexityBundle` class is used to store both time and space complexities simultaneously. A function `getTreeComplexity()` in `ComplexityEvaluator` implements this traversal, however the complexity evaluation for each node is left for the child classes, `TCComplexityEvaluator` and `TMVAComplexityEvaluator`, which are capable of calculating the algorithm-specific complexities. The actual equations used to compute the complexities for TC and TMVA are given in sections 4.1.7 and 4.2.3 respectively.

Additionally, as we saw in the last code snippet, the `ComplexityEvaluator` has a function `getTreeWithBestComplexity()` which takes in several trees and compares them. The way in which the trees are compared depends on a user-definable parameter in the `Config` class. Currently trees can be compared by estimated time complexity, space complexity or simply by counting the number of partially covered classes in the tree (since trees with lower numbers of partially covered classes are generally quicker to evaluate).

As well as computing the complexities for TC and TMVA, the algorithm-specific child classes, `TCComplexityEvaluator` and `TMVAComplexityEvaluator`, are also capable of calculating estimated complexities for the standard Convolution and MVA algorithms. If a special flag called `TERMINATE_IF_HIGH_COMPLEXITY` is set in the `Config` file then the algorithm will terminate during the preprocessor stage if the `ComplexityEvaluator` decides that the tree version of the algorithm actually has a higher complexity than the sequential version. A simpler way of measuring the applicability of the tree algorithms was also developed which focuses on measuring the sparsity of the model, rather than the algorithm complexity. This approach is discussed in section 5.3.8.

5.3.4 Tree Traversal

The traversal of the tree planted during the preprocessor stage is, of course, integral to both Tree Convolution and Tree MVA. In view of this it was decided to create two generic traversal classes which can be used by either algorithm. The first class, `TreeTraverser`, implements single-threaded traversal and is extended by the TC and TMVA single-threaded solvers, while the second, `TreeTraverserMulti`, implements multi-threaded traversal and is extended by the multi-threaded solvers, as can be seen in Figure 5.7.

5.3.4.1 Single-threaded Traversal

The `TreeTraverser` class implements both recursive and iterative single-threaded, postorder tree traversal. Postorder traversal was adopted since it is the preferred traversal order used in Lam and Lien's paper [21]. Algorithm 10 below shows the recursive version (since it is visually much cleaner than the iterative version). It can be seen that the actual computation performed at nodes during the traversal is delegated to abstract methods which are then overwritten in the single-threaded solvers that extend `TreeTraverser`, so as to provide algorithm-specific implementations. This allows the tree traversal code to be reused by both algorithms. Whether iterative or recursive traversal is used depends on a user-definable constant in the `Config` class. Though it is recommended that the iterative version is used for large networks to avoid stack overflow errors due to high recursion depth.

Algorithm 10 Code snippet from `TreeTraverser` for recursive traversal.

```
1   /** Traverses a tree from specified root node,
2     * in single-threaded, postorder fashion.
3     * @param node The current node.
4     * @param pv The population vector to be used during the computation.
5     */
6    public void recursiveTraverse(Node node, PopulationVector pv) {
7      if (node.leftChild() != null) {
8        recursiveTraverse(node.leftChild(), pv);
9      }
10     if (node.rightChild() != null) {
11       recursiveTraverse(node.rightChild(), pv);
12     }
13
14     if (node.isLeaf()) {
15       initLeafNode(node, pv);
16     } else {
17       computeSubnetNode(node, pv);
18     }
19   }
20
21   public abstract void initLeafNode(Node node, PopulationVector pv);
22   public abstract void computeSubnetNode(Node node, PopulationVector pv);
```

5.3.4.2 Multi-threaded Traversal

The `TreeTraverserMulti` class implements multi-threaded recursive tree traversal, which is used by the multi-threaded TC and TMVA solvers. Before traversal can begin a fixed thread pool is defined, with the maximum size of the pool being dependent on the number of cores available to the JVM. The recursive traversal approach is similar to the single-threaded algorithm shown above, however the multi-threaded version is more interesting since it involves thread synchronisation. In particular, the following rule must hold; each non-leaf node in the tree can only be evaluated if all of its child nodes have been evaluated. In practice the algorithm is modified to return a Java `Future` object which represents the result of an asynchronous computation. In the modified algorithm the results from lines 8 and 11 (in Algorithm 10 above) are stored in a `Future` object array, which represents the current node's child thread computations. At line 13 a small amount of code is added which waits for the child threads to complete before the current node is processed (lines 14-18). Fortunately Java's `ExecutorService` manages the thread pool for us, making the implementation fairly clean and understandable.

5.3.5 Tree Convolution Implementation

Now that the generic classes used by both TC and TMVA have been explained we are in a position to look at the finer technical details for each algorithm. We start with Tree Convolution since this was implemented first. For a theoretical background on Tree Convolution see section 4.1.

The entry point for the Tree Convolution implementation is the `TreeConvolutionSolver` class which extends the `QNSolver` class from earlier projects and defines several methods every queueing network solver must implement. In particular, the `TreeConvolutionSolver` overrides the functions `computeNormalisingConstant()` and `computePerformanceMeasures()`. The constructor of `TreeConvolutionSolver` takes in a representation of the model we are trying to solve and also the number of threads to use. It then uses dependency injection to instantiate all of the required components, and importantly it spawns either a single-threaded or multi-threaded solver instance depending on the number of available threads. In order to compute the normalising constant and performance measures it delegates to these components and, on completion, sets the results. A primary goal during the design phase was to keep this class as simple as possible which was achieved by encapsulating the important phases of the algorithm in separate classes. This had the added bonus of allowing specific parts of the algorithm to be implemented in several different ways and then compared, which was invaluable when optimising the algorithm's performance.

The `SingleThreadTCSolver` and `MultiThreadTCSolver` essentially carry out the same operations except the multi-threaded version extends the `TreeTraverserMulti` class, allowing the phases of the computation to be done in parallel across multiple cores, as discussed in the previous section. We focus on the single-threaded solver to facilitate a simpler explanation of the lower level algorithm implementation.

The single-threaded solver contains the high-level structure for computing the normalization constant, per class throughputs and per class per station mean queue lengths. As we saw in section 4.1.5 there are several methods by which the performance measures can be computed for Tree Convolution. After some experimentation with storing multiple g-arrays at a single node in order to reduce the number of tree traversals required, as suggested by Lam and Lien in [21], it was found that this did not make a significant difference to the overall computation time. This was mainly because tree traversal is not a significant factor in determining the algorithm's run time. Rather it is the overall number of computations carried out at each node, which depends primarily on the number of partially covered classes at a given node. Nonetheless both a single and multi g-array storage method was implemented for the computation of mean queue lengths. The method that is used can be changed simply by changing a flag called `Q_COMPUTE_SEQUENTIALLY` in the `Config` class. For computing throughputs, only the first method given in section 4.1.5 was implemented, since it saves some space compared to the second suggested method. The computation of throughputs is generally much faster than the queue length computation, which tended to be a bottleneck in the algorithm. For this reason more effort was put into optimising the queue length calculation by trying out various possible implementations.

5.3.5.1 `ConvolutionLayer`

Since a few different approaches were implemented for the computation of performance measures (see section 4.1.5), it was decided to split the core convolution code into two classes; `ConvolutionLayer`, which contains methods for running the different approaches, and `ConvolutionCore`, which implements the low level convolution equations used by all approaches. The `ConvolutionLayer` therefore acts as an intermediate layer between the solvers and the `ConvolutionCore` implementation. This allowed different strategies to be implemented simultaneously and compared for optimisation purposes. The approaches that are used depend on user-definable flags in the `Config` class.

5.3.5.2 `ConvolutionCore`

The `ConvolutionCore` contains implementations for the most low-level convolution operations, for example computing the g-arrays at leaf nodes and subnetwork nodes using the standard convolution equations (implementations of equations (4.3) and (3.24) respectively). In addition it also contains an implementation of the *feedback filtering* approach (see section 4.1.4), which can be applied at a node if one of its children is a leaf node representing a load independent station. Feedback filtering was found to dramatically improve the performance of the algorithm for certain networks. In particular, the computation of mean queue lengths, which involves cloning leaf nodes and performing convolutions from these new leaf nodes to the root, was significantly faster once feedback filtering had been implemented.

5.3.5.3 Memory Management

An important aspect of the Tree Convolution algorithm is how to store the intermediate results at nodes in order to make space savings. In this implementation the g-arrays are stored as a `HashMap<PopulationVector, BigRational>` within each `Node` object, storing a value for each possible population vector given the partially covered classes at that node. For example, if classes 1 and 2 are partially covered at a node and both have a maximum population of 1, then values are calculated for the following population vectors: (0,0), (0,1), (1,0), (1,1), where the first number is the population of class 1 and the second of class 2. These values are then stored in the `HashMap`. At the root node there will be no partially covered classes and the g-array will contain a single value corresponding to the normalisation constant, which can be retrieved using a `getG()` function on the root `Node` object. The important point to note is that we only store results for partially covered classes, which is how Tree Convolution saves space compared to sequential Convolution. In practice it was found that some networks which previously caused `OutOfMemoryExceptions` when evaluated with sequential solvers, can now be solved successfully, indicating that Tree Convolution is more memory efficient than standard Convolution for sparse networks, as expected. This claim is justified in chapter 6.

5.3.5.4 Optimisation Log

In order to give an impression of how the Tree Convolution algorithm improved as the implementation progressed a diary of changes and algorithm runtimes was kept during the implementation phase. Note that these times were obtained using a small example network given in the Lam and Lien paper [21].

#	Time	Notes
1	345ms	Time before any optimisation applied, lots of unnecessary computation.
2	329ms	Improved throughput computation, implemented method 1 outlined in section 4.1.5.
3	209ms	Better mean queue length computation.
4	176ms	After refactoring code into modules and making small loop optimisations.
5	140ms	After refining mean queue length computation, only recalculate leaf nodes when required.
6	125ms	After refining mean queue length computation further. Only calculate the leaf values once for full population, when it is required, then reuse the values in subsequent computations.
7	93ms	Only recalculate for partially covered classes when calculating mean queue lengths.
8	62ms	After fully implementing feedback filtering (see section 4.1.4).
9	41ms	After creating multi-threaded version.

