GENERALISED MARKOVIAN ANALYSIS OF TIMED TRANSITION SYSTEMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF SCIENCE

AT THE UNIVERSITY OF CAPE TOWN

IN FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By
William J. Knottenbelt
June 1996

Supervised by
Prof P.S. Kritzinger

Abstract

This dissertation concerns analytical methods for assessing the performance of concurrent systems. More specifically, it focuses on the efficient generation and solution of large Markov chains which are derived from models of unrestricted timed transition systems. Timed transition systems may be described using several high-level formalisms, including Generalised Stochastic Petri nets, queueing networks and Queueing Petri nets. A system modelled with one of these formalisms may be mapped onto a Markov chain through a process known as state space generation. The Markov chain thus generated can then be solved for its steady-state distribution by numerically determining the solution to a large set of sparse linear equations known as the steady-state equations.

Existing techniques of state space generation are surveyed and a new space-saving probabilistic dynamic state management technique is proposed and analysed in terms of its reliability and space complexity. State space reduction techniques involving on-the-fly elimination of vanishing states are also considered. Linear equation solvers suitable for solving large sparse sets of linear equations are surveyed, including direct methods, classical iterative methods, Krylov Subspace techniques and decomposition-based techniques. Emphasis is placed on Krylov subspace techniques and the Aggregation-Isolation technique, which is a recent decomposition-based algorithm applicable to solving general Markov chains.

Since Markov chains derived from real life models may have very large state spaces, it is desirable to automate the performance analysis sequence. Consequently, the new state management technique and several linear equation solvers have been implemented in the Markov chain analyser DNAmaca. DNAmaca accepts a high-level model description of a timed transition system, generates the state space, derives and solves the steady-state equations and produces performance statistics. DNAmaca is described in detail and examples of timed transition systems which have been analysed with DNAmaca are presented.

Acknowledgements

I would like to express my thanks and appreciation to:

- My supervisor, Prof Pieter S. Kritzinger, for his guidance and skilful management throughout the past 18 months.
- My family for their love, support and encouragement.
- My fellow DNA lab colleagues and my fellow Masters students in the Department of Computer Science, for their companionship and encouragement.
- Michael Sczittnick, the author of USENUM, for the use of his Markov Chain analyser and his assistance with USENUM model specifications.
- DNA lab member Heinz Kabutz, for extensively beta-testing DNAmaca on several models produced during the course of his masters thesis.
- Peter Kemper, from the University of Dortmund, Germany, for helping me to verify the results produced by DNAmaca by running USENUM models with large state spaces using the computer resources at the University of Dortmund.
- Farooq Khan and Prof Djamal Zeghlache, both from RST department of L'Institut National des Télècommunications, Evry, France, for beta-testing DNAmaca and for suggesting the teletraffic switch application.
- Abderezak Touzene, from King Saud University, Saudi Arabia, for his advice and assistance while implementing his Aggregation-Isolation algorithm.
- DNA lab staff member Andrew Hutchison, for his willing assistance with modelling test cases using the MicroSnap queueing network analyser.

- Roya Ulrich, currently at the International Computer Science Institute, Berkeley, California, for her help with USENUM and performance analysis concepts.
- The Electrical Engineering department at the University of Cape Town for the use of their 4-processor SPARC 20 during the results-collection phase.
- Sandi Donno and Aleks Strez, our local system adminstrators, for keeping my computing environment running smoothly.
- The South African Foundation for Research and Development (FRD) for funding this research.

Contents

Abstract **ii**

Acknowledgements **iii**

1 Introduction **1**

1.1 Motivation and Objectives 1

1.2 Other Work 3

1.2.1 USENUM 3

1.2.2 MARCA 5

1.3 Dissertation Outline 7

2 Background theory **9**

2.1 Introduction 9

2.2 Modelling Formalisms 10

2.2.1 Generalised Stochastic Petri Nets (GSPNs) 10

2.2.2 Queueing Networks 14

2.2.3 Queueing Petri nets (QPNs) 17

2.3 Markov Theory 18

2.3.1 Stochastic Processes 18

2.3.2 Markov Processes 19

3 State Space Exploration Techniques **24**

3.1 Introduction 24

3.2 Traditional state space exploration techniques 25

3.2.1 Exhaustive dynamic techniques 26

3.2.2 Probabilistic static techniques 27

3.3 A new probabilistic dynamic technique 31

3.3.1 Reliability analysis 33

3.3.2 Approximation 35

3.3.3 Space complexity 36

3.4 Vanishing state elimination 40

4 Steady State Solution Techniques **47**

4.1 Introduction 47

4.2 Direct methods 50

4.2.1 Gaussian Elimination 50

4.2.2 *LU* Decomposition 52

4.2.3 Grassman's Algorithm 54

4.3 Classical iterative methods 55

4.3.1 Jacobi's Method 56

4.3.2 Gauss-Seidel 57

4.3.3 Successive Overrelaxation (SOR) 57

4.4 Krylov subspace techniques 58

4.4.1 Principles of Krylov Subspace Techniques 60

4.4.2 Basis construction algorithms 63

4.4.3 Generalised CG Techniques 66

4.4.4 Conjugate Krylov Subspace Techniques 77

4.5 Decomposition-based techniques 88

4.5.1 Principles of Decomposition-based Techniques 88

4.5.2 Aggregation-Isolation algorithm 91

5 Interface Language Specification **97**

5.1 Introduction 97

5.2 Language elements 98

5.2.1 Model Description 98

5.2.2 Generation Control 102

5.2.3 Solution Control 103

5.2.4 Performance Measures/Results 104

5.2.5 Output options 106

6 The DNAmaca Performance Analyser **107**

6.1 Introduction 107

6.2 DNAmaca Components 107

6.2.1 The Parser 109

6.2.2 The State Space Generator 110

6.2.3 The Functional Analyser 114

6.2.4 The Steady State Solver 115

6.2.5 The Performance Analyser 118

7 Example Timed Transition Systems and Solutions **121**

7.1 Introduction 121

7.2 Multimedia teletraffic switch 121

7.3 Interactive computer system 125

7.4 TFTP telecommunications protocol 126

8 Conclusion **131**

8.1 Summary 131

8.2 Future work 132

A DNAmaca model files **134**

A.1 Multimedia teletraffic switch model 134

A.2 Interactive computer system model 136

A.3 TFTP communications protocol 139

Bibliography **149**

Chapter 1

Introduction

1.1 Motivation and Objectives

When developing a complex concurrent system, such as a telecommunications protocol or a railway signalling control system, it is important to prevent costly redesigns and serious errors by assessing the correctness and performance of the system before implementation. This can be achieved by constructing and analysing an abstraction or model which captures the essential aspects of the system's behaviour. In this dissertation we will concern ourselves with efficient ways of establishing the *performance* of concurrent systems starting from such a system *model*.

There are two general approaches to obtaining performance statistics for a system: *analytical methods* and *simulation*. Analytical methods make use of formal, abstract models from which exact results can be obtained by solving a set of equations derived from the model. Simulation, on the other hand, can be used to model systems at arbitrary levels of detail, producing inexact results bounded by confidence intervals. The accuracy of the results may be improved by extending simulation execution time and/or by making the model more detailed. However, relative to analytical models, there is a high cost and effort involved in constructing accurate simulation models and the length of execution time required to produce reliable results can be extremely long. For example, in simulations of modern high-speed slotted networks, where a very large number of events occur in short periods of time, the simulation of 8000 framing periods, corresponding to an elapsed system time of just 1 second, requires almost four hours of processing time on a SPARC5 workstation [Ulr95, pg. 105]. An analytical model of the same system requires just 20 seconds to solve.

This dissertation focuses on a widely-used analytical technique known as *Markov chain modelling*. Markov chains model the low-level stochastic behaviour of a system by describing what possible states the system may enter and how the system moves from one state to another in time. Markov chains are limited to describing such systems which have discrete states and which satisfy the property that the future behaviour of the system only depends on the current state. Despite these limitations, Markov chains are a flexible representation capable of modelling the phenomena found in complex concurrent systems, including blocking, synchronisation, preemption, state-dependent routing and complex traffic arrival models.

Markovian models of real-life systems may involve many hundreds of thousands, or millions, of states. It is thus infeasible to manually specify each of the states and the transitions between states. Several high-level *formalisms* from which Markov chains can be derived have thus come about. Examples of such formalisms are Generalised Stochastic Petri nets [AMCB84], queueing networks and Queueing Petri nets [BB89]. Systems specified with one of these high-level formalisms are known as *timed transition systems*. Performance statistics for these systems can be obtained by mapping the states of the system onto a Markov chain and then solving a set of linear equations to determine the chain's steady-state distribution.

Since it is possible to derive a Markov chain model from any formalism describing a timed transition system, it thus makes sense to have a tool which implements the complete performance analysis sequence on general timed transition systems. That is, the tool should accept a high-level model description, derive a Markov chain through a state space exploration, and then solve the chain to produce performance statistics.

A major problem which immediately presents itself in the design of such a tool is the state space explosion problem. One approach to this problem is to restrict the structure of system models, for example, by imposing a hierarchy. This allows for the application of efficient analysis techniques which exploit the restricted structure. We will not adopt this approach, however, since we seek techniques which can be used to derive performance statistics for *general unrestricted* timed transition systems. In particular, we will investigate efficient techniques for two problems: generating large state spaces and solving large sets of linear equations.

The generality of our approach also poses the problem of developing an interface language which is general enough to handle the description of any timed transition system. We propose such a language.

There are two general Markov chain solvers that the author is aware of: USENUM [Scz87], developed around 1987 at the University of Dortmund, and MARCA [Ste91] developed at the University of North Carolina. We have developed our own analyser called DNAmaca, which makes two main contributions:

- DNAmaca uses a new probabilistic dynamic state management technique. This results in considerable memory savings over conventional exhaustive or static techniques.
- DNAmaca includes implementions of four classes of applicable sparse linear equation solvers, namely direct methods, classical iterative techniques, Krylov Subspace techniques and decomposition-based techniques.

With DNAmaca we are able to solve general, unrestricted Markov chains of as many as 500 000 tangible states with only 64 Mb RAM.

1.2 Other Work

In this section, we review the Markov chain analysers USENUM and MARCA. We describe their state and transition representations, interface languages, state state exploration algorithms and steady-state solution techniques. We also briefly review some application packages which make use of the analysers as their underlying solution engines. The reader is referred to chapters 3 and 4 for explanations of the technical terms and abbreviations used in this section.

1.2.1 USENUM

USENUM [Scz87] is a Markov chain analyser developed at the University of Dortmund by Michael Sczittnick in 1987. The analyser was originally implemented on a BS2000 system using SIMULA, then rewritten in C for use on UNIX systems. With USENUM, it is possible to analyse models with up to about 100 000 states (both vanishing and tangible) on a machine with 64 Mb RAM.

State and Transition representation: In USENUM, a state is represented as an integer-valued row state descriptor vector. Transitions from the current state to the next state are specified by:

- an **enabling condition** expressed as a conditional C expression on elements of the current state vector.
- a **transition effect** expressed as C assignments to elements of the next state vector based on operations on the elements of the current state vector.
- a **transition rate or weight**. In the case of a timed transition, this is the mean transition firing rate while, in the case of an instanteous transition, this is a relative transition firing probability. The transition rate or weight may be state-dependent.

Interface Language: A model is specified across six files using a simple interface language containing C code fragments. Each of these files controls an aspect of the Markov chain generation and solution process. In particular, the user specifies:

- A **control** or master file.
- A **model description** file which specifies the high-level structure of the model, including a description of the state vector, an initial starting state and rules for transitions between states.
- A **state generation** control file which controls aspects of the state space exploration process, such as the maximum number of states.
- A **functional analysis** control file which allows for the specification of invariants which should hold on each generated state and which contains options for the detection of recurrent and transient state classes.
- A **steady-state solution** control file which specifies options such as the steady-state method to be used, the desired accuracy of the solution and the maximum number of iterations.
- A **quantitative analysis** file which specifies performance statistics in the form of state and count measures.

Three C programs are automatically generated from these files: a state space generator, a functional analyser and a combined steady-state and performance analyser. These programs are compiled and executed in sequence.

State Space exploration algorithm: USENUM uses an exhaustive depth-first search state space exploration algorithm which stores states in a hash table of linked lists. USENUM does not perform on-the-fly vanishing state elimination during state space generation. Instead, it eliminates vanishing states using matrix multiplication and inversion operations before beginning the steady-state solution. Timeless traps can be detected.

Numerical solution techniques: USENUM supports three classes of steady-state solution techniques:

- **Direct methods**: LU decomposition and Grassmann's algorithm.
- **Classical Iterative methods**: Jacobi Over-relaxation (JOR) and Successive Over-relaxation (SOR).
- **Decomposition-based techniques**: Block SOR.

Krylov subspace techniques are not implemented. However, transient analysis is supported through randomization and matrix powering.

Applications: Two front-end graphical applications have been developed which make use USENUM as their underlying solution engine: MACOM [KMCS90], a queueing network analyser tailored to telecommunications models and QPN-Tool [BK94], a Queueing Petri net analyser. Both applications run on SUN UNIX workstations under the SunView window system. Both automatically generate USENUM files from models without user intervention.

1.2.2 MARCA

MARCA (MARkov Chain Analyser) [Ste91] [Ste94, §10.2] is a Markov Chain analyser developed at North Carolina State University by Professor William J. Stewart. The package is written in FORTRAN and runs on UNIX systems.

State and Transition representation: In MARCA, a state is represented as an integer-valued row vector with elements known as *buckets*. Each bucket is viewed as containing a number of *balls* which represent the value of the corresponding vector elements. Transitions are represented by movements of balls from one bucket to another, and the *rate of transition* is defined as the rate at which the *source* bucket loses balls to the *destination* bucket. Such transitions can be *timed* or *instantaneous*.

Interface language: MARCA models are specified using a data file and two FORTRAN subroutines called *RATE* and *INSTANT*. The data file contains a description of the state descriptor vector, a maximum value for each bucket in the state descriptor, an initial state, and a list of transitions which can occur between buckets. The data file may also contain information such as the solution method for computing the steady-state distribution; alternatively, this information may be entered interactively during the solution process.

The subroutine *RATE* must be written to return the rate at which transitions occur between every possible pair of source and destination buckets, as well as the destination states that result from these transitions. The *RATE* subroutine is not restricted to changing to the source and destination buckets only, but may define the destination state completely. Note that this implies that the total number of balls in a state descriptor need not be conserved but can be created and destroyed as needed.

The subroutine *INSTANT* must examine destination states and determine whether or not they are vanishing. If they are, the subroutine must return a possible set of destination states and associate with each the probability that it is the result of the instanteous transition.

State Space exploration algorithm: MARCA implements an exhaustive breadth-first search state space exploration algorithm which stores states in a list. On-the-fly vanishing state elimination is performed during state space generation. There is no timeless trap detection.

Numerical solution techniques: MARCA implements four classes of steady-state solution methods:

- **Direct methods**: Sparse Gaussian elimination.

- **Classical Iterative methods**: SOR, symmetric SOR (SSOR), power method, fixed-point iterations with preconditioning.
- **Krylov Subspace methods**: Arnoldi method (2 variants), GMRES (3 variants).
- **Decomposition-based techniques**: An iterative aggregation/disaggregation solver for nearly completely decomposable (NCD) chains. MARCA includes a unique facility for detecting near-decomposable components of the transition matrix.

After the steady-state vector has been computed, MARCA determines the distribution of the balls in each of the buckets and the mean and standard deviation of the distributions. For more complex performance measures, MARCA allows the user access to the list of states and the steady-state probability vector.

In addition, MARCA supports transient analysis through randomization, Runge-Kutta, Adams ODE solver and matrix powering techniques.

Applications: A graphical front-end application for MARCA, known as XMARCA [KS95], has been developed. XMARCA is a sophisticated queueing network analyser which runs under the X-window system on UNIX systems. XMARCA allows users to build queueing networks from components such as stations, queues, servers and connectors and then analyse them using MARCA. XMARCA automatically generates the relevant MARCA files, including the *RATE* and *INSTANT* subroutines, and there is no need for user intervention.

1.3 Dissertation Outline

The layout of the rest of this dissertation is as follows:

Chapter 2 presents background material. In particular, we discuss three formalisms for describing timed transition systems, as well as the Markov theory necessary for their analysis.

Chapter 3 tackles the problem of efficient state space generation. We consider traditional state space exploration techniques and introduce a new probabilistic dynamic technique which is analysed in terms of its reliability and space complexity. The chapter concludes with a consideration of strategies for the efficient elimination of vanishing states which can occur in several time-augmented Petri net representations.

Chapter 4 presents a taxonomy of linear equation solvers, including direct methods, classical iterative methods, Krylov Subspace techniques and decomposition-based techniques. Particular attention has been paid to Krylov Subspace techniques and the Aggregation-Isolation algorithm [Tou95] which is a recently developed decomposition-based technique applicable to solving general Markov Chains.

Chapter 5 considers the requirements involved in designing a general interface language for specifying timed transition systems. A language which meets these requirements is presented.

Chapter 6 presents the DNAmaca performance analyser, which implements many of the concepts outlined in the previous chapters. DNAmaca provides a complete performance analysis sequence including model specification, state space generation, functional analysis, steady-state solution and the computation of performance statistics.

Chapter 7 illustrates the effectiveness of DNAmaca as a modelling tool by considering three examples of timed transition systems.

Chapter 8 presents conclusions and suggestions for future work.

Chapter 2

Background theory

2.1 Introduction

In the first section of this chapter, we discuss three formalisms for performance modelling which fall into the class of timed transition systems: Generalised Stochastic Petri nets (GSPNs), queueing networks and Queueing Petri nets (QPNs).

A timed transition system has one or more attributes which jointly characterise its behaviour. These attributes may have different values from time to time. A vector of these attributes, known as the *state descriptor vector*, characterises the configuration or *state* of the system at any point in time. All possible states of the system may be obtained by enumerating all possible values of the state descriptor. After defining each formalism, we describe what constitutes a *state* of the system and define a *state descriptor vector* whose components completely describe a state of the system. We also discuss the advantages and disadvantages of each methodology. We refer mainly to [BK95], [Rei92] and [LZGS84].

In the second section, we present an overview of Markov theory, which is the vehicle we will use for obtaining performance statistics from system models. Markov theory has been extensively covered in the literature; here we refer to [Kle75], [KS60], [Ste94] [BK95] and [BDMC+94].

2.2 Modelling Formalisms

2.2.1 Generalised Stochastic Petri Nets (GSPNs)

Petri nets are a modelling formalism for describing the behaviour of concurrently-executing asynchronous processes. Examples of systems which have been successfully modelled with Petri nets include communication protocols, parallel programs, multiprocessor memory caches and distributed databases [Pet81, Rei92].

The simplest kind of Petri nets are Place-Transition nets, which were originally conceived by Carl Adam Petri in 1962 as a formal means of establishing the *correctness* of concurrent systems. Place-Transition nets consist of four components [BK95]:

- **places**, drawn as circles, which model conditions or objects.
- **tokens**, drawn as black dots, which represent the specific value of the condition or object.
- **transitions**, drawn as rectangles, which model activities that change the values of conditions and objects.
- **arcs**, drawn between places and transitions and vice versa, which specify which objects are changed by a certain activity.

Definition 2.1 *A* Place-Transition net *is a 5-tuple* $PN = (P, T, I^-, I^+, M_0)$ *where*

- $P = \{p_1, \ldots, p_n\}$ *is a finite and non-empty set of places.*
- $T = \{t_1, \ldots, t_m\}$ *is a finite and non-empty set of transitions.*
- $P \cap T = \emptyset$.
- $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$ *are the backward and forward incidence functions, respectively. If* $I^-(p,t) > 0$*, an arc leads from place* p *to transition* t*, and if* $I^+(p,t) > 0$ *then an arc leads from transition* t *to place* p*.*
- $M_0 : P \rightarrow \mathbb{N}_0$ *is the initial marking defining the initial number of tokens on every place.*

Definition 2.2 *The dynamic behaviour of a Place-Transition net is determined by the enabling and firing of transitions, given as follows [BK95]:*

1. *A* **marking** *of a Place-Transition net is a function $M : P \rightarrow \mathbb{N}_0$, where $M(p)$ denotes the number of tokens in p.*
2. *A transition $t \in T$ is* **enabled** *at M, iff $M(p) \geq I^-(p,t), \forall p \in P$.*
3. *A transition $t \in T$, enabled at marking M, may* **fire** *yielding a new marking M' where*

$$M'(p) = M(p) - I^-(p,t) + I^+(p,t), \forall p \in P$$

4. *We say M' is* **directly reachable** *from M and write $M \rightarrow M'$. Let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$. A marking M' is* **reachable** *from M iff $M \rightarrow^* M'$.*

Using Place-Transition nets, we can test that a system has certain desirable *correctness* characteristics such as freedom from deadlock, liveness and boundedness. However, since Place-Transition nets do not include a notion of time, it is not possible to model the *performance* of a system. Consequently, several classes of *time-augmented* Petri nets have been developed, either by attaching time delays to transition firings or by specifying sojourn times of tokens on places.

One of the most flexible and most widely used time-augmented Petri net representations are Generalised Stochastic Petri nets (GSPNs) [AMCB84]. GSPNs have two different types of transitions: *immediate* transitions and *timed* transitions. Once enabled, immediate transitions fire in zero time, while timed transitions fire after an exponentially distributed firing delay. Firing of immediate transitions has priority over the firing of timed transitions.

The formal definition of a GSPN is as follows:

Definition 2.3 *A GSPN is a 4-tuple $GSPN = (PN, T_1, T_2, W)$ where*

- *$PN = (P, T, I^-, I^+, M_0)$ is the underlying Place-Transition net.*
- *$T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$,*
- *$T_2 \subset T$ denotes the set of immediate transitions, $T_1 \cap T_2 = \emptyset$, $T = T_1 \cup T_2$*

- $W = (w_1, \ldots, w_{|T|})$ *is an array whose entry* w_i
  - *is a (possibly marking dependent) rate* $\in \mathbb{R}^+$ *of an exponential distribution specifying the firing delay, when transition* t_i *is a timed transition, i.e.* $t_i \in T_1$ *or*
  - *is a (possibly marking dependent) weight* $\in \mathbb{R}^+$ *specifying the relative firing frequency, when transition* t_i *is an immediate transition, i.e.* $t_i \in T_2$.

Each distinct marking that is reachable from some initial marking M_0 corresponds to a state of the system. Thus the concepts of state and marking are interchangeable in the context of GSPNs and a suitable state descriptor is:

$$M = (p_1, p_2, \ldots, p_n)$$

where $n = |P|$. The set of all markings that are reachable from M_0 constitute the *state space* or *reachability set* of the Petri net.

The state space of a GSPN contains two types of markings. Since immediate transitions fire in zero time, the sojourn time in markings which enable immediate transitions is zero. Such markings are called *vanishing* markings because these states will never be observed by an observer who randomly examines the stochastic process of a GSPN, even though the stochastic process sometimes visits them. On the other hand, markings which enable timed transitions only will have an exponentially distributed sojourn time. Such markings are not left immediately and are referred to as *tangible* markings.

Since no time is spent in vanishing markings, vanishing markings have no effect on the resulting performance statistics derived for a GSPN and they are often eliminated during or immediately after state space generation.

The graphical representation of GSPNs becomes very complex for realistic models. One way of reducing this complexity is to distinguish between individual tokens. Coloured GSPNs (CGSPNs) [DCB93] make this distinction by attaching *colour* to tokens and by defining *firing modes* on transitions.

Before formally defining a CGSPN, we must first define *multisets* and *Coloured Petri nets* (CPNs). CPNs are the coloured variants of Place-Transition nets on which CGSPNs are based.