The fact that the network under consideration was so small (only four stations), means that the time required for evaluation even before optimisations were applied was well under a second. However, the above table shows how by implementing various optimisations, the runtime was drastically reduced. Of course if these tests were performed on a larger network then these improvements would have a much greater influence overall. Note that for larger models the multi-threaded approach will have a much more significant impact, since, in this case, the small model under consideration provides little opportunity for parallel computation.

5.3.6 Tree MVA Implementation

The Tree MVA algorithm was more difficult to implement than Tree Convolution, mainly due to the fact that the relevant papers by Tucci [34] and Hoyme [17] were less clear about some of the algorithm details. For example Hoyme's paper, although it provides a clear and concise outline of the algorithm, hardly mentions how partially covered classes should be integrated into the algorithm. Tucci, on the other hand, gives a detailed mathematical presentation of the algorithm, but how it all fits together is more difficult to follow. The final implementation was only possible by studying both papers carefully and drawing important aspects from each. For a theoretical overview of TMVA see section 4.2.

In a similar fashion to Tree Convolution, TMVA's main entry point, the `TreeMVASolver`, extends the `QNSolver` interface and is responsible for instantiating either a single-threaded or multi-threaded solver and other components, such as the `TreePlanter`, depending on the inputs to the constructor. Since MVA does not involve the computation of a normalization constant, only the method `computePerformaceMeasures()` from `QNSolver` is overridden. This method runs the tree planting stage and then delegates to the instantiated solver.

The `SingleThreadTMVASolver` and `MultiThreadTMVASolver` classes are somewhat simpler than their Tree Convolution counterparts, due to the fact that for TMVA all performance measures are computed in one tree traversal. This is unlike Tree Convolution which requires that various partial traversals are performed to compute performance measures after the initial traversal, as we have seen. Therefore, the TMVA single and multi-threaded solvers simply extend the generic tree traversal classes, `TreeTraverser` and `TreeTraverserMulti` respectively, and define the computations which are performed at the leaf and subnetwork nodes. This is done by delegating to a class called `TMVACore`, which implements all of the TMVA equations.

5.3.6.1 `TMVACore`

The vast majority of development time for TMVA was spent on the `TMVACore` class, implementing the fundamental equations. This was partly due to the fact that much of the general infrastructure for Tree Convolution could be reused for TMVA, but also indicative of the amount of careful reading and testing which was required to implement the TMVA equations accurately.

As was mentioned briefly in the background section for TMVA, there are differences between Tucci and Hoyme's versions of the algorithm. In Tucci's version, sequential MVA is applied to all leaf node pairs and then the TMVA equations are applied for each subnetwork node higher up the tree. In Hoyme's version, the leaf nodes themselves are initialised and then the TMVA equations are applied at every subnetwork node, including the leaf parents. Both versions were implemented and compared. In practice it was found that Tucci's version performed slightly better, though for networks containing only one station and for certain special cases involving unbalanced trees, Hoyme's initialization ideas were kept as part of the core implementation.

To begin with the main focus was on implementing the residence time equation (4.14), the marginal probability equations (4.15, 4.16) and the equation to adjust mean queue lengths (4.18). However, the greatest challenge was combining these equations into a single subnetwork TMVA merge function, and ensuring that the loops were all carried out only over the relevant partially covered classes. In Algorithm 11 on the next page, a shortened version of the main TMVA merge function is presented. This shows how a single merge step is done at a subnetwork node. The node that is passed into the function, `ab`, is the parent node we are trying to obtain throughput and queue lengths results for. The rest of the code can be summarised as follows:

- **Line 4** - we follow Tucci's approach and use sequential MVA if the current node is the parent of two leaf nodes.
- **Lines 9-14** - we get the covered classes for the parent node and its two children (covered classes include both partially and fully covered classes at a node) and then find the common partially covered classes between the children by computing the union.
- **Line 16** - a `HashMap` is initialized ready to store marginal probabilities so they can be easily reused. This is one of the optimisations that was described in section 4.2.2. We only store the probabilities for the right hand child since the main equations can all be rearranged to only use these probabilities, saving further calculation time.
- **Line 21** - we see a convenient use of the `ClassCoverageUtils` class (which is referred to in the code by 'ccu'). `ClassCoverageUtils` implements several useful methods for dealing with sets of classes. At line 21 we see an example of the `contract()` function which is used to reduce one population vector down to a certain size. Here the population vector for all classes is reduced down to a vector the size of the common partially covered classes.
- **Line 24** - the main loop starts. It loops over the population vectors for all partially covered classes, starting from the zero vector and ending with the maximum population for each class.
- **Line 26** - a new `MVAResults` object is created, which is just a convenient wrapper object for storing throughputs, queue lengths and response times.
- **Lines 32 and 35** - the residence times for the left and right child are computed by calling a function which implements equation (4.14). Note that `pStore`, the cache of previously calculated marginal probabilities, is passed in so that they can be reused, saving time.
- **Line 39** - the throughput at the parent node is computed by calling a function which implements equation (4.17).
- **Line 43** - the results so far (i.e. the throughputs) are stored at the parent node.
- **Line 46** - this is where the code for adjusting the throughputs and mean queue lengths is in the real implementation. This is done using equation (4.18), which can be applied to both queue lengths (as shown) and also to throughputs.
- **Line 49** - a handy function which returns the next population vector permutation in sequence, given the current vector and the maximum vector we are trying to reach, is called in the `ClassCoverageUtils` class.
- **Lines 53-54** - the results in the child nodes are cleared out to save memory since they are no longer required (note that this is not as simple for the Tree Convolution algorithm since the computation of performance measures requires more than one tree traversal). Also the `pStore` mapping holding the results of marginal probabilities goes out of scope at the end of this function, meaning they will also be removed from memory by the garbage collector. This is desirable since the marginal probabilities for a given node will not be used again.

Algorithm 11 Code snippet from `TMVACore` showing part of TMVA merge algorithm.

```
public void computeSubnetNode(Node ab) {
  Node a = ab.leftChild(); Node b = ab.rightChild();
  // Use sequential MVA at parent of two leaf nodes as Tucci suggests.
  if (ab.childrenAreLeaves()) {
    sequentialMVASolver.solve(ab, qnm.N); return;
  }

  // Get covered classes and find common partially covered classes at child nodes.
  TreeSet<Integer> ab_cs, a_cs, b_cs, commonPcs;
  ab_cs = ab.getAllCoveredClasses();
  a_cs = a.getAllCoveredClasses();
  b_cs = b.getAllCoveredClasses();
  commonPcs = new TreeSet<Integer>(a.pcs);
  commonPcs.addAll(b.pcs);

  HashMap<PTuple, Double> pStore = new HashMap<PTuple, Double>();
  int ri, rx;
  double wa, wb;
  PopulationVector n_ab, npc, npcmax;
  npc = new PopulationVector(0, commonPcs.size());
  npcmax = ccu.contract(qnm.N, ccu.all, commonPcs);

  // Main loop over common partially covered class populations.
  while (npc != null) {
    n_ab = ccu.expand(npc, commonPcs, qnm.N, ab_cs);
    MVAResults resAB = new MVAResults(ab.stations.size(), ab_cs.size());

    ri = 0;
    for (int r : commonPcs) {
      wa = wb = 0;
      if (a_cs.contains(r)) {
        wa = calcResidenceTimeGeneric(a, true, r, npc, pStore);
      }
      if (b_cs.contains(r)) {
        wb = calcResidenceTimeGeneric(b, false, r, npc, pStore);
      }

      rx = ccu.convertIndex(ri, commonPcs, ab_cs);
      resAB.X[rx] = calcSubnetThroughput(rx, n_ab, wa, wb);
      ri++;
    }

    ab.mvaRes.put(n_ab, resAB);

    for (int r : ab_cs) {
      // Code for adjusting throughput and mean queue lengths goes here.
    }

    npc = ccu.nextPermutationUpwards(npc, npcmax);
  }

  // Clear out child node results, not required anymore.
  a.mvaRes.clear();
  b.mvaRes.clear();
}
```

5.3.6.2 Ported sequential MVA solver

It should be noted that in order to implement Tucci's version of the TMVA algorithm, an adaptation of the sequential MVA algorithm implemented within JMVA was created within a class called `StandardMVASolver`. This class contains a ported version of the sequential MVA algorithm, updated to use some of the data structures which are used throughout the JCoMoM package. At line 5 in the above code, this sequential MVA algorithm is used as Tucci suggests. However, having a local implementation of the sequential MVA algorithm was also useful for testing (see section 5.3.11).

5.3.6.3 Optimisation Log

As for Tree Convolution, a log of changes and runtimes was kept during the implementation of TMVA to give an impression of the algorithm's progression towards its final state. For the sake of comparison, these times were once again obtained using the small example network given in the Lam and Lien paper [21].