Definition 2.4 *A* **multiset** *m over a non-empty set S, is a function $m \in [S \rightarrow \mathbb{N}_0]$. The non-negative integer $m(s) \in \mathbb{N}_0$ is the number of appearances of the element s in the multi-set m.*

Definition 2.5 *A* **Coloured Petri net** *(CPN) is a 6-tuple $CPN=(P,T,C,I^-,I^+,M_0)$, where*

- *P is a finite and non-empty set of places,*
- *T is a finite and non-empty set of transitions,*
- *$P \cap T = \emptyset$,*
- *C is a colour function defined from $P \cup T$ into finite and non-empty sets,*
- *I^- and I^+ are the backward and forward incidence functions defined on $P \times T$ such that $I^-(p,t), I^+(p,t) \in [C(t) \rightarrow C(p)_{MS}], \forall\ (p,t) \in P \times T$,*
- *M_0 is a function defined on P describing the initial marking such that $M_0(p) \in C(p)_{MS}, \forall p \in P$.*

Definition 2.6 *The dynamic behaviour of a CPN is given as follows:*

1. *A transition $t \in T$ is* **enabled** *in a marking M w.r.t. a colour $c' \in C(t)$, denoted by $M[(t,c') >$, iff $M(p)(c) \geq I^-(p,t)(c')(c), \forall p \in P, c \in C(p)$.*
2. *An enabled transition $t \in T$ may furthermore* **fire** *in a marking M w.r.t. a colour $c' \in C(t)$ yielding a new marking M', denoted by $M \rightarrow M'$ or $M[(t,c') > M'$, with $M'(p)(c) = M(p)(c) + I^+(p,t)(c')(c) - I^-(p,t)(c')(c), \forall p \in P, c \in C(p)$.*

Definition 2.7 *A* **Coloured GSPN** *(CGSPN) is a 4-tuple $CGSPN = (CPN, T_1, T_2, W)$ where*

- *$CPN = (P,T,C,I^-,I^+,M_0)$ is the underlying Coloured Petri net.*
- *$T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$,*

- $T_2 \subset T$ *is the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$, $T = T_1 \cup T_2$,
- $W = (w_1, \ldots, w_{|T|})$ *is an array whose entry* w_i *is a function of* $[C(t_i) \rightarrow \mathrm{IR}^+]$ *such that* $\forall c \in C(t_i) : w_i(c) \in \mathrm{IR}^+$
  - *is a (possibly marking dependent) rate of a negative exponential distribution specifying the firing delay with respect to colour c, if* $t_i \in T_1$ *or*
  - *is a (possibly marking dependent) firing weight with respect to colour c, if* $t_i \in T_2$.

Note that CGSPNs do not have any additional modelling power over GSPNs since every CGSPN may be uniquely unfolded into a GSPN representing the same model.

GSPNs provide a natural way of modelling synchronisation, but several difficulties arise when attempting to model queues [BK95, pg. 152–153]. Even simple scheduling strategies like FCFS are difficult to represent with low-level Petri net elements; in addition, advance knowledge of the maximum number of elements in a queue is required and it is extremely difficult to model service times given by complex distributions, e.g. a Coxian distribution.

2.2.2 Queueing Networks

Queueing networks [BCMP75, LZGS84, Wal88b] are a widely-used performance analysis technique for those systems which can be naturally represented as networks of queues. Systems which have been successfully analysed with queueing networks include computer systems, communication networks and flexible manufacturing systems.

A queueing network consists of three types of components:

- **Service centres** (see Fig. 2.2.2), each of which consists of one or more *queues* and one or more *servers*. The servers represent the resources of the system available to service customers. An arriving customer will immediately be served if a free server can be allocated to the customer or if a customer in service is preempted. Otherwise, the customer must wait in one of the queues, until a server becomes available.
- **Customers**, which demand service from the service centres and which represent the load on the system. Usually customers are grouped into classes, where customers in one class exhibit similar behaviour and normally place similar demands on the centres.

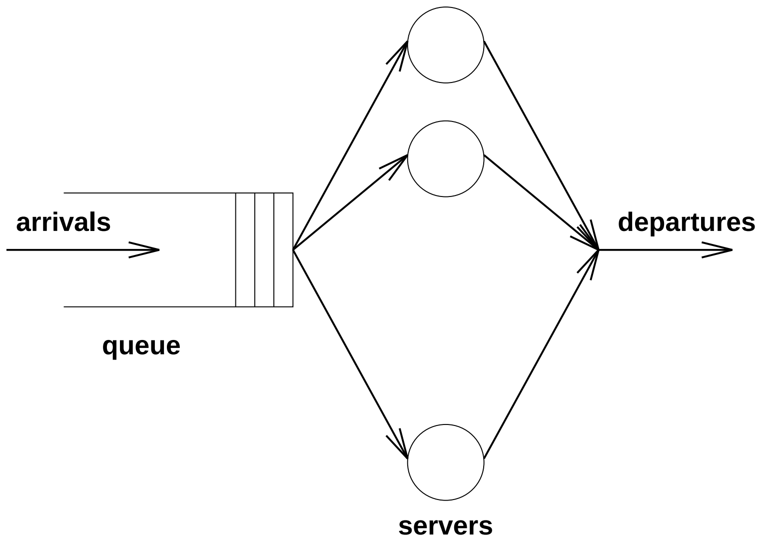


Figure 1: A service centre in a queueing network

- **Routes**, which are the paths which workloads follow through a network of service centres. The routing of customers may be dependent on the state of the network. If the routing is such that no customers may enter or leave the system, the system is said to be *closed*. If customers arrive externally and eventually depart, the system is said to be *open*. If some classes of customers are closed and some are open, then the system is said to be *mixed*.

To be fully specified, a queueing network requires the following parameters to be defined:

- The **number of service centres**.
- The **number of queues** at each service centre. For each of these queues we further need to define:
  - The **capacity of each queue**, which may be infinite or finite of capacity k.
  - The **queue scheduling discipline**, which determines the order of customer service. Different customer classes may have different scheduling priorities. Common scheduling rules include First-Come First-Served (FCFS), Last-Come First-Served Preemptive-Resume (LCFS-PR), Round Robin (RR) and Processor Sharing (PS).

  - For open classes of customers, we need to define an **input source distribution** specifying the arrival distribution of each customer class at each queue. This distribution is usually given by an exponential distribution with parameter λ.

- The **number of servers** at each service centre. For each of these servers we further we need to define:

  - The **service time distribution** for each customer class at each server. This is usually exponential with parameter μ. More general distributions can be approximated using a Coxian distribution [Ste94, pg. 51–52].

- The **routing probability matrix** for each customer class. This matrix specifies the probabilistic routing of customers between service centres, with the ijth element giving the probability that a customer leaving service centre i will proceed to service centre j. These transitions are assumed to be instantaneous.

The state of individual service centres in a queueing network may be described by a vector. For example, the state of a single-server centre with a Coxian service distribution and blocking may be described by the number of customers in the queue, the phase of service and a parameter to indicate whether or not the server is blocked. The state descriptor of a queueing network as a whole may then be built up by concatenating the vectors describing the state of the individual service centres.

A certain class of queueing networks which satisfy *reversibility* [Kel79] can be efficiently analysed using so-called product-form solution techniques, the two most well-known of which are Mean Value Analysis (MVA) and the convolution method. Unfortunately, these elegant algorithms fail if one of the prerequisites for the product-form property is violated by the network. In particular, if phenomena such as synchronization, simultaneous resource possession, blocking or batching occur, then usually no proper product-form queueing network model can be found. In this case, strictly numerical procedures have to be used. In particular, one may always derive and solve a Markov chain model of the system.

Queueing networks are widely used because they are often easy to define, parameterise and evaluate. However, they lack of facilities to describe synchronisation mechanisms.

2.2.3 Queueing Petri nets (QPNs)

Queueing Petri nets (QPNs) [Bau93] attempt to incorporate the concept of queues into a coloured GSPN formalism. In this way, synchronisation mechanisms and queues with various scheduling strategies can be integrated into one model. A QPN extends the concept of a CGSPN by partitioning the set of places into two subsets: *queued places* and *ordinary places*.

A queued place (see Fig. 2.2.3) consists of two parts: a queue and a depository for tokens which have completed their service at this queue. Tokens, when fired onto a queued place by any of its input transitions, are inserted into the queue according to the scheduling strategy of the queue. Tokens in a queue are not available for transitions. After completion of its service, the token is placed onto the depository. Tokens on this depository are available to all output transitions of the queued place. An enabled timed transition will fire after an exponentially distributed time delay and an immediate transition fires with no delay, as in GSPNs.

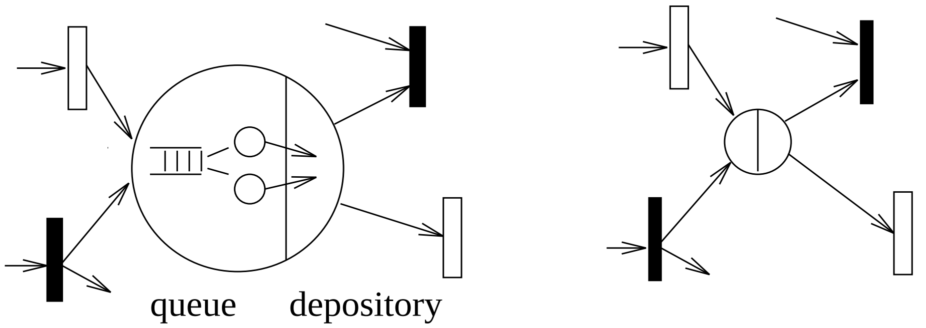


Figure 2: A queued place and its shorthand notation

The formal definition of a QPN is as follows:

Definition 2.8 *A* **Queueing Petri net** *(QPN) is a triple* $QPN = (CGSPN, P_1, P_2)$ *where:*

- *CGSPN is the underlying Coloured GSPN*
- $P_1 \subseteq P$ *is the set of queued places and*
- $P_2 \subseteq P$ *is the set of ordinary places,* $P_1 \cap P_2 = \emptyset$*,* $P = P_1 \cup P_2$*.*

A state or marking M of a QPN consists of two parts $M = (n, m)$ where n specifies the state of all queues and m is the marking of the underlying CGSPN. For a queued place $p \in P_1$, $m(p)$ denotes the marking of the depository. The initial marking M_0 of a QPN is given by $M_0 = (O, m_0)$ where O is the state describing that all queues are empty and m_0 is the initial marking of the CGSPN.

There are three possible types of state transitions that may take place in a QPN:

- An enabled immediate transition may fire.
- An enabled timed transition may fire if no immediate transitions are enabled.
- A service in a queued place may complete if no immediate transitions are enabled.

Similar to GSPNs, the firing of immediate transitions has priority over both the firing of timed transitions and the service of tokens in queues. Thus, the state space of a QPN comprises both vanishing and tangible states. As for GSPNs, these states are usually eliminated during or immediately after state space generation.

QPNs allow for a convenient description of queues within a Petri net paradigm. However, the complexity of the performance analysis, determined by the size of the state space, is still the same as that obtained by modelling the queue with CGSPN elements.

2.3 Markov Theory

2.3.1 Stochastic Processes

As we have mentioned, the behaviour of a system can often be characterised by enumerating all the states that the system may enter and by describing how the system evolves from one state to another over time. In its most general form, such a system can be represented by a *stochastic process.*

Definition 2.9 *A* **random variable** χ *is a variable whose value depends on the outcome of a random experiment. If the value space of* χ *is countable but not necessarily finite, then the random variable is* **discrete** *and its behaviour is characterised by a probability mass function:*

$$p_\chi(x) = P\{\chi = x\}$$

If the value space of χ is uncountable, then the random variable is **continuous** *and its behaviour is characterised by a cumulative distribution function:*

$$F_\chi(x) = P\{\chi \leq x\}$$

Definition 2.10 *A* **stochastic process** *is a family of random variables $\{\chi(t)\}$ indexed by the time parameter t. If the time index set $\{t\}$ is countable, the process is a* **discrete-time** *process, otherwise the process is a* **continuous-time** *process. The possible values or states that members of $\{\chi(t)\}$ can take on constitute the* **state space** *of the process. If the state space is discrete the process is called a* **chain**.

2.3.2 Markov Processes

In many cases, the future evolution of a system depends only on the current state of the system and not on past history. Such *memoryless* systems can be represented by *Markov processes*. Stated formally, Markov processes satisfy the *Markov property* which states that for all integers n and for any sequence $t_0, t_1, \ldots, t_n$ such that $t_0 \leq t_1 \leq \ldots \leq t_n \leq n$ we have

$$P\{\chi(t) \leq x \mid \chi(t_n) = x_n, \chi(t_{n-1}), \ldots, \chi(t_0)\} = P\{\chi(t) \leq x \mid \chi(t_n) = x_n\}$$

This property requires that the next state can be determined knowing nothing other than the current state, not even how much time has been spent in the current state. A consequence of the Markov property is that the sojourn time τ spent in a state must satisfy:

$$P\{\tau \geq s + t \mid \tau \geq t\} = P\{\tau \geq s\} \quad \forall\ s, t \geq 0 \tag{1}$$

As we shall see, this condition places restrictions on the distribution of time spent in a state.