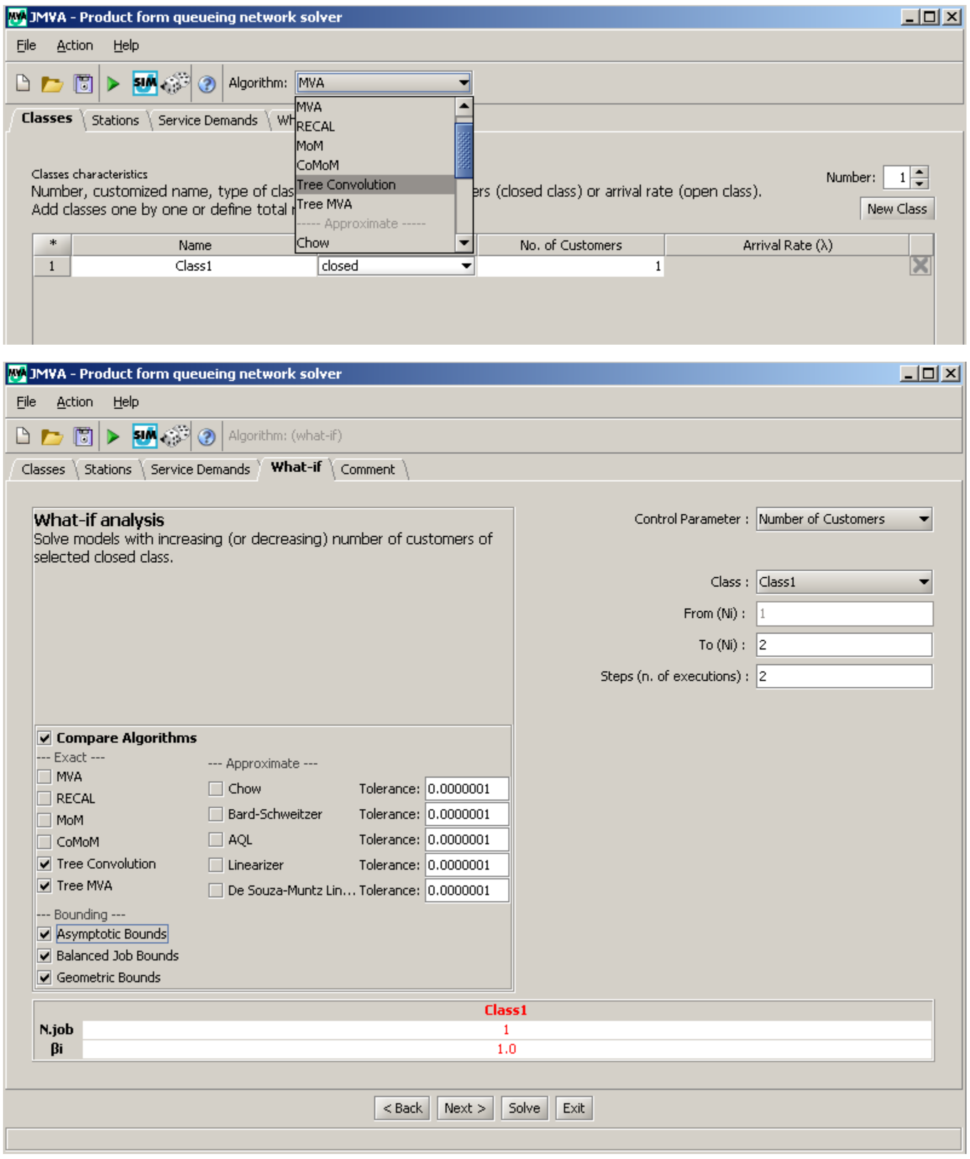
#	Time	Notes
1	495ms	Time before any optimisation applied, lots of unnecessary computation.
2	281ms	Store marginal probabilities in Map once calculated, then reuse to save time. Also rearrange equations to only depend on probabilities at a single child node (see section 4.2.2).
3	182ms	Remove unnecessary code and implement better leaf node computation (again see section 4.2.2).
4	29ms	Apply loop optimizations, only loop over common partially covered classes (this was the the most difficult part of the algorithm to figure out but was definitely worth the effort, allowing a substantial saving in time).
5	16ms	After creating multi-threaded version.

As we noted before, the small size of the network under consideration makes these improvements look relatively insignificant (since the algorithm ran in under a half a second to begin with). However, when scaled to a larger network, the savings in time are substantial. Additionally, the multi-threaded solver will have much more effect on larger networks where there is more chance to exploit multi-core, parallel computation.

5.3.7 Integration with JMVA

Integrating both Tree Convolution and Tree MVA into JMVA was not as difficult as integrating the bounding analysis solvers, since most of the back-end infrastructure was already in place. As both algorithms were implemented in the external JCoMoM package, two new classes were added to the JMVA `analytical` package, `SolverMultiClosedTreeConvolution` and `SolverMultiClosedTreeMVA`, to act as intermediaries, allowing the new solvers to be easily linked in with the existing JMVA infrastructure. These classes simply take in an input model and then delegate to the relevant solver in the JCoMoM package. When the solver has completed its evaluation, the results are extracted from the model so that they can be used and displayed within JMVA.

In terms of changes to the GUI, the two new algorithms were added to JMVA's main algorithm drop-down menu and also to the what-if panel, as can be seen Figure 5.8. Additionally, a simple options panel was added, as shown in Figure 5.9. This allows various tree algorithm configuration settings to be modified within JMVA itself, since users will not have access to the `Config` class as it is part of the private JCoMoM package.


JMVA - Product form queueing network solver

File Action Help

Algorithm: MVA

MVA
RECAL
MoM
CoMoM
Tree Convolution
Tree MVA
----- Approximate -----
Chow

Classes | Stations | Service Demands | Wh

Classes characteristics
Number, customized name, type of clas… …rs (closed class) or arrival rate (open class).
Add classes one by one or define total …

Number: 1
New Class

*	Name	Type	No. of Customers	Arrival Rate (λ)
1	Class1	closed	1	

JMVA - Product form queueing network solver

File Action Help

Algorithm: (what-if)

Classes | Stations | Service Demands | What-if | Comment

What-if analysis
Solve models with increasing (or decreasing) number of customers of selected closed class.

Control Parameter : Number of Customers
Class : Class1
From (Ni) : 1
To (Ni) : 2
Steps (n. of executions) : 2

☑ **Compare Algorithms**

--- Exact ---
☐ MVA
☐ RECAL
☐ MoM
☐ CoMoM
☑ Tree Convolution
☑ Tree MVA

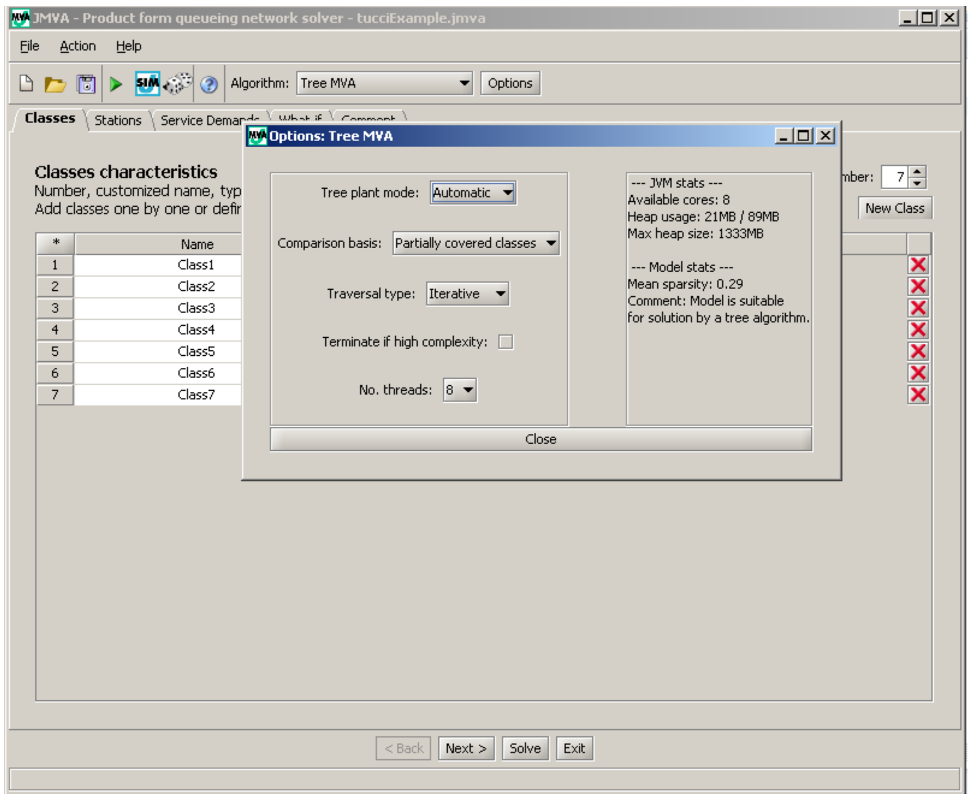
--- Approximate ---
☐ Chow Tolerance: 0.0000001
☐ Bard-Schweitzer Tolerance: 0.0000001
☐ AQL Tolerance: 0.0000001
☐ Linearizer Tolerance: 0.0000001
☐ De Souza-Muntz Lin… Tolerance: 0.0000001

--- Bounding ---
☑ Asymptotic Bounds
☑ Balanced Job Bounds
☑ Geometric Bounds

	Class1
N.job	1
βi	1.0

< Back | Next > | Solve | Exit

Figure 5.8: JMVA screenshots, showing integration of TMVA and TC solvers (ticked items were introduced in this project).



Algorithm: Tree MVA

Classes characteristics
Number, customized name, typ
Add classes one by one or defir

mber: 7

*	Name
1	Class1
2	Class2
3	Class3
4	Class4
5	Class5
6	Class6
7	Class7

Tree plant mode: Automatic

Comparison basis: Partially covered classes

Traversal type: Iterative

Terminate if high complexity: ☐

No. threads: 8

```
--- JVM stats ---
Available cores: 8
Heap usage: 21MB / 89MB
Max heap size: 1333MB

--- Model stats ---
Mean sparsity: 0.29
Comment: Model is suitable
for solution by a tree algorithm.
```

Figure 5.9: JMVA screenshot, showing new tree algorithm configuration panel.

5.3.8 Measuring Sparsity

In order to measure the applicability of the tree algorithms for a given network, a way of measuring the network's sparsity was developed. The 'mean sparsity' of the network is calculated as follows. For each station in the network we count the number of classes that visit that station and then divide this by the total number of classes in the network. This gives a per class sparsity measure between 0 and 1. We then take the average of these values to get the network mean sparsity. As can be seen in Figure 5.9, this measure was integrated into the new options panel so that users can use it to decide which algorithm is the best for the network they are trying to solve. In general, the higher the mean sparsity of a network, the less likely it is that the tree algorithms will provide a more efficient solution than a sequential algorithm such as MVA or Convolution.