Definition 2.11 *A* **homogeneous Markov chain** *is a Markov chain whose probabilities are stationary with respect to time. That is:*

$$P\{\chi(t) \leq x \mid \chi(t_n) = x_n\} = P\{\chi(t - t_n) \leq x \mid \chi(0) = x_n\}$$

Discrete-time Markov Chains

A discrete-time Markov chain is a Markov process with a discrete state space which is observed at a discrete set of times. Without loss of generality, we can take the time index set $\{t\}$ to be the set of counting numbers $\{0, 1, 2, \ldots\}$. The observations at these times define the random variables $\chi_0, \chi_1, \chi_2, \ldots$ at time steps $0, 1, 2, \ldots$ respectively.

Definition 2.12 *The variables $\chi_0, \chi_1, \ldots$ form a* **discrete-time Markov chain** *if for all n ($n = 1, 2, \ldots$) and all states x_n we have:*

$$\begin{aligned} P\{\chi_{n+1} = x_{n+1} \mid \chi_0 = x_0, \chi_1 = x_1, \ldots, \chi_n = x_n\} \\ = P\{\chi_{n+1} = x_{n+1} \mid \chi_n = x_n\} \end{aligned}$$

For a discrete-time Markov chain, the only sojourn time distribution which satisfies the sojourn time condition of Eq. (1) is the geometric distribution.

A homogenous discrete-time Markov chain may be represented by a one-step transition probability matrix P with elements:

$$p_{ij} = P\{\chi_{n+1} = x_j \mid \chi_n = x_i\}$$

where χ_t represents the state of the system at discrete time-step $t \in \mathbb{N}$. That is, p_{ij} gives the probability of x_j being the next state given that x_i is the current state. Note that the entries of P must satisfy:

$$0 \leq p_{ij} \leq 1 \ \forall\ i, j \quad \text{and} \quad \sum_j p_{ij} = 1 \ \forall\ i$$

Definition 2.13 *Let S_0 denote a subset of the state space S, and $\overline{S_0}$ its complement. Then S_0 is* **closed** *or* **final** *if no single-step transition is possible from any state in S_0 to any state in $\overline{S_0}$.*

Definition 2.14 *A Markov chain is* **irreducible** *if every state can be reached from every other state. Otherwise, the state space contains one or more* **closed** *subsets of states and the chain is* **reducible***.*

Let $f_j^{(m)}$ denote the probability of leaving state x_j and first returning to that same state in m steps. Then the probability of ever returning to the state x_j is given by:

$$f_j = \sum_{m=1}^{\infty} f_j^{(m)}$$

Definition 2.15 *For any state x_j:*

- *if $f_j = 1$ then x_j is* **recurrent***; else*
- *if $f_j < 1$ in which case x_j is* **transient**.

Definition 2.16 *State x_j is* **periodic** *with period η if the Markov chain returns to state x_j only at time steps $\eta, 2\eta, 3\eta, \ldots$ where $\eta \geq 2$ is the smallest such integer. If $\eta = 1$ then x_j is* **aperiodic**.

Definition 2.17 *The* **mean recurrence time** *of recurrent state x_j is*

$$M_j = \sum_{m=1}^{\infty} m f_j^{(m)}$$

which is the average number of steps taken to return to state x_j for the first time after leaving it. If $M_j = \infty$, state x_j is **recurrent null***; otherwise $M_j < \infty$ and x_j is* **recurrent nonnull**.

Theorem 2.1 *The states of an irreducible Markov chain are either all transient or all recurrent nonnull or all recurrent null. If the states are periodic, then they all have the same period [Kle75, pg. 29].*

The most important part of Markov chain analysis is to determine how much time is spent in each of the states x_j. We define:

$$\pi_j^{(m)} = P\{\chi_m = x_j\}$$

as the probability of finding the Markov chain in state x_j at time step m.

Definition 2.18 *Let z be a vector describing a probability distribution whose elements z_j denote the probability of being in state x_j. Then, z is a* **stationary probability distribution** *of a DTMC with one-step transition matrix P if and only if $zP = z$.*

Definition 2.19 *The* **limiting probability distribution** $\{\pi_j\}$ *of a discrete-time Markov chain is given by:*

$$\pi_j = \lim_{m \to \infty} \pi_j^{(m)}$$

Note that the existence of a stationary distribution of a Markov chain does not necessarily imply the existence of a limiting probability distribution, and vice versa. The next theorem addresses the issue of when the limiting and stationary probabilities exist.

Theorem 2.2 *In an irreducible and aperiodic homogeneous Markov chain, the limiting probabilities $\{\pi_j\}$ always exist and are independent of the initial probability distribution. Moreover one of the following conditions hold:*

- *Every state x_j is transient or every state x_j is recurrent null, in which case $\pi_j = 0$ for all x_j and there exists no stationary distribution (even though the limiting probability distribution exists). In this case, the state space must be infinite.*
- *Every state x_j is recurrent nonnull with $\pi_j > 0$ for all x_j, in which case the set $\{\pi_j\}$ is a limiting and stationary probability distribution and*

  $$\pi_j = \frac{1}{M_j}$$

  In this case the π_j are uniquely determined from the set of equations:

  $$\sum_i \pi_j = \pi_i p_{ij} \quad \textit{subject to} \quad \sum_i \pi_i = 1 \tag{2}$$

If $\pi = (\pi_1, \pi_2, \ldots)$ is a vector of limiting probabilities, Eq. (2) can be rewritten as

$$\pi = \pi P$$

where P is the transition probability matrix. The vector π is called the *steady-state solution* of the Markov chain.

The states of a recurrent nonnull discrete-time Markov chain are said to be *ergodic*, as is the Markov chain itself. If the state space of the Markov chain is finite (which we will always assume is the case), the chain is called *finite*; if, in addition, the chain is irreducible, then it is *ergodic*.

Continuous-Time Markov Chains

A continuous-time Markov chain is a Markov process with a discrete state space and a state that may change at any time. The formal definition is as follows:

Definition 2.20 *The stochastic process* $\{\chi(t)\}$ *forms a* **continuous-time Markov chain** *if for all integers* n *and for any sequence* $t_0, t_1, \ldots, t_n, t_{n+1}$ *with* $t_0 < t_1 < \ldots < t_n < t_{n+1}$ *we have:*

$$P\{\chi(t_{n+1}) = x_{n+1} \mid \chi(t_0) = x_0, \chi(t_1) = x_1, \ldots, \chi(t_n = x_n)\}$$
$$= P\{\chi(t_{n+1}) = x_{n+1} \mid \chi(t_n) = x_n\}$$

For the case of a continuous-time Markov chain, the only sojourn time distribution which satisfies the sojourn time condition of Eq. (1) is the exponential distribution.

A homogeneous continuous-time Markov chain may be represented by a set of states and an *infinitesimal generator matrix* Q where $Q_{ij},\ i \neq j$ represents the exponentially distributed transition rate between states x_i and x_j. The parameter of the exponential distribution of the sojourn time in state x_i is given by $-Q_{ii}$ where $Q_{ii} = -\sum_{j\neq i} Q_{ij}$. Note that the entries of Q must satisfy:

$$\sum_j Q_{ij} = 0 \quad \forall\ i$$

Definition 2.21 *Let* z *be a vector describing a probability distribution whose elements* z_j *denote the probability of being in state* x_j*. Then,* z *is a* **stationary probability distribution** *of a CTMC with infinitesimal generator matrix* Q *if and only if* $\pi Q = 0$.

Theorem 2.3 *In a finite, homogeneous, irreducible, continuous-time Markov chain, the limiting probabilities* $\{\pi_j\}$ *always exist and are independent of the initial probability distribution. Moreover, the set* $\{\pi_j\}$ *is also a stationary probability distribution which can be uniquely determined from solving the set of equations:*

$$q_{jj}\pi_j + \sum_{i\neq j} q_{ij}\pi_i \quad \textit{and} \quad \sum_i \pi_i = 1 \tag{3}$$

The set of equations given by Eq. (3) is sometimes also referred to as the set of *global balance equations*. In vector form, they may be written as:

$$\pi Q = 0$$

where $\pi = (\pi_1, \pi_2, \ldots)$ is the steady-state probability vector.

Chapter 3

State Space Exploration Techniques

3.1 Introduction

The first step in the analytical performance analysis of general timed transition systems is to determine what the reachable states in a system are and how they relate to one another. We will approach this problem by using explicit state enumeration techniques; note that if the underlying model has certain pre-existing structural properties, or if the only objective is to decide the correctness of the system being modelled, then other techniques exist to handle very large state spaces [Kem95, B^+94]. The objective here, however, is to generate the state spaces of *unrestricted* systems for the purpose of *performance analysis*.

Most state space generation techniques use a depth first search (DFS) approach. This requires the use of two data structures. First, a DFS stack is needed to store unexplored states. Second, a table of explored states must be maintained in order to avoid redundant state exploration. Given a DFS stack S, a set of explored states E, an initial state s_i and a function succ(s) which yields the set of successors states of state s, Fig. 3 presents an outline of a basic state exploration algorithm.

The DFS stack is accessed sequentially and is limited by the depth of the state graph; thus it is usually not critical to memory requirements. However, the table of explored states must hold enough information to determine whether states encountered are new, or have

push(S, s_i)
$E = \{s_i\}$
while (S not empty) **do begin**
    pop(S, s)
    for each $s' \in$ succ(s) **do begin**
        if $s' \notin E$ **do begin**
            push(S, s')
            $E = E \cup \{s'\}$
        end
    end
end

Figure 3: Basic state space exploration algorithm

in fact been explored before. That is, one has to be able to rapidly store and retrieve information about *every* reachable state. Consequently, the layout and management of the explored-state table is crucial to both the time and space efficiency of a state space generation technique.

It is worthwhile to note that information about transitions between states can be obtained efficiently as the DFS search proceeds. As each unexplored state s is popped off the DFS stack, the function succ(s) finds the set of enabled transitions at state s and then fires each one to determine the successor states. Provided information about the firing rates of these enabled transitions is available, we can easily adapt the algorithm to construct the infinitesimal generator matrix of transition rates between states. Note that the generator matrix need not be stored in memory; instead, as each state is popped off the DFS stack, the matrix can be written to secondary storage row-by-row for later use.

3.2 Traditional state space exploration techniques

State space exploration algorithms are distinguished by two aspects:

- **Memory allocation strategy**. *Static* techniques preallocate large blocks of memory for the explored-state table. Since the number of states in the system is in general not known beforehand, the preallocated memory may not be sufficient (resulting in a failure of the method) or may be a gross overestimation (resulting a waste of resources, especially in a multi-user environment). *Dynamic* techniques, on the other hand, allocate memory only as needed; the more states, the more memory is allocated.

- **Reliability**. *Exhaustive* techniques store the complete state space and guarantee complete state space coverage. However, storing the complete state space uses a large amount of memory; this severely limits the number of states that can be explored. *Probabilistic* techniques use space-saving techniques (usually based on hashing) to drastically reduce the memory required to store states. However, this reduction comes at the price of possibly incorrectly recording a state as explored when it is in fact an unexplored state. This can result in the omission of a state (and also some or all of its successors) from the hash table. Since omitting states will result in incorrect information about the transitions between states - and thus incorrect performance statistics - it is important to keep the probability of missing even one state small (say less than 1%).

This framework can be used to classify traditional state exploration algorithms into two main groups: exhaustive dynamic techniques and probabilistic static techniques.

3.2.1 Exhaustive dynamic techniques

Linked list/dynamic array

Perhaps the most obvious way of tackling the problem of explicit state enumeration is to use a dynamic array or a linked list to store the complete state descriptor of every state encountered so far. This simple scheme is shown in Fig. 4.