5.3.9 Generating Random Networks

Since Tree Convolution and Tree MVA are primarily algorithms for sparse networks, a way of generating such networks in a randomised fashion was desirable for testing purposes. Therefore, a new class `RandomNetworkGenerator` was added to the JCoMoM `Utilities` package which is capable of generating networks with random stations, classes, populations and service demands within specified ranges. In addition, a sparsity percentage can be specified which, for example, if set to 50% will ensure that the generator creates a network in which about half of the service demands are set to zero. The current implementation of this generator is fairly rudimentary in that it does not fully guarantee that the generated network will exhibit the sparseness property even if the sparsity percentage is set high; it only guarantees that the service demands matrix will have a certain sparsity. The two ideas are not the same, though there is a correlation. For the purposes of testing, however, this provided an adequate approximation.

5.3.10 Commandline Tool

An existing commandline tool, in JCoMoM's `Control` package, provides a simple way of running the algorithms within the package. Its usage can be summarised as:

```
java -jar MoM.jar <Algorithm> <Output Indices> <Input File> [<No. of Threads>]
```

where `<Algorithm>` specifies the algorithm to be used, `<Output Indices>` specifies whether the performance measures should be calculated and printed out, `<Input File>` points to a file containing the definition of the model to be solved, and `<No. of Threads>` is an optional parameter specifying the number of threads to run the algorithm with (which defaults to the number of processors available to the JVM).

The commandline tool was extended to include the two new tree algorithms. It was also modified to allow easy access to the random network generation tool which was outlined in the previous section. If the `<Input File>` parameter is set to the word "random", the random network generator is invoked so that the specified algorithm is run on a random model. The number of stations and classes, and also the sparsity percentage of the random model, can be specified as extra parameters to the commandline tool, making it very easy to generate random models of varying sizes.

5.3.11 Testing and Validation

Both the Tree MVA and Tree Convolution algorithms were extensively unit tested. The tests for both algorithms can be found in a `Tests` subpackage within the `QueuingNet.TreeAlgorithms` package. A class called `GeneralTests` contains various tests on the networks used as examples in Lam and Lien [21] and Tucci [34] and also incremental tests, starting from single station/class networks and working up to multiple station/class networks with varying service demands and populations. Unbalanced trees and randomly generated networks are also used in some of the tests to ensure a more complete coverage.

The class `TestUtils` contains useful functionality for testing, including methods which can test the consistency of the results when the same network is evaluated by two different solvers. In fact this idea forms the backbone of the testing framework. The class `AllTests` runs tests for each permutation of solver pairs. For example, we can test sequential Convolution against Tree Convolution, Tree Convolution against Tree MVA, and so on. Tests are run using all permutations of the sequential Convolution, sequential MVA, Tree Convolution and Tree MVA solvers to ensure that each solver gives the same results for a particular model. Since Tree Convolution and Tree MVA also have multi-threaded implementations these were also included in the testing permutations. For each solver pair, all of the tests in `GeneralTests` are run. This approach helped to build up a sense of consistency across all solvers related to this project.

In terms of the GUI, user acceptance testing was performed to check that the updated parts of the JMVA interface function correctly. If this project gets incorporated into the publicly available release of JMT, then further user testing will take place to iron out any minor bugs.

Chapter 6

Evaluation

Now that both the bounding solvers and tree algorithms have been integrated into JMVA, we can move onto evaluating the performance and usefulness of these methods for practical purposes by carrying out an objective analysis. We also attempt to prove the claims made in the related research papers and compare the new algorithms to existing solutions by carrying out an experimental campaign to test the algorithms under a variety of conditions. It should be noted that all of the experiments were carried out on a machine with 6GB RAM and an Intel Core i7 2.2GHz processor capable of running 8 threads in parallel (for the multi-threaded tree algorithms).

6.1 Geometric Bounds Evaluation

Of the three bounding methods implemented as part of this project, Geometric Bounding is the method we are most interested in analysing, since both Asymptotic Bounds and Balanced Jobs Bounds are much more well established methods. Nonetheless we use the implemented AB and BJB solvers as benchmark by which we can assess the improvements made by GBs. Figure 6.1 below shows some initial results for a single-class network with 5 stations using random service demands. The graphs show a comparison for 4 different whole system measures over a range of populations, between GBs, BJBs and MVA (which is included to show the exact results). As can be seen the Geometrics Bounds (cyan) are significantly closer to the exact results obtained via MVA (red) than BJBs (dark blue). Note that ABs are not included as they are less accurate than BJBs and graph scaling would make it harder to compare BJBs and GBs.

In order to quantify the improvements made by GBs, tests were run for multiple networks and the mean maximum error was measured for different population sizes. The results of this experiment are shown in Figure 6.2. Note that all the networks used to obtain the results contained a single class, since GBs can only operate on single class networks. As can be seen, GBs have a consistently lower maximum error and, as the left hand graph shows, GBs also converge faster as the population increases. GBs also have another practical advantage over BJBs, since they are capable of evaluating per station mean queue lengths and response times which is not possible with BJBs. Hence GBs offer a more accurate and robust alternative to BJBs for practical purposes.

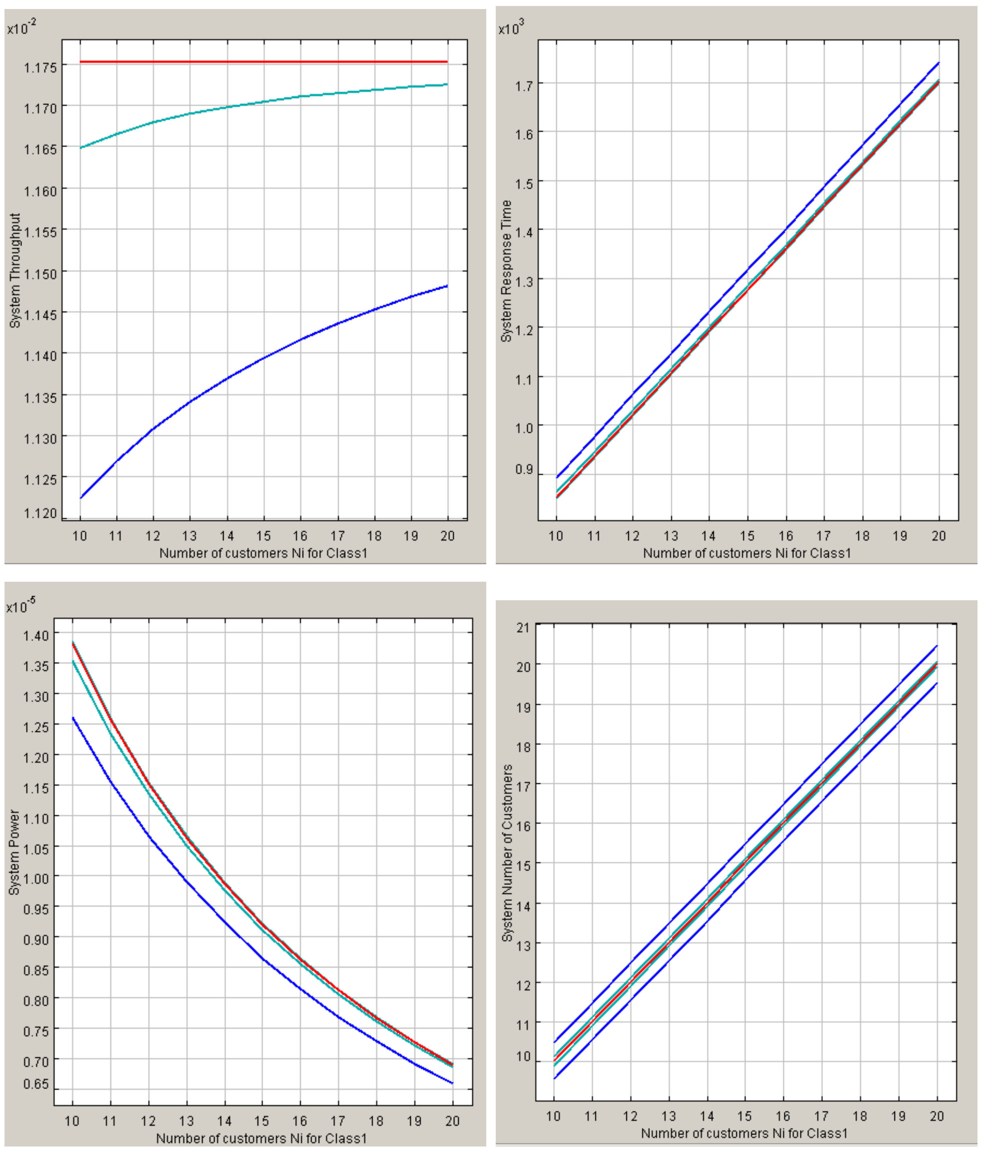


Figure 6.1: Comparisons between exact MVA (red), BJBs (dark blue), and GBs (cyan)

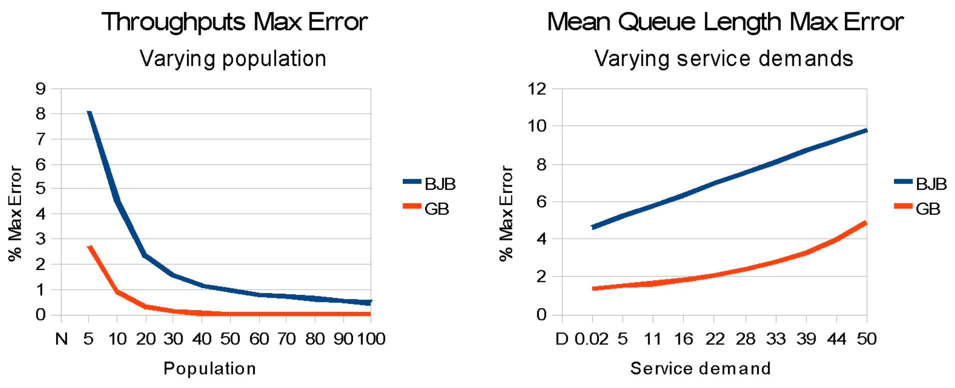


Figure 6.2: Maximum bounding error comparison between BJBs and GBs obtained from mean results of several networks with 5 stations. Left: shows throughput max bounding error, varying population. Right: shows max mean queue length error, varying demands for a single station.

In terms of runtime, Figure 6.3 shows a comparison between the three bounds solvers for a varying number of stations. All three of the bounding methods run in well under 1 millisecond and hence mean values had to be taken over a number of trials to increase the accuracy of the results. Note also that the times shown are for computing bounds for all performance measures used in JMVA, which requires a further iteration through all stations at the end of the calculation. As the graph shows, GBs have a consistently greater runtime than ABs and BJBs, as was expected. Additionally, the runtime of GBs increases more significantly with each added station. ABs and BJBs, however, only have to loop through the stations during their initial calculation phase in which they calculate various measures based on the station demands. On the other hand, GBs have to iterate through the stations for the actual calculation of throughput and queue length bounds, which helps to explain the shape of the graph. Our implementation of GBs also uses the BJB solver to get initial limits on throughput, which gives tighter GBs, but further adds to the runtime. For most practical circumstances the times involved are so small, that the extra microseconds required to compute GBs will be worth it, given that the resulting bounds will be significantly tighter.

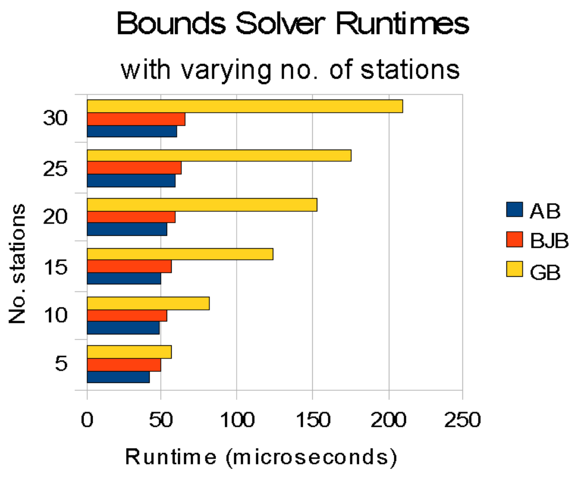


Figure 6.3: Runtimes comparison between bounding solvers.

6.2 Tree Algorithms Evaluation

6.2.1 Runtime comparison

We now move onto an experimental evaluation of both Tree MVA and Tree Convolution. A good place to start is with comparing the runtimes of the new algorithms against MVA, which is the fastest exact algorithm currently implemented in JMVA. Figure 6.4 shows the results over a range of populations for the example sparse network given in Tucci's paper [34]. The network contains 7 classes, 8 stations and service demands which ensure the network has the sparseness property. It should be noted that the x-axis is the population per class and not the total population of the whole network. As can be seen, for this sparse network, the results are very promising. The runtime for sequential MVA increases rapidly as the population rises, whereas Tree MVA and Tree Convolution show a much steadier increase. In fact when the population per class reached about 15, the network was no longer solvable by sequential MVA as the JVM ran out of memory. Sequential Convolution struggled to evaluate the model when the population per class was only 4, so we do do include it in the graph. We also see that Tree Convolution outperforms Tree MVA as the population gets larger. This was predicted by Tucci in [34].

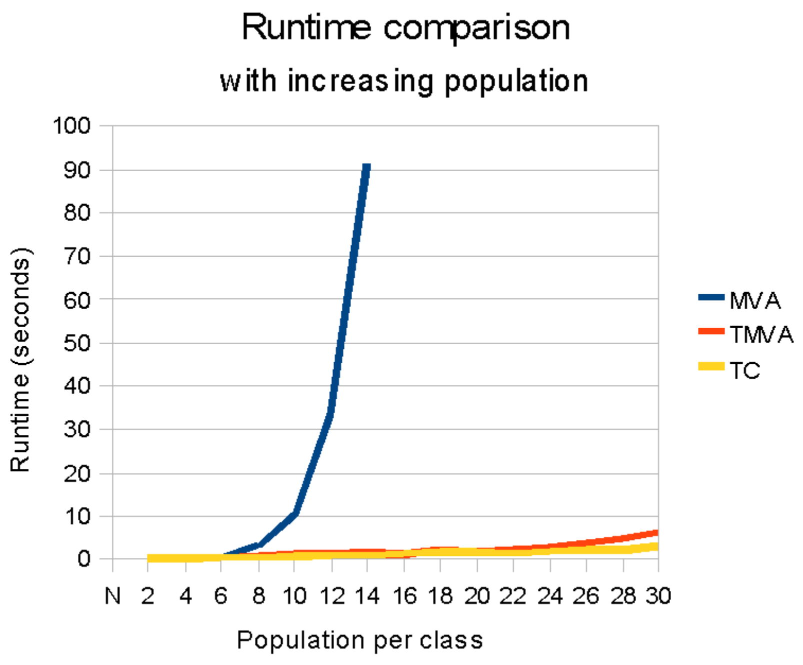


Figure 6.4: Runtimes comparison between exact solvers.

6.2.2 Memory usage comparison

VisualVM, a JVM monitoring tool, was used to determine the memory usage for the new algorithms over the same sparse network. Before each test, JMVA was restarted and the garbage collector was run to ensure that the used heap space was reset to base levels. The results were obtained by collecting heap dumps over a range of populations. Figure 6.5 shows the results of this experiment.

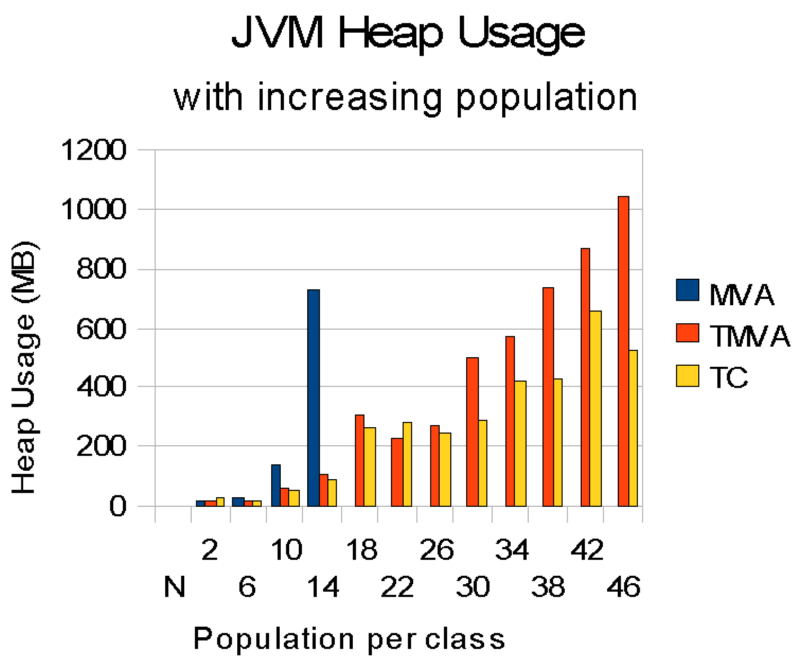


Figure 6.5: JVM heap usage comparison.

Following a similar pattern to the runtime graph, sequential MVA's heap usage increases rapidly as the population rises. As we noted before, this causes MVA to fail when the population reaches around 15. Tree Convolution and Tree MVA show a very steady increase in memory usage in comparison. We can see at $N = 14$ that both tree algorithms only use a fraction of the memory used by MVA. As the population increases further we begin to see that Tree MVA's memory usage increases faster than Tree Convolution. Nonetheless both algorithms manage to solve models with a population more than double the size solvable by sequential MVA.

6.2.3 Stress Testing

From the above analysis it is clear that both Tree Convolution and Tree MVA outperform their sequential counterparts in terms of runtime and memory usage. In order to get a better picture of the performance of the tree algorithms we now perform some stress tests by varying the population, stations and classes in the Tucci model over extreme ranges. Figure 6.6 below shows the runtimes for TMVA and TC as the total population in the network was increased. Figure 6.7 shows the runtimes as the number of stations increases (note the logarithmic scale). Figure 6.8 shows the runtimes as the number of classes were increased (an effort was made to keep the sparseness of the model constant). For each experiment the base model had 8 stations, 7 classes and a per class population of 2.

Population	TMVA	TC
70	1.36s	0.5s
140	1.78s	1.54s
210	6.36s	2.99s
280	22.12s	8.02s
350	88.15s	20.73s
420	Fail	53.6s
490	N/A	102.05s
560	N/A	Fail

Figure 6.6: Increasing the population to failure.

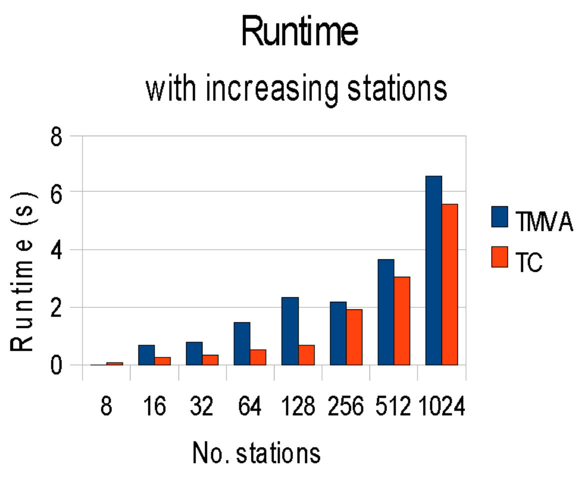


Figure 6.7: Increasing stations (runtime not including tree planting time).

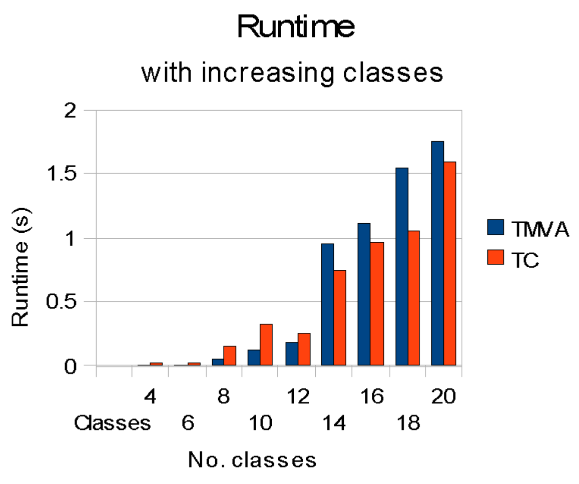


Figure 6.8: Increasing classes.