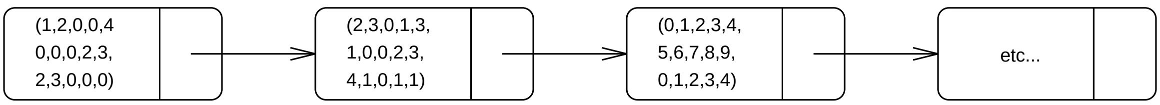


Figure 4: Simple linked list state storage scheme

This approach is used by the MARCA [Ste91, KS95] analyser. It guarantees complete state space coverage and is attractive for it's simplicity of coding. However, given a state descriptor of d bytes, storage for n states requires dn bytes of memory (assuming optimal storage in a dynamic array). This can be extremely limiting considering that the state descriptor size d may easily be hundreds of bytes long. For example, given 32MB of available memory and a state descriptor of 100 bytes, this approach limits the state space to about 320 000 states.

This method also involves a considerable search overhead whenever a state is popped off the exploration stack. "Unsuccessful" searches of the list/dynamic array (which establish that a state has not yet been explored) involve n state descriptor comparisons, while "successful" searches (which establish that a state has already been explored) involve on average $\frac{n}{2}$ state descriptor comparisons.

Hash table with full state information

The large search overhead incurred by the list method can be remedied by using a hash table with separate chaining to break up the state list into several smaller lists/dynamic arrays. Given r rows in the hash table, a hash function h_1 based on the contents of a state descriptor s is used to return a key $h_1(s)$. The value of $h_1(s)$ ranges from 0 to $r - 1$ and denotes which of the rows in the hash table should contain the state. This arrangement is shown in Fig. 5.

This is the approach adopted in the USENUM analyser [Scz87] and the correctness analyser of the DNAnet Petri net tool [ABK95]. It guarantees complete state space coverage and reduces search times to an average n/r state descriptor comparisons for unsuccessful searches and an average $n/2r$ comparisons for successful searches. However, memory requirements for the scheme are greater since there is now hash table overhead to consider. Given h bytes of overhead for each hash table row, total memory required is $dn + rh$ bytes.

3.2.2 Probabilistic static techniques

Holzmann's bit-state hashing method

Holzmann's bit-state hashing method [Hol91, Hol95] maximizes state coverage in the face of limited memory by minimising the memory used to store the explored-state table. Here the table takes the form of a bit vector T. A hash function h is used to map states onto positions in this bit vector. Initially all bits in T are set to zero. When a state s is inserted into the table, its corresponding bit $T[h(s)]$ is set to one. To check whether a state s is already in the table, the value of $T[h(s)]$ is examined. If it is zero, the state has not yet been

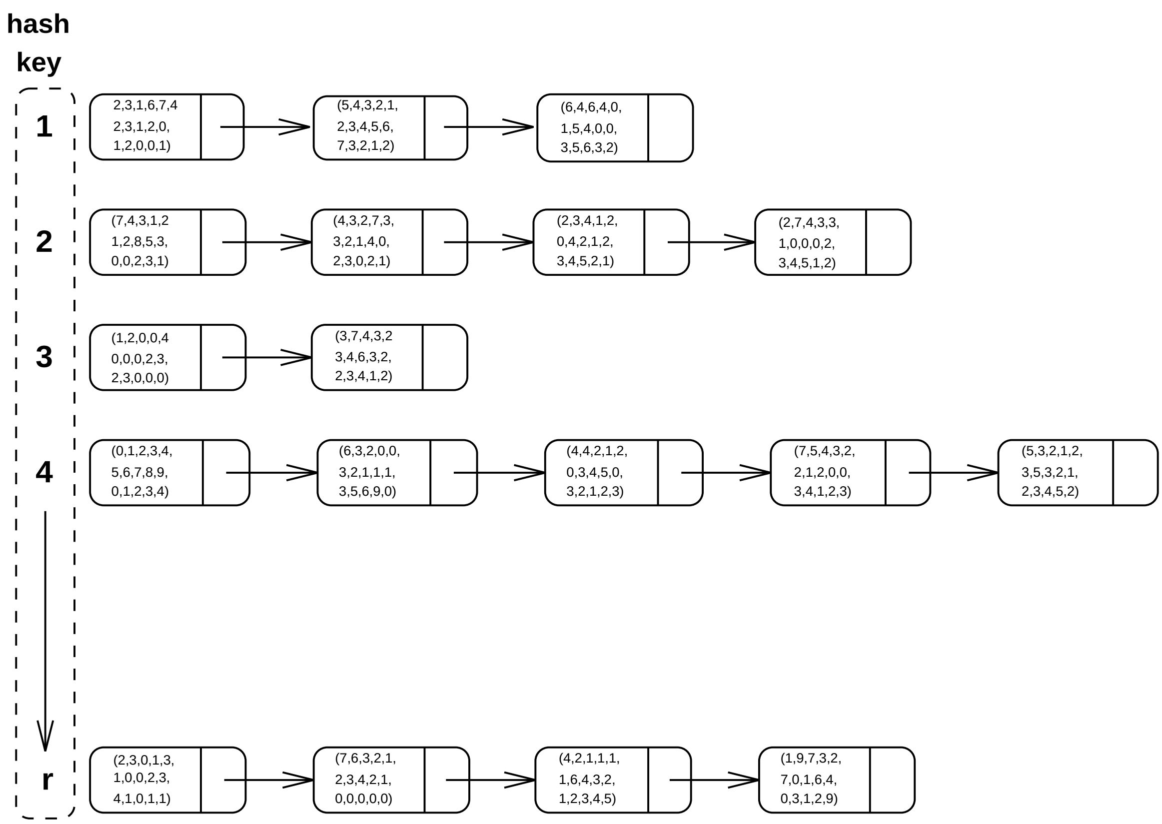


Figure 5: Hash table with full state information

explored; otherwise it is assumed that the state has already been explored. The problem is that two distinct states can hash to the same hash value, with the result that one of the states will be incorrectly classified as explored. Given n states to be inserted into the table and a bit vector of t bits, the probability of no hash collisions p is given by:

$$p = \frac{t(t-1)\ldots(t-n+1)}{t^n} = \frac{t!}{(t-n)!\ t^n}$$

By using Stirling's approximation for $n!$ and assuming the favourable case of $n << t$, Wolper and Leroy [WL93] approximate the probability of no hash collisions p as:

$$p \approx e^{-\frac{n^2}{t}}$$

where n is the number of states and t is the size of the bit vector. Unfortunately the table sizes required to keep p very low (as we require) are impractically large. The situation can be improved a little by using two independent hash functions h_1 and h_2. When inserting a state s, both $T[h_1(s)]$ and $T[h_2(s)]$ are set to 1; we only conclude s has been explored if $T[h_1(s)]$ and $T[h_2(s)]$ are set to one. Now Wolper and Leroy show the probability of no hash collisions is approximately

$$p \approx e^{-\frac{4n^3}{t^2}}$$

but the table sizes required to keep p low are still impractically large. The strength of Holzmann's method thus lies in its ability to maximize coverage in the face of limited memory and not in it's ability to provide complete state space coverage.

Leroy and Wolper's hash compaction method

The problem with Holzmann's bit-state hashing method is that the ratio of states to hash table entries must be kept extremely low if our aim is to provide a good probability of complete state space coverage. Consequently, a large amount of the memory allocated for the bit vector is wasted. Wolper and Leroy [WL93] observed that it would be better to store which positions in the table are occupied instead. This can be done by hashing states onto compressed values of k bit keys; these keys can then be stored in a smaller hash table which supports a collision resolution scheme. Given a hash table with $m \geq n$ slots, the memory required by this scheme is approximately $(mk+m)/8$ bytes, since we need to store the keys, as well as information about which hash table slots are occupied.

This approach simulates a bit-state hashing scheme with a table size of 2^k, so the probability of no collision is approximately given by

$$p \approx e^{-\frac{n^2}{2^k}}$$

Leroy and Wolper recommend using compressed values of $k = 64$ bits, i.e. 8-byte compression.

Stern and Dill's improved hash compaction method

In their description of standard hash compaction, Leroy and Wolper give no details of how states are mapped onto slots in the smaller hash table; it is implicitly assumed that hash values (used to determine where in the hash table to store the k-bit compressed values) are calculated using the k-bit compressed values. However, Stern and Dill [SD95] observed that the omission probability can be dramatically reduced in two ways. Firstly, by calculating the hash values and compressed values independently and, secondly, by using a collision resolution scheme which keeps the number of probes per insertion low. This improved technique is so effective that it requires only 5-bytes per state in situations where Wolper and Leroy's standard hash compaction requires 8-bytes per state.

Given a hash table with m slots, states are inserted into the table using two hash functions $h_1(s)$ and $h_2(s)$ which generate the probe sequence $h^{(0)}(s), h^{(1)}(s), \ldots, h^{(m-1)}(s)$ with $h^{(i)}(s) = (h_1(s) + ih_2(s)) \bmod m$ for $i = 0, 1, \ldots, m-1$. This double hashing scheme prevents the clustering associated with simple rehashing algorithms such as linear probing. A separate independent compression function h_3 is used to calculate the k-bit compressed state values which are stored in the table.

The complete procedure for inserting a state s into the table is as follows:

- $h_3(s)$ is calculated to determine the state's compressed value.
- $h_1(s)$ and $h_2(s)$ are used to determine a probe sequence $h^{(0)}(s), h^{(1)}(s), \ldots, h^{(m-1)}(s)$ for inserting the state s. Slots are probed in this order, until one of two conditions are met:
  - If the slot currently being probed is empty, the compressed value is inserted into the table at that slot and the state's successors are pushed onto the state exploration stack.

– If the slot is occupied by a compressed value equal to the $h_3(s)$, we assume (possibly incorrectly) that the state has already been explored. No states are pushed onto the state exploration stack.

Given m slots in the hash table, n of which are occupied by states, the probability of no state omissions p is given approximately by

$$p \approx \prod_{k=0}^{n-1} \left[\sum_{j=0}^{k} \left(\frac{2^k - 1}{2^k} \right)^j \frac{m-k}{m-j} \prod_{i=0}^{j-1} \frac{k-i}{m-i} \right]$$

This formula takes $O(n^3)$ operations to evaluate. Stern and Dill derive a $O(1)$ approximation given by

$$p \approx \left(\frac{2^k - 1}{2^k} \right)^{(m+1)\ln(\frac{m+1}{m-n+1}) - \frac{n}{2(m-n+1)} + \frac{2n+2mn-n^2}{12(m+1)(m-n+1)^2} - n}$$

Stern and Dill also derive a more straightforward formula for the approximate maximum omission probability q for a full table (i.e. with $m = n$):

$$q \approx \frac{1}{2^k} m(\ln m - 1)$$

which shows the omission probability is approximately proportional to $m \ln m$. Increasing k, the number of bits per state, by one roughly halves the maximum omission probability.

3.3 A new probabilistic dynamic technique

None of the methods mentioned above has the advantage of being both probabilistic and dynamic. In this section we propose a new technique which uses dynamic storage allocation while yielding a good collision avoidance probability. We use a hash table of linked lists (as used in an exhaustive, dynamic technique) but instead of storing full state descriptors in the lists, we store compressed state descriptors (as in hash compaction).

Two independent hash functions are used. The *primary* hash function h_1 is used to determine which hash table row should be used to store a compressed state and the *secondary* hash function h_2 is used to compute the compressed state descriptor values. Both h_1 and h_2 are assumed to distribute states randomly and independently of one another; the H_3 class of hash functions defined by Carter and Wegman [CW79] satisfies this property. If a state's secondary key is already present in the hash table row given by its primary key, then the state is deemed to have been explored and no further action is taken. Otherwise, the secondary key is added to the hash table row and its successors are pushed onto the state exploration stack. This scheme is illustrated in Fig. 6.