From Figure 6.6 we can see the Tree Convolution is capable of solving models with larger populations than Tree MVA. This can be explained by the heap usage statistics shown in Figure 6.5, since with higher populations TMVA starts to consume more memory. This may partially be due to the fact that more time was spent optimizing Tree Convolution than Tree MVA (as TMVA was not in the original plan for this project).

From Figure 6.7 we can observe that both algorithms perform well with large numbers of stations in the network, when the sparsity of the network is maintained. Again we see that Tree Convolution performs slightly better overall. In order to increase the stations while maintaining a similar sparsity, the station demands were simply copied from the original network each time new stations were created. Note that the tree planting time was ignored since with large numbers of stations tree planting took a lot longer.

From Figure 6.8 we can see that increasing the number of classes even by a relatively small amount affects the runtime significantly. Note that this graph is more prone to inaccuracy than the others, since increasing the number of classes while maintaining the same network sparsity is difficult. Therefore, this graph serves only as an indicator of the general trends seen and should not be taken as a definitive result. If this experiment was run with a different network, the increase in runtime may be much smaller.

Overall, we can see that the Tree Convolution implementation outperforms Tree MVA for this particular network. Both algorithms however allow the exact solution of much larger models than sequential MVA.

6.2.4 Non-sparse networks

So far we have only considered the performance of Tree Convolution and Tree MVA over sparse networks, since that is their intended usage. However, we also consider their use over non-sparse networks. For this experiment a random non-sparse network (all service demands above 0, meaning that all classes visit all stations) was used with 2 classes and 4 stations. As Figure 6.9 below shows, sequential MVA performs much better as the population increases. It is unfortunate that the vast time and memory savings we have shown for the tree algorithms do not extend to non-sparse networks. However, this was the expected result since in non-sparse networks there are no partially covered classes to take advantage of. Additionally, the cost of merging stations using the tree algorithm equations, when all classes visit all stations, is greater than the cost of applying sequential MVA.

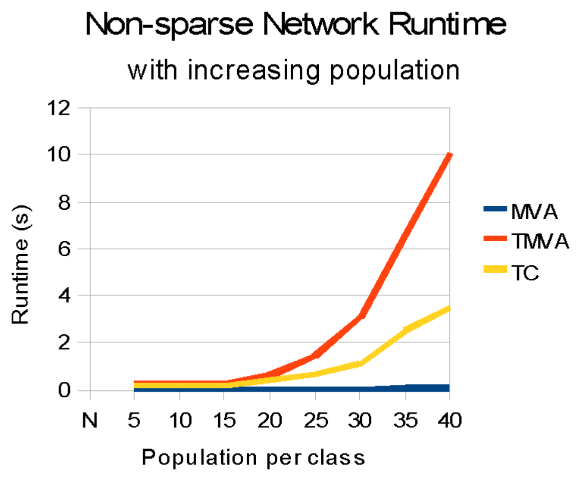


Figure 6.9: Runtime for non-sparse network as population increases.

6.2.5 Single vs multi threading

All of the above tests were performed using the multi-threaded version of Tree MVA and Tree Convolution. We now run some experiments to quantify the difference that multi-threading makes to the runtime of the algorithms. These tests were run on a quad-core machine capable of running 8 threads in parallel. Figure 6.10 below shows the results of running both the single and multi-threaded implementation of Tree Convolution and Tree MVA over varying populations and number of stations. From the graphs we see that the influence of multi-threading only becomes significant when the network becomes larger. In particular, the effect is more pronounced as the number of stations increases, as the multi-threaded implementation is able to utilize multiple cores to do subnetwork calculations in parallel. Of course the results will vary depending on the machine on which the algorithms are run.

Another interesting point is that single-threaded TMVA outperforms single-threaded TC as the number of stations increases. However, multi-threaded TC performs the best of all. This is due to the fact that there is more opportunity to exploit parallelism in TC during the computation of performance measures.

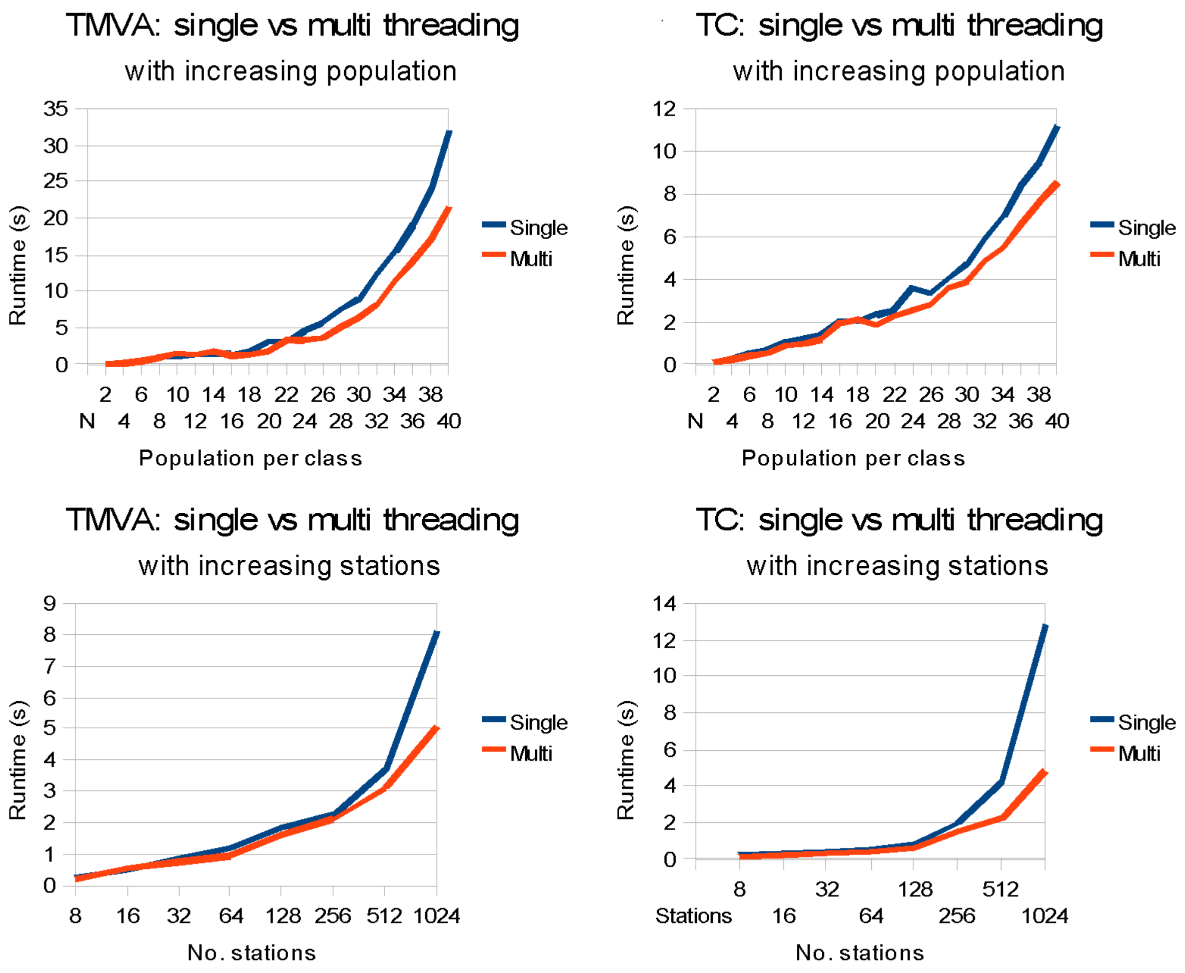


Figure 6.10: Comparisons of single and multi threaded implementation runtimes over varying populations and number of stations.

6.2.6 Tree Convolution Breakdown

One significant point to highlight is that for Tree Convolution, the computation of mean queue lengths often becomes the bottleneck in the algorithm as the population gets large. Figure 6.11 shows this trend. As the population increases, the time to compute performance measures increases much faster than the time to compute the normalisation constant. This is mainly due to the fact that mean queue lengths have to be computed for each station and class, requiring lots of additional traversals and convolution operations. This becomes much more costly as the population increases.

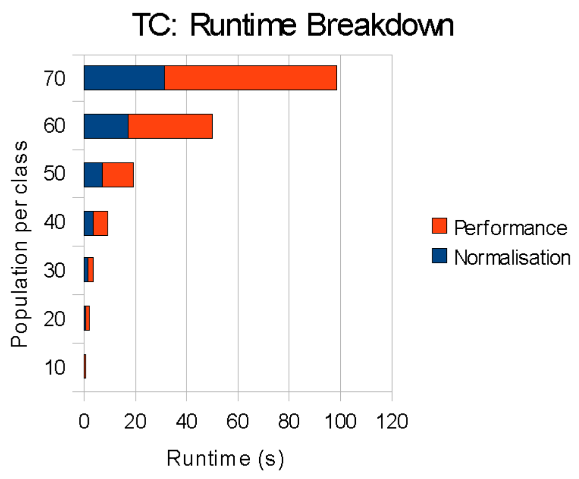


Figure 6.11: Runtime breakdowns for Tree Convolution as population increases. Blue represents time to compute normalisation constant, orange represents time to compute performance measures.

6.2.7 Key Observations

The following observations were made while performing experiments on the tree algorithms:

- Both tree algorithms perform much better than existing sequential methods for sparse networks, both in terms of runtime and memory usage.
- Tree Convolution outperforms Tree MVA in most cases, however this may be due to better utilisation of parallel processing in the Tree Convolution implementation.
- The tree algorithms perform worse than sequential techniques for non-sparse networks, as expected.
- The runtime and memory usage depend on the tree planting procedure used (in order to achieve the graphs above both the heuristic and simple planting approaches described in section 5.3.3.1 had to be used).
- The performance of the tree algorithms is highly dependent on the settings configured in the `Config` class. In fact for various configurations it has been seen that the tree planting procedure used can mean the difference between the algorithm taking milliseconds to run and it taking minutes to run. So while the tree algorithms show great promise, there is also a large variability in their performance and they must be used with an understanding of the underlying principles in order to yield good performance. For this reason some suggestions on how to make these algorithms more 'user friendly' are outlined in section 7.1 and could form part of a future project.

6.2.8 Real World Example

In order to show the potential use of the tree algorithms in a real world scenario, we show how they can be used to analyse the network shown in Figure 6.12. The model contains 4 classes representing HTTP, authentication, transaction and database requests. The request router forwards requests to the relevant server depending on the request type. As the diagram shows, HTTP requests are forwarded to the HTTP content manager, authentication and transaction requests are forwarded to the security manager, and database queries are forwarded to the database manager. The HTTP content manager and DB manager decide how to handle a request and then offload the work to one of their child servers. The security manager performs some initial security checks before further splitting the requests into user authentication and transaction requests and forwarding them the the relevant servers. For the purposes of evaluation we assume that this network runs on a fast local area network (LAN) and that there are therefore no communication delays. The child HTTP, Auth, Transaction and DB servers can work in parallel and their parent nodes distribute the work evenly between them. In order to keep the network sparse we also make the reasonable assumption that the request router takes a negligible amount of time to forward requests and therefore we do not include it in the model.

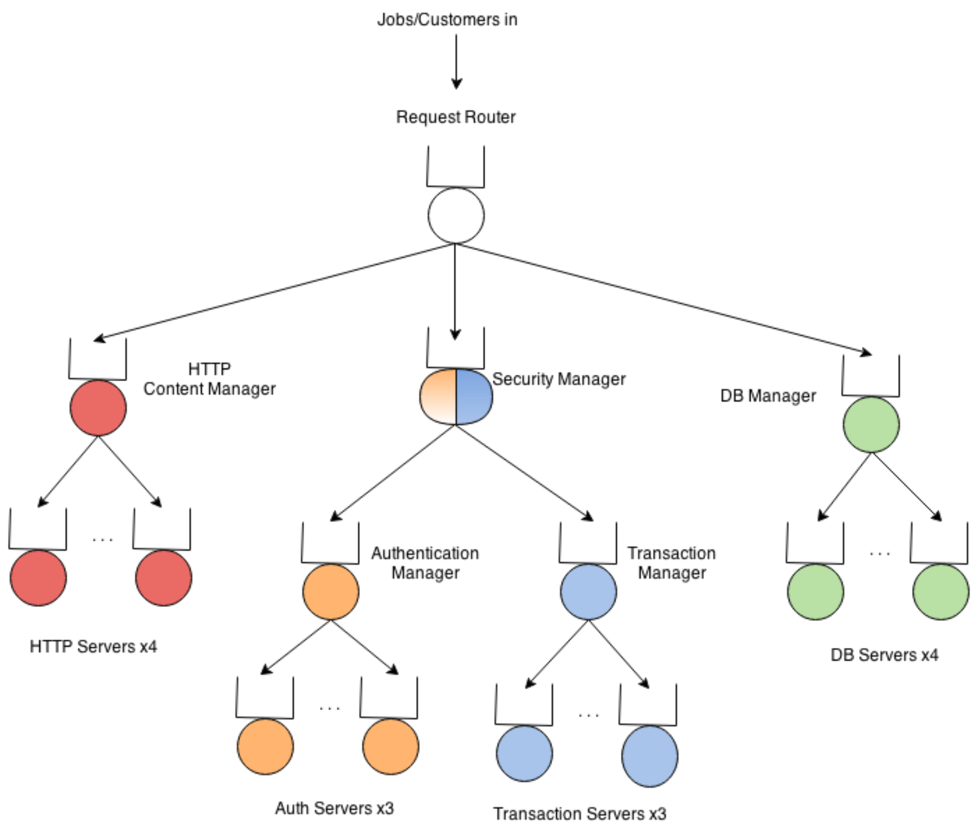


Figure 6.12: Real-world hierarchical network (N.B. network is closed, but arrows back to start are not shown for simplicity). Red nodes handle HTTP requests, orange nodes handle authentication requests, blue nodes handle transaction requests, and green nodes handle database queries.

The specific service demands used for this example are given in Appendix A. In general we assume that security related requests take slightly longer than HTTP and database queries. We also assume that for each array of child servers there is one server which performs slightly worse than the others to introduce some non-uniformity to the system.

This network was solved using MVA, Tree MVA and Tree Convolution. The ratio of the requests was assumed to be 4:2:1:1 with HTTP requests making up the majority of the traffic, database queries taking up half as much, and authentication and transactions taking up half as much again. Figure 6.13 below shows the performance of the algorithms over various loads. It can clearly be seen that Tree MVA and Tree Convolution significantly outperform MVA. With the largest load tested, MVA would have taken well over half an hour to run (if there was sufficient memory), whereas Tree MVA and Tree Convolution were able to solve the network exactly in under 10 seconds.

Load	**MVA**	**TMVA**	**TC**
(100, 50, 25, 25)	2.65s	0.08s	0.05s
(200, 100, 50, 50)	34.56s	0.25s	0.15s
(300, 150, 75, 75)	3mins 1s	0.45s	0.3s
(400, 200, 100, 100)	10mins 53s	0.8s	0.45s
(800, 400, 200, 200)	N/A	8.73s	3.54s

Figure 6.13: Real-world example, runtimes for various loads.

Since the sparse network presented here is a typical example of the networks used in many industry settings; making use of layers, hierarchical structures and subdivision of labour. It is reasonable to suggest that the newly implemented tree algorithms may have useful applications in industry, particularly when the results of an evaluation need to be highly accurate.

6.2.9 Comparison with MoM, CoMoM and RECAL

It was found during the later stages of this project that the MoM, CoMoM and RECAL algorithms implemented in previous projects do not give accurate results for sparse networks. Since these three algorithms are all rather complex, there was not enough time to look into what was causing this. Unfortunately this meant that it was not possible to carry out a fair comparison with the new tree algorithms. However, it is expected that Tree Convolution and Tree MVA will outperform these algorithms for sparse networks, though for non-sparse networks the previously implemented algorithms will be more efficient. Of course the other algorithms will need to be fixed before this claim can be justified. For the time being JMVA was patched so that the algorithms which give inaccurate results for sparse networks now show an error message if the users attempts to solve a sparse network using them. Tree Convolution and Tree MVA are therefore the best methods currently available within JMVA for solving sparse networks.

6.3 Qualitative Aspects

So far we have analysed the performance of the newly implemented algorithms. We now briefly discuss how easy these algorithms are to use and how effectively they were integrated into the existing software. In order to ensure these aspects of the project were of a high standard, extensive user acceptance testing was performed by myself and others. The main changes made to the GUI were through the introduction of bounding algorithms, since this required a new way to represent results both in tabular and graphical format. In view of this, most of the user testing was focused on these areas in order to ensure GUI changes were consistent with the rest of the interface. A particular focus for the bounding algorithms was on their ease of use and clear presentation of results. The tree algorithms were more simple to integrate and test since they reused most of the GUI components from existing algorithms. While implementing each part of the project, manual testing was used extensively, in addition to automated testing, to provide feedback on the robustness and usability of the solution. The new algorithms were also tested with invalid inputs to ensure their correct behaviour in different scenarios. During the later stages of the project a meeting was held with Giuseppe Serazzi, the JMT project coordinator from Politecnico di Milano in Italy. This meeting provided some interesting insights into other areas of JMT which are currently being developed and also provided some useful feedback concerning the consistency of the user interface.

6.4 Strengths and Weaknesses

The strengths of the work completed for this project include:

- The implementation of Geometric Bounding techniques for single-class networks offers significantly tighter bounds than the current best-established techniques, such as Balanced Job Bounds.
- For the evaluation of sparse networks, both Tree MVA and Tree Convolution drastically outperform existing exact methods, both in terms of runtime and memory used.
- Tree MVA and Tree Convolution were extended to use multiple cores, allowing for even faster analysis of sparse networks.
- Suitable tree planting techniques and complexity evaluation tools were implemented for use with the tree algorithms.
- Integration of bounding and tree algorithms into existing software, including the ability to graphically view bounding results in JMVA. Additionally, the tree algorithms were integrated into the existing commandline tool in the JCoMoM package.
- The software for both bounding solvers and tree solvers was designed so at to be understandable, maintainable and easily extensible. In particular, the tree algorithms package was refactored so that the main components, such as tree traversal techniques and tree planting, can be easily reused in the future.
- Hopefully the positive results presented in this report will encourage further research into these areas, particularly the application of tree algorithms to a wider class of networks and possibly also as the basis of new approximate evaluation techniques.

Some of the limitations associated with this project are:

- Tree MVA is currently less memory efficient than Tree Convolution. With more time to optimize this could have been improved.
- The most significant limitation of the tree algorithms is that they only work well for sparse networks. However, this was expected to be the case and is due to the theory behind the algorithms, not the implementation.
- The tree algorithms, though showing very promising performance for sparse networks, require a fair amount of configuration and must be used with intelligence and care. Future work could include methods for making these algorithms more user-friendly, for example by self-configuring based on the network under evaluation. The beginnings of this have already been started with the complexity evaluation and tree planting stage (which considers multiple alternatives), however a more comprehensive solution would be desirable.

Chapter 7

Conclusion

In this report we have presented the design, implementation and evaluation of several new queueing network analysis algorithms. In the first part of the project we implemented some existing bounding methods, including Asymptotic Bounds and Balanced Job Bounds, and then went on to implement a relatively modern approach, Geometric Bounds. As the evaluation showed, the Geometric Bounds implementation was able to provide significantly tighter bounds than the other methods, though incurring a slightly higher computational cost. The integration of these techniques within JMVA will enable very fast analysis to occur during the initial stages of network analysis when exact results are not required. Hopefully they will be particularly useful for the fast identification of network bottlenecks.