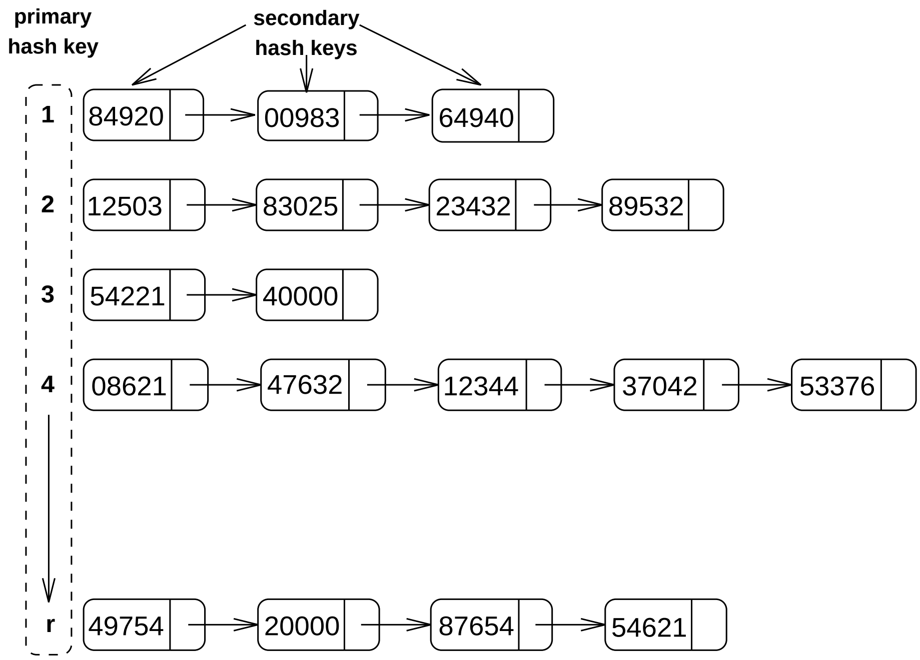


Figure 6: Hash table with compressed state information

The complete procedure for inserting a state s into the table is as follows:

- The primary key $h_1(s)$ is calculated to determine which row of the hash table should hold the new state.
- The secondary key $h_2(s)$ is calculated and compared to the secondary keys stored in the hash table row given by $h_1(s)$.
- If the secondary key is already present, the state is deemed to have been "explored" and no further action is taken. Otherwise the secondary key is added to the hash table row and the state's successors are pushed onto the state exploration stack.

Note that two states s_1 and s_2 are classified as being equal if and only if $h_1(s_1) = h_1(s_2)$ and $h_2(s_1) = h_2(s_2)$; this may happen even when the two state descriptors are different, so that collisions may occur, as in all other probabilistic methods.

3.3.1 Reliability analysis

We will now calculate the probability of complete state space coverage. We are given that:

- there are r rows in the hash table.
- there are $t = 2^b$ unique secondary key values.
- $h_1(s)$ is the primary hash function used to determine which hash table row should hold state s. It returns key values from 0 to $r - 1$.
- $h_2(s)$ is the secondary hash function which returns a compressed state vector. Its key values range from 0 to $t - 1$.
- $h_1(s)$ and $h_2(s)$ distribute states randomly and independently of one another.
- there are n unique states $s_1, s_2, \ldots, s_n$ to be inserted into the hash table.

Let X_ℓ^n be a random variable denoting the number of states allocated to row $\ell, 1 \leq \ell \leq r$, given that there are n unique state identifiers to be inserted into the table. Then, since we assumed that h_1 distributes states randomly, X_ℓ^n will have a binomial distribution with parameters n and $p = 1/r$, i.e.,

$$P\{X_\ell^n = j\} = \left(\begin{array}{c} n \\ j \end{array} \right) (\frac{1}{r})^j (1 - \frac{1}{r})^{n-j} = \left(\begin{array}{c} n \\ j \end{array} \right) \frac{(r-1)^{n-j}}{r^n}$$

Denoting the number of clashes in row ℓ by C_ℓ and considering the case when there are j states in row ℓ, we have:

$$P\{C_\ell = 0 | X_\ell^n = j\} = \frac{t(t-1)(t-2)\ldots(t-j+1)}{t^j} = \frac{t!}{(t-j)!\, t^j}$$

where $t = 2^b$ is the number of unique secondary key values and b is a positive integer denoting the number of bits used to store the secondary key. Then,

$$\begin{aligned} P\{C_\ell = 0\} &= \sum_{j=0}^{n} P\{C_\ell = 0 | X_\ell^n = j\} P\{X_\ell^n = j\} \\ &= \frac{1}{r^n} \sum_{j=0}^{n} \left(\begin{array}{c} n \\ j \end{array} \right) \frac{(r-1)^{n-j} t!}{(t-j)!\, t^j} \end{aligned}$$

If C_r denotes the total number of clashes across all rows r of the hash table, the probability p of no clash in *any* row of the hash table is simply given by:

$$\begin{aligned} p &= P\{C_r = 0\} \\ &= (P\{C_\ell\})^r \\ &= \left(\frac{1}{r^n} \sum_{j=0}^{n} \binom{n}{j} \frac{(r-1)^{n-j} t!}{(t-j)!\, t^j} \right)^r \end{aligned} \tag{4}$$

since it is assumed that the primary hash function distributes states randomly. The probability q of omitting at least one state is of course simply $q = 1 - p$.

An experiment was conducted to compare the values of p computed from Eq. (4) against values obtained from a simulation. Using a small hash table of $r = 128$ rows and $b = 10$ bit keys, experiments were performed with $n = 50, 100, 150, \ldots, 500$ states. Each experiment was repeated 10000 times and the proportion of runs where clashes occurred was noted. The analytical and simulated results (with 95% confidence intervals) are presented in Fig. 7.

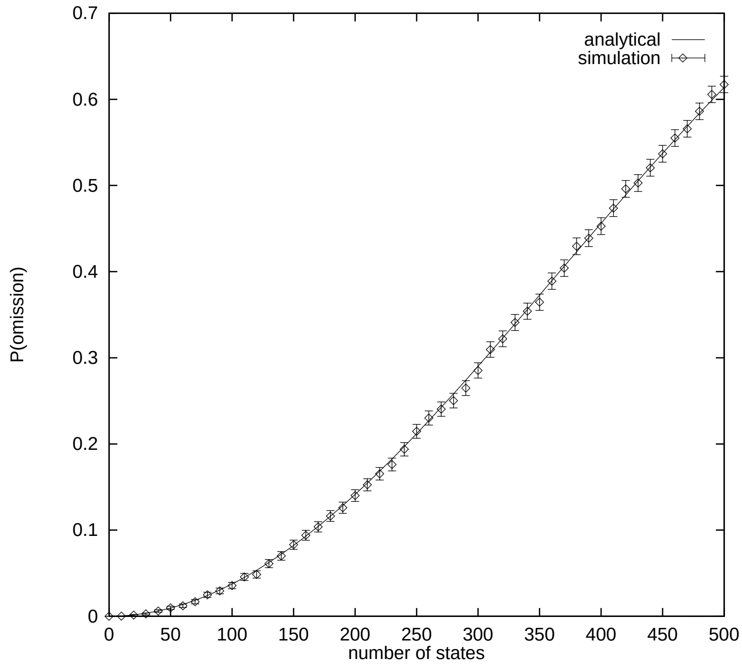


Figure 7: Analytical vs. simulated results for the probability of state omission q

3.3.2 Approximation

Evaluating the right hand side of Eq. (4) involves $O(n^2)$ operations. However, an $O(1)$ approximation can be found through an approach similar to that used by Stern and Dill [SD95] in their analysis of improved hash compaction.

We consider inserting n states into the hash table, one at a time. Let N_k be the event that no omission takes place when the $(k+1)$st state is inserted into the hash table, given that the previous k states have been inserted without any omissions.

The probability of event N_k will depend on the number of secondary key comparisons that have to be made when inserting state s_{k+1} into the target row given by it's primary hash key $h_1(s_{k+1})$. If there are j items in the target row,

$$P\{N_k|X^k_{h_1(s_{k+1})} = j\} = 1 - \frac{j}{t} \approx (1 - \frac{1}{t})^j$$

since $(1+nx) \approx (1+x)^n$ for $|x| << 1$ (here $1/t$ is very small). We have already established that

$$P\{X^k_{h_1(s_{k+1})} = j\} = \left(\begin{array}{c} n \\ j \end{array}\right) (\frac{1}{r})^j (1 - \frac{1}{r})^{n-j}$$

Thus by the law of total probability,

$$\begin{aligned}
P\{N_k\} &= \sum_{j=0}^{k} P\{N_k|X_{h_1(s_{k+1})} = j\} P\{X_{h_1(s_{k+1})} = j\} \\
&\approx \sum_{j=0}^{k} (1 - \frac{1}{t})^j \left(\begin{array}{c} k \\ j \end{array}\right) (\frac{1}{r})^j (1 - \frac{1}{r})^{k-j} \\
&= \sum_{j=0}^{k} \left(\begin{array}{c} k \\ j \end{array}\right) \left(\frac{1}{r}(1 - \frac{1}{t})\right)^j (1 - \frac{1}{r})^{k-j}
\end{aligned}$$

Applying the binomial theorem yields:

$$\begin{aligned}
P\{N_k\} &= \left(\frac{1}{r}(1 - \frac{1}{t}) + 1 - \frac{1}{r}\right)^k \\
&= \left(1 - \frac{1}{rt}\right)^k \\
&\approx e^{-\frac{k}{rt}} \qquad (5)
\end{aligned}$$

Now let N'_k be the unconditional event that no omission takes place when inserting the $(k+1)$st state into the hash table. Stern and Dill show that the probability p of no omission when inserting all n states is given by:

$$\begin{aligned} p &= P\{N'_{n-1} \wedge N'_{n-2} \wedge \ldots \wedge N'_0\} \\ &= P\{N'_{n-1} | N'_{n-2} \wedge \ldots \wedge N'_0\} P\{N'_{n-2} \wedge \ldots \wedge N'_0\} \\ &= P\{N_{n-1}\} P\{N'_{n-2} \wedge \ldots \wedge N'_0\} \end{aligned}$$

which, when applied recursively, yields:

$$p = \prod_{k=0}^{n-1} P\{N_k\}$$

Now we can substitute the expression for $P\{N_k\}$ from Eq. (5) to yield:

$$\begin{aligned} p &\approx \prod_{k=0}^{n-1} e^{-\frac{k}{rt}} \\ &= e^{-\sum_{k=0}^{n-1} \frac{k}{rt}} \\ &= e^{-\frac{n(n-1)}{2rt}} \end{aligned} \tag{6}$$

If $n(n-1) << 2rt$ (as will be the case in practical schemes where $q << 1$), we can use the fact that $e^x \approx (1+x)$ for $|x| << 1$ to approximate p by:

$$p = 1 - \frac{n(n-1)}{2rt}$$

so that probability q of an omission is:

$$q \approx \frac{n(n-1)}{2rt} = \frac{n(n-1)}{r2^{b+1}} \tag{7}$$

Thus the probability q of omitting a state is $O(n^2)$ and is inversely proportional to the hash table size r. Increasing the size of the compressed bit vectors b by one bit approximately halves the omission probability.

3.3.3 Space complexity

If we assume that the hash table rows are implemented as dynamic arrays, the number of bytes of memory required by the new scheme is:

$$M = hr + nb/8. \tag{8}$$

Here h is the number of bytes of overhead per hash table row. For a given number of states and a desired omission probability, there are a number of choices for r and b which all lead to schemes having different memory requirements. How can we choose r and b to minimize the amount of memory required? Rewriting Eq. (7):

$$r \approx \frac{n(n-1)}{2^{b+1}q} \tag{9}$$

and substituting this into Eq. (8) yields

$$M \approx \frac{hn(n-1)}{2^{b+1}q} + \frac{nb}{8}$$

Minimizing M with respect to b gives:

$$\frac{\partial M}{\partial b} \approx -\frac{n(n-1)(\ln 2)h}{2^{b+1}q} + n/8 = 0$$

Solving for b yields:

$$b \approx \log_2\left(\frac{h(n-1)\ln 2}{q}\right) + 2 \tag{10}$$

As an example, consider designing a system for up to $n = 2\,000\,000$ states and a desired maximum omission probability $q = 0.01$. We will assume a dynamic array overhead of 8 bytes for each row of the hash table, i.e. $h = 8$. This corresponds to a straightforward implementation using one 32 bit word for the number of elements in the array and a 32 bit pointer to the start of the array. Solving equation (10) gives $b = 32$ and substituting b into equation (9) yields $r = 46\,566$ for a total memory consumption of about $M = 8.4$ MB. Fig. 8 shows the omission probabilities for such a hash scheme as calculated using the $O(1)$ approximation of equation (6).

Fig. 9 shows the amount of memory required for other choices of b and r and confirms that $b = 32$ bit with $r = 46\,566$ rows is the optimal configuration for $n = 2\,000\,000$ and $q = 0.01$.

Table 1 shows the the optimal memory requirements in megabytes (MB) and the corresponding values of b and r for state space sizes ranging from 10^5 to 10^8.

In practice, it is difficult to implement schemes when b does not correspond to whole number of bytes. Practical considerations also dictate that we should take into account the limited memory resources available on typical workstations. Tables 2 and 3 thus compare the number of states that can be stored using 4-byte and 5-byte compression for various memory sizes and omission probabilities q. The results show that 5-byte compression is better for

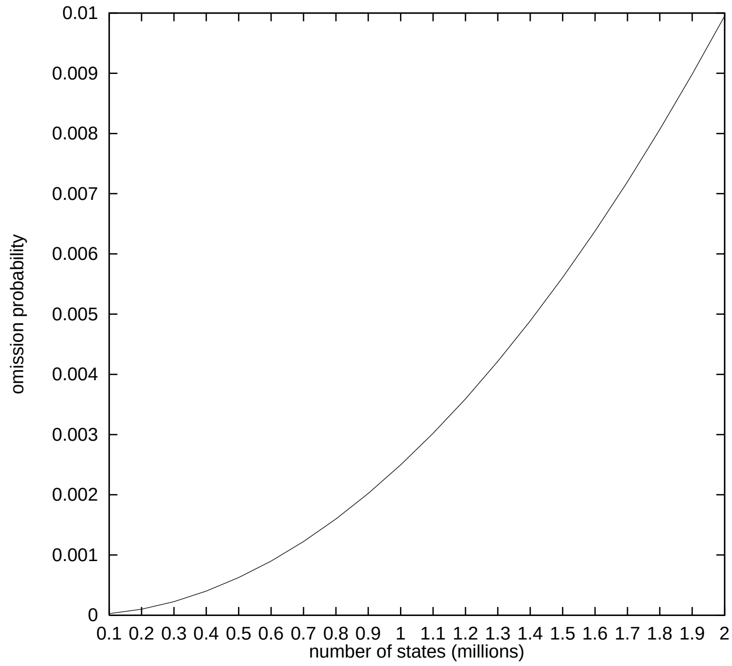


Figure 8: Omission probabilities for $b = 32$ and $r = 46\,566$

high reliability runs where $q = 0.001$, while 4-byte compression is better for low reliability runs where $q = 0.1$. For the intermediate case of $q = 0.01$, 4-byte compression performs better when there is less than 64MB memory available, while 5-byte compression is better when 64MB or more is available.

q	number of states											
	10^5			10^6			10^7			10^8		
	MB	b	r	MB	b	r	MB	b	r	MB	b	r
0.001	0.4061	31	2328	4.483	34	29104	48.96	38	1818997	530.7	41	2273737
0.01	0.3649	28	1863	4.061	31	23283	44.83	34	291038	489.6	38	1818989
0.1	0.3238	24	2980	3.649	28	18626	40.61	32	116415	448.3	34	2910383

Table 1: Optimal values for memory usage and the values for b and r used to obtain them for various system state sizes and omission probabilities q

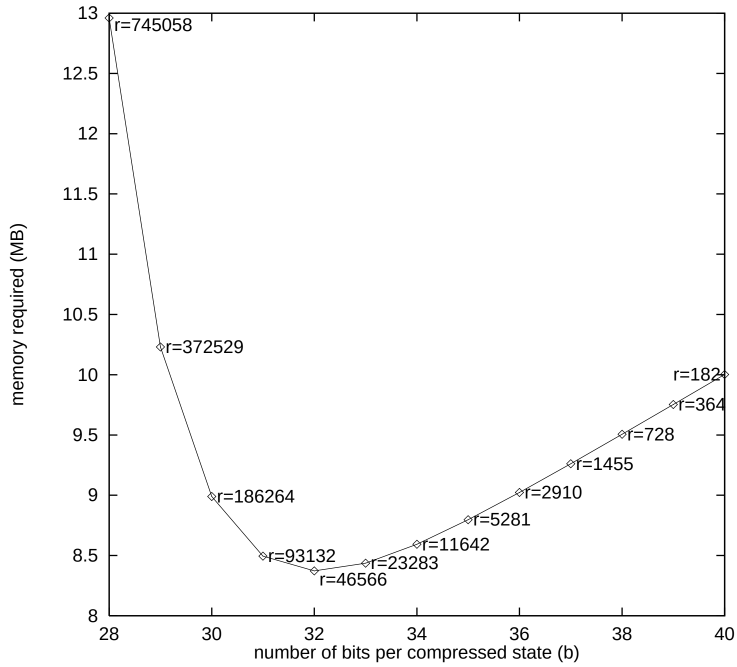


Figure 9: Memory required for various values of b and r for $n = 2\,000\,000$ states and $q = 0.01$

q	Available memory (megabytes)					
	8		16		32	
	n	r	n	r	n	r
0.001	1.49×10^6	257039	2.52×10^6	739670	4.1×10^6	1952386
0.01	1.91×10^6	42676	3.68×10^6	157997	6.9×10^6	553219
0.1	1.99×10^6	4614	3.96×10^6	18287	7.9×10^6	71853
q	64		128		256	
	n	r	n	r	n	r
0.001	6.41×10^6	4792060	9.77×10^6	11114490	1.46×10^7	24714769
0.01	1.24×10^7	1793662	2.14×10^7	5315710	3.52×10^7	14409217
0.1	1.54×10^7	277693	2.99×10^7	1041894	5.65×10^7	3723296

Table 2: Number of states that can be stored and optimal number of hash rows for 4-byte compression given various memory sizes and omission probabilities q

q	Available memory (megabytes)					
	8		16		32	
	n	r	n	r	n	r
0.0001	1.58×10^6	11378	3.13×10^6	44516	6.13×10^6	170706
0.001	1.60×10^6	1161	3.20×10^6	4635	6.40×10^6	18455
0.01	1.60×10^6	116	3.20×10^6	465	6.40×10^6	1860
0.1	1.60×10^6	12	3.20×10^6	47	6.40×10^6	186
q	64		128		256	
	n	r	n	r	n	r
0.0001	1.18×10^7	631990	2.21×10^7	2212877	3.97×10^7	7174652
0.001	1.27×10^7	73149	2.51×10^7	287412	4.94×10^7	1110770
0.01	1.28×10^7	7437	2.56×10^7	29691	5.10×10^7	118329
0.1	1.28×10^7	745	2.56×10^7	2979	5.11×10^7	11912

Table 3: Number of states that can be stored and optimal number of hash rows for 5-byte compression given various memory sizes and omission probabilities q

3.4 Vanishing state elimination

In this section we consider an "on-the-fly" technique for reducing the number of states that are stored in the explored-state table. Reducing the number of states during the state space exploration phase reduces both the memory required to store the state space as well as the effort required to solve for the steady state of the underlying Markov chain.

There are two types of transitions in timed transition systems: *timed* transitions which fire with an exponential delay and *instantaneous* transitions which, as their name implies, take no time to fire. If one or more instantaneous transitions are enabled in a given state, no time will be spent in that state and the state is *vanishing*. We will let V denote the set of vanishing states in a system. If, on the other hand, one or more timed transitions but no instantaneous transitions are enabled in a state, time will be spent in the state and the state is *tangible*. We will let T denote the set of tangible states in a system.

A crucial step in performance analysis is determining what proportion of time is spent in each of the system's reachable states. Given n states, this involves solving the set of steady state equations

$$\pi Q = 0 \quad \text{subject to} \quad \sum_{k=0}^{n} \pi_k = 1$$

where $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ is the n-vector of steady-state probabilities and Q is the $n \times n$

infinitesimal generator matrix of transition rates between states. For every $s \in V$, the steady state probability is by definition zero; the only function vanishing states serve is to help define the way the tangible states relate to one another. In fact, vanishing states are usually eliminated before solving the steady state equations using a two-step process [BDMC+94, §8.5.1]. Given $n_v = |V|$ vanishing states and $n_t = |T|$ tangible states, the generator matrix Q is first partitioned into the form:

$$Q = \begin{pmatrix} Q_t & Q_{t,v} \\ P_{v,t} & P_v \end{pmatrix}$$

where Q_t is the $n_t \times n_t$ matrix of transition rates between tangible states, $Q_{t,v}$ is the $n_t \times n_v$ matrix of transition rates from tangible to vanishing states, P_v is the $n_v \times n_v$ matrix of transition probabilities between vanishing states and $P_{v,t}$ is the $n_v \times n_t$ matrix of transition probabilities between vanishing and tangible states. Then the $n_t \times n_t$ matrix Q' representing the transitions between tangible states once vanishing states have been eliminated is given by:

$$Q' = Q_t + Q_{t,v} N P_{v,t} \tag{11}$$

where

$$N = \sum_{n=0}^{\infty} (P_v)^n = (I - P_v)^{-1}$$

The matrix $Q_{t,v} N P_{v,t}$ represents the effective rates of transition firing sequences which start and end at tangible states but which pass through one or more vanishing states. Calculating N is usually computed using LU-decomposition, which is an operation of $O(n_v^3)$ complexity.

This method reduces the size of the generator matrix from $n \times n$ to $n_t \times n_t$, thus decreasing the effort needed to solve the steady state equations. However, the method can only be applied once Q has been generated, i.e. at a point where it is too late to save memory during the state generation process. It also involves a time-consuming inversion operation.

Instead, it is possible to calculate the matrix of transition rates between tangible states Q' directly *during state space generation* using a process known as "on-the-fly" elimination of vanishing states.

Fig. 10 illustrates the principle of the algorithm. On the left is part of a reachability graph constructed using a simple DFS algorithm. States 1, 5, 6, 7 and 8 are tangible and states 2, 3 and 4 are vanishing. We wish to modify the DFS algorithm so that it now constructs

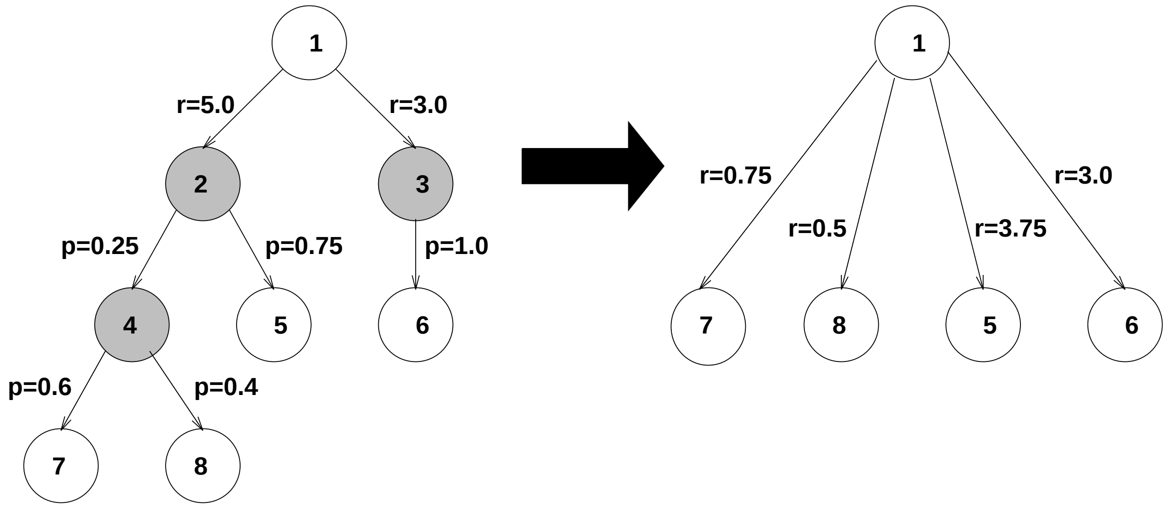


Figure 10: A simple example of vanishing state elimination

the tangible reachability graph shown on the right, where all vanishing states have been eliminated and the effective transition rates between tangible states have been calculated.

Fig. 11 presents the modified algorithm. The idea is to use two stacks during state space generation: one stack S_T is used to store unexplored tangible states in a usual DFS fashion, while another stack S_V is used as temporary storage for information about those vanishing states which are currently being eliminated. This information takes the form of a record $< s, r >$ where s is the state in question and r is the rate of entry into the state.