We then moved on to look at Tree Convolution and Tree MVA for the efficient solution of sparse networks. Both of these algorithms took a significant investment in time to understand and implement correctly due to their complexity. However, as we have demonstrated both the Tree Convolution and Tree MVA implementations dramatically outperform existing sequential techniques, when run on sparse networks. The integration of these techniques into JMVA has greatly increased the number of networks for which exact results can be obtained in a reasonable amount of time. This will also provide a way of measuring the accuracy of the approximate MVA methods, implemented in a previous project, over larger networks without having to resort to simulation techniques.

While the results of this project have been mostly positive, the implementation of the tree algorithms is by no means perfect. A large percentage of the development time for both algorithms was spent getting the algorithms into a working state, since there were many low-level details which were not apparent from the relevant research papers. As such, relatively little time was spent on optimizing the algorithms. Though the results are still impressive in comparison to existing sequential methods, both algorithms have the potential to run even faster given more work on optimization and possibly a re-implementation in a language with more control over memory allocation such as C/C++.

As well as being the first publicly available implementation of these algorithms, the main original contribution of this project is in the development of tools capable of analysing the complexity of a network tree during the tree planting stage. These methods allow several planted trees to be compared using different methods, including calculations of expected time and space complexity, comparison of network sparsity, and analysis of the partially covered classes in network trees. These techniques provide good approximations in most cases, however there is potential for further work on the preprocessor stage of both tree algorithms, since this stage directly effects how fast they are capable of running.

7.1 Future Work

On the whole, the work done as part of this project has provided positive results, however there are still areas which have potential for further work:

- The most immediate area which could benefit from more work is the preprocessor stage of the tree algorithms. Since the tree planting heuristics used in the initial stages of the algorithms have a large impact on performance, a more sophisticated preprocessor tool would be desirable. Such a tool would be able to perform an in-depth analysis of a network taking into account factors such as expected complexity, machine capabilities, JVM heap usage, network structure and scale, class routing information, and potential for parallel computation. After analysis the tool would then be able to choose an appropriate planting heuristic and perhaps plant the tree in a dynamic fashion, taking into account some of the information acquired during analysis. Some work has already been done with these ideas in mind, however the concepts need to be unified into a coherent, user-adjustable tool.

- In terms of the bounding solvers, a future extension could be the implementation of Proportional Bounds (PBs) as discussed in [6] and [22]. PBs are theoretically more accurate than ABs and BJBs but require a small increase in computational cost. They were not included in the original plan for this project but it would be interesting to compare the accuracy and efficiency of PBs with Geometric Bounds.

- While the overall architecture of JMT is based on solid design principles, over time the code base has, in places, become messy, inconsistent and often hard to understand. This is probably due to the open nature of the JMT project and the lack of coding standards. Unfortunately, there was not much time to address these issues during this project. In fact an entire project could probably be based on applying software engineering practices to the JMT code base. This will become increasingly important in the future, since with the constant addition of new algorithms, the current code base will eventually become unmaintainable unless an effort is made to apply coding standards and sound design principles.

- Although the theory has not yet been developed, a future project could look at applying the general tree algorithm approach to approximate methods, such as the AMVA algorithms implemented in [13]. Potentially this could further increase the scope of models that JMVA is capable of evaluating. Additionally, since these algorithms would be based on many of the same principles as Tree Convolution and Tree MVA, many of the components implemented in this project, such as tree planting, tree representation and tree traversal, could be easily reused.

Bibliography

[1] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers, J. ACM, 22(2):248-260, 1975.

[2] M. Bennani and D.A. Menasc. Resource Allocation for Autonomic Data Centers Using Analytic Performance Models, Proc. Second IEEE Intl Conf. Autonomic Computing, 2005.

[3] J.W. Bradshaw. An Efficient Implementation of the Class-Oriented Method of Moments for Computer Performance Analysis. MSc thesis, Imperial College London, September 2012.

[4] J. Buzen, Computational algorithms for closed queueing networks with exponential servers, CACM, vol. 16, no. 9, pp. 527-531, September 1973.

[5] P.J. Denning and J.P. Buzen. The Operational Analysis of Queueing Network Models, ACM Computing Surveys, 10(3):225-261, 1978.

[6] G. Casale, R.R. Muntz, G. Serazzi. Geometric Bounds: A Noniterative Analysis Technique for Closed Queueing Networks, IEEE Transactions on Computers, 57(6):780-794, June 2008.

[7] G. Casale. CoMoM: Efficient class-oriented evaluation of multiclass performance models, IEEE Transactions on Software Engineering, 35(2):162-177, 2009.

[8] G. Casale. Exact analysis of performance models by the Method of Moments. Performance Evaluation 68 (2011), pp. 487-506, 2009. doi:10.1016/j.peva.2010.12.009.

[9] G. Serazzi, M. Bertoli, G. Casale. Java Modelling Tools: User Manual, October 2013.

[10] K.M. Chandy, U. Herzog and L.S. Woo. Parametric analysis of queueing networks, IBM J. of Research and Development, 19(1):36-42, January 1975.

[11] K.M. Chandy and D. Neuse. Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems, Comm. ACM, 25(2):126-134, 1982.

[12] W.C. Cheng, R.R. Muntz. Bounding Errors Introduced by Clustering of Customers in Closed Product-Form Queueing Networks, J. ACM, 43(4):641-669, 1996.

[13] A. Chugh. Algorithms for System Performance Analysis. MEng thesis, Imperial College London, June 2012.

[14] A.E. Conway, N.D. Georganas, RECAL - a new efficient algorithm for exact analysis of multiple-chain closed queueing networks, Journal of the ACM, 33(4):768-791, October 1986.

[15] C.A. Floudas. Nonlinear and Mixed Integer Optimization. Journal of Global Optimization, 12(1):108-110, 1995.

[16] W.J. Gordon, G.F. Newell. Closed queuing systems with exponential servers. Operations Research, 15(2):254-265, April 1967.

[17] K.P. Hoyme et al. A Tree-Structured Mean Value Analysis Algorithm, ACM Transactions on Computer Systems, 4(2):176-185, May 1986.

[18] *Java Modelling Tools*, http://jmt.sourceforge.net/

[19] L. Kleinrock. Power and deterministic rules of thumb for probabilistic problems in computer communications. Proc. International Conference on Communication, pp. 43.1.1-43.1.10, June 1979.

[20] J. Kriz. Throughput Bounds for Closed Queueing Networks, Performance Evaluation, 4(1):1-10, 1984.

[21] S.S. Lam, Y.L. Lien. A tree convolution algorithm for the solution of queueing networks, CACM, 26(3):203-215, March 1983.

[22] C.H. Hsieh and S. Lam. Two Classes of Performance Bounds for Closed Queueing Networks, Performance Evaluation, 7(1):3-30, 1987.

[23] E.D. Lazowska, J. Zahorjan, G.S. Graham, K.C. Sevcik. Quantitative System Performance: Computer System Analysis Using Queueing Network Models, Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1984, chapters 1-7.

[24] R.R. Muntz and J.W. Wong. Asymptotic Properties of Closed Queueing Network Models, Proc. Eighth Ann. Princeton Conf. Information Sciences and Systems, pp. 348-352, 1974.

[25] A. Phansalkar et al. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites, Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS05), 2005.

[26] M. Reiser and H. Kobayashi. Queueing networks with multiple closed chains: Theory and computational algorithms, IBM J. of Research and Development, 19(3):283-294, May 1975.

[27] M. Reiser and H. Kobayashi. On the convolution algorithm for separable queueing networks, Proc. International Symposium of Computer Performance, pp. 109-117, 1976.

[28] M. Reiser and S.S. Lavenberg. Mean value analysis of closed multichain queueing networks, Journal of the ACM, 27(2):313-322, April 1980.

[29] M. Reiser. Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks, Performance Evaluation 1, pp. 7-18, 1981.

[30] C.H. Sauer. Computational algorithms for state-dependent queueing networks, ACM TOCS, 1(1):67-92, February 1983.

[31] P.J. Schweitzer. Approximate analysis of multi-class closed networks of queues. Proceedings of International Conference on Stochastic Control and Optimization, pp. 25-29, 1979.

[32] TMurget Technologies. Perceived Performance: Tuning a system for what really matters, White paper, TMurget Technologies, September 2003.

[33] D. Simchi-Levi, M.A. Trick. Introduction to "Little's Law as Viewed on Its 50th Anniversary". Operations Research, 59(3):536-549, May 2011, doi:10.1287/opre.1110.0941.

[34] S. Tucci. The Tree MVA Algorithm, Journal of Performance Evaluation, 5(3):187-196, August 1985.

[35] E. Varki and L.W. Dowdy. Analysis of Fork-Join Queueing Networks, Proc. ACM SIGMETRICS 96, pp. 232-241, 1996.

[36] K. Whisnant, Z. Kalbarczyk, and R.K. Iyer. A System Model for Dynamically Reconfigurable Software, IBM Systems J., 42(1):45-59, January 2003.

[37] J. Zahorjan, K.C. Sevcik, D.L. Eager, and B. Galler. Balanced Job Bound Analysis of Queueing Networks, Comm. ACM, 25(2):134-141, February 1982.

Appendix A

Real-world Example Service Demands

The service demands that were used to obtain the results for the real-world example network presented in section 6.2.8 are as follows:

*	AuthRequest	HttpRequest	DBQuery	TransactionRequest
AuthServer	1	0	0	0
HTTPManager	0	0.5	0	0
DBManager	0	0	0.5	0
TransactionManager	0	0	0	1
DBServer1	0	0	0.5	0
DBServer2	0	0	0.5	0
DBServer3	0	0	0.5	0
DBServer4	0	0	1	0
AuthServer1	2	0	0	0
AuthServer2	2	0	0	0
AuthServer3	3	0	0	0
HTTPServer1	0	1	0	0
HTTPServer2	0	1	0	0
HTTPServer3	0	1	0	0
HTTPServer4	0	2	0	0
TransactionServer1	0	0	0	1
TransactionServer2	0	0	0	1
TransactionServer3	0	0	0	2

Figure A.1: Real-world example service demands.