The algorithm begins by initialising S_T with the initial *tangible* states of the system, given some (possibly vanishing) initial state s_i.

Then, as each tangible state s is popped off S_T, unexplored tangible successors of s are pushed back onto S_T, while any vanishing successors of s are pushed onto S_V. Any state pushed onto S_V is immediately explored in a DFS fashion; vanishing sucessors are pushed back onto S_V while unexplored tangible successors are pushed onto S_T. This continues until S_V is empty and all the tangible successors of s have been established.

Rates between tangible transitions are determined by using the information stored on S_V together with the functions $\text{prob}(s, s')$ and $\text{rate}(s, s')$ to propagate transition probabilities through clusters of vanishing states. $\text{prob}(s, s')$ gives the probability of a transition from state $s \in V$ to state s', while the function $\text{rate}(s, s')$ gives the rate of a transition from $s \in T$ to state s'.

```
/* initialise S_T with initial tangible states */
if s_i ∈ T do begin
    push(S_T, s_i)
    E = {s_i}
end else do begin
    push(S_V, < s_i, 1.0 >)
    while (S_V not empty) do begin
        pop(S_V, < v, p >)
        for each v' ∈ succ(v) do begin
            if v' ∈ T do begin
                if v' ∉ E do begin
                    push(S_T, v')
                    E = E ∪ {v'}
                end
            end else do begin
                p' = p * prob(v, v')
                if p' > ϵ push(S_V, < v', p' >)
            end
        end
    end
end

/* perform state space exploration, eliminating vanishing states */
while (S_T not empty) do begin
    pop(S_T, s)
    for each s' ∈ succ(s) do begin
        if s' ∈ T do begin
            transition(s, s', rate(s, s'))
            if s' ∉ E do begin
                push(S_T, s')
                E = E ∪ {s'}
            end
        end else do begin
            push(S_V, < s', rate(s, s') >)
            while (S_V not empty) do begin
                pop(S_V, < v, p >)
                for each v' ∈ succ(v) do begin
                    p' = p * prob(v, v')
                    if v' ∈ T do begin
                        if v' ∉ E do begin
                            push(S_T, v')
                            E = E ∪ {v'}
                        end
                        transition(s, v', p')
                    end else if p' > ϵ push(S_V, < v', p' >)
                end
            end
        end
    end
    store s and tangible successors of s on secondary storage for later use
end
```

Figure 11: State space exploration algorithm with on-the-fly vanishing state elimination

Since our aim is to explore the state space for the purposes of performance analysis, reachability graph information has been explicity incorporated into the algorithm, through the use of the transition function. A call to transition(s, s', r) denotes that there is a transition firing sequence from $s \in T$ to $s' \in T$ with an effective transition firing rate of r. This reachability graph information need not be stored in memory but can be written out state by state to secondary storage for later use.

Cycles of vanishing states pose an interesting problem. Since we do not have an explored-state table for vanishing states, some method is necessary to preventing an infinite loop while exploring the states on S_V. Considering the reachability graph in Fig. 12 for example, vanishing states 2, 4 and 5 form a cycle; this cycle has the potential to cause an infinite loop when the states on S_V are explored in DFS fashion.

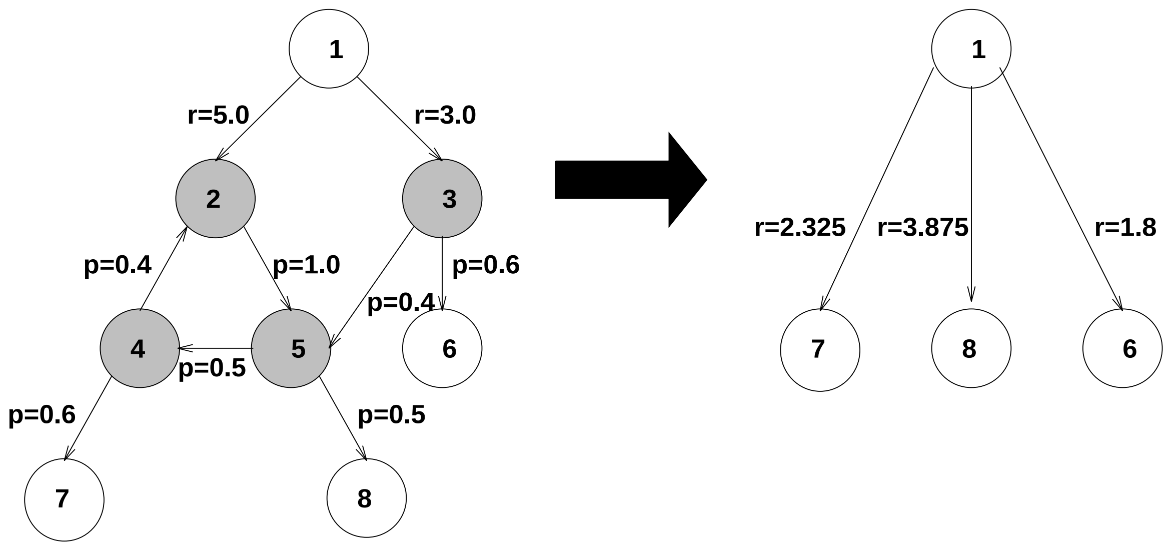


Figure 12: A more complicated example of vanishing state elimination involving a cycle

There are two solutions to this problem. One is to have a local explored-state table for clusters of vanishing states so that cycles can be recognised. Once cycles have been identified, Eq. (11) can then be applied locally to each cluster. Applying this strategy to the example, we consider eliminating the vanishing states 2, 3, 4 and 5 to determine the effective transition firing rates between tangible state 1 and it's tangible successor states 6, 7 and 8.

We have

$$Q_t = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \qquad Q_{t,v} = \begin{pmatrix} 5 & 3 & 0 & 0 \end{pmatrix}$$

$$P_{v,t} = \begin{pmatrix} 0 & 0 & 0 \\ 0.6 & 0 & 0 \\ 0 & 0.6 & 0 \\ 0 & 0 & 0.5 \end{pmatrix} \quad N = (I - P_v)^{-1} = \begin{pmatrix} 1.25 & 0 & 0.625 & 1.25 \\ 0.1 & 1 & 0.25 & 0.5 \\ 0.5 & 0 & 1.25 & 0.5 \\ 0.25 & 0 & 0.625 & 1.25 \end{pmatrix}$$

so that

$$Q' = Q_t + Q_{t,v}(I - P_v)^{-1} P_{v,t} = \begin{pmatrix} 1.8 & 2.325 & 3.875 \end{pmatrix}$$

The other way of dealing with cycles is simply to drop vanishing states whose propagated effective entry rate p' falls below a certain threshold value ϵ. This has been implemented in the algorithm shown in Fig. 11. This method is simpler than the first since it does require a local explored-state table for vanishing states, nor does it involve matrix inversion. It works by approximating the matrix

$$N = (I - P_v)^{-1} = \sum_{n=0}^{\infty} (P_v)^n$$

by computing N as

$$N = \sum_{n=0}^{k} (P_v)^n$$

for some large value of k. This works since it can be shown that

$$(P_v)^n \to 0 \quad \text{as} \quad n \to \infty$$

for any substochastic P_v with at least one row sum < 1. Goodman [Goo88, pg. 158–160] gives a proof for this result (*albeit* in the context of the long-term behaviour of transient states in an absorbing Markov Chain).

Fig. 13 shows a cycle of vanishing states that cannot be eliminated using either of the two methods we have considered. The first method of vanishing state elimination, i.e. applying Q' locally, will fail since $(I - P_v)^{-1}$ does not exist, while the second technique, i.e. dropping vanishing states when the propagated entry rate p' falls below ϵ, will enter an infinite loop since the propagated probability in the cycle never falls below 1. Final strongly connected components of vanishing states such as that formed by states 2, 4 and 5 are known as

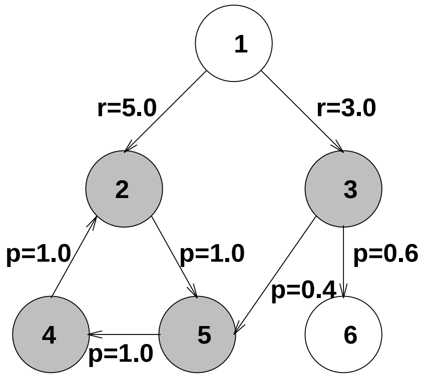


Figure 13: A timeless trap involving states 2, 4 and 5

timeless traps. Timeless traps correspond to functional errors in the Markov chain and render performance analysis impossible. To detect timeless traps, it is better to use the first elimination method since timeless traps will manifest themselves as errors during the calculation of $(I - P_v)^{-1}$, whereas the second technique will enter an infinite loop unless heuristic techniques are used. An example of such a heuristic technique is an "elimination timeout" which expires after a large number of attempts has been made to eliminate a cluster of states on S_V.

Chapter 4

Steady State Solution Techniques

4.1 Introduction

Once the state space of a system has been generated, the next stage in the performance analysis sequence is to establish what proportion of time the system spends in each of its states. This phase is by far and away the most resource-intensive phase in the performance analysis sequence, both with regard to the amount of memory used and the computational power required.

As described in Chapter 2, finding the steady state distribution $\pi = (\pi_1, \pi_2, ...\pi_n)$ of an irreducible finite *discrete time* Markov Chain (DTMC) with discrete-valued state space $S = \{1, 2, ...n\}$ involves solving a set of equations of form:

$$\pi = \pi P, \quad \sum \pi_i = 1 \tag{12}$$

where P is the $n \times n$ one-step transition probability matrix and π is the chain's stationary probability distribution vector of order n. The corresponding formula for a *continuous time* Markov Chain (CTMC) is:

$$\pi Q = 0, \quad \sum \pi_i = 1 \tag{13}$$

where Q is the infinitesimal generator matrix. Note that systems (12) and (13) are related since (under certain regularity conditions) a CTMC may be transformed into its associated embedded DTMC [SMC90].

In both cases above, we wish to solve for π. Several standard algorithms for solving linear systems of form $Ax = b$ exist. We may make use of these algorithms by rewriting Eq. (12) as:

$$(I - P^T)\pi^T = 0$$

and Eq. (13) as:

$$-Q^T\pi^T = 0$$

Both formulations result in a singular homogeneous system of linear equations of form:

$$Ax = 0 \tag{14}$$

where A is a (possibly very large) real and unsymmetric sparse $n \times n$ matrix with the following properties [Bar89]:

- $a_{ij} \leq 0$ for $i, j = 1, 2, ..., n$ and $i \neq j$.
- $\sum_{i=1}^{n} a_{ij} = 0$ for $j = 1, 2, ..., n$ (i.e. A has zero column sums).
- A is irreducible.
- $a_{ii} = 1$. This can be stated without loss of generality since Ax can be transformed to By where $B = AD^{-1}$ and $y = Dx$ with $D = \text{diag}(a_{11}, a_{22}, ...a_{nn})$. Once we have solved for y, x is easily obtained as $x = D^{-1}y$.

There are two general classes of methods for solving linear systems of form $Ax = b$: *direct methods* compute an exact solution in a fixed number of arithmetic operations determined by the size of the problem (direct methods are generally $O(n^3)$), while *iterative methods* form a sequence of vectors $x^{(0)}$, $x^{(1)}$, $x^{(2)}, \ldots$ which converges to the solution of $Ax = b$.

Iterative methods converge at a unknown rate, i.e. the number of operations to obtain a given accuracy is unknown, but they have several advantages over direct methods when dealing with large systems [Ste94, pg. 61–62]:

- Unlike direct methods, iterative methods **do not modify the matrix** A. This simplifies the sparse storage scheme, avoids matrix fill-in and prevents accumulation of round-off error in elements of A.

- Iterative methods generally **only involve matrix-vector operations of the form Ax or A^Tx**. These matrix vector products can be calculated efficiently using a good sparse matrix storage scheme.
- Iterative methods allow the user to **control the accuracy** of the solution. Direct methods always compute the solution to full machine precision, which may be unnecessary.
- Iterative methods are ideal for conducting a **sequence of experiments** where the parameters from one run to the next vary only slightly. In this case, the result vector from the previous run may be used as a good starting vector for the next run.

Direct methods, however, are still useful for smaller state spaces or for those ill-conditioned matrices where iterative methods take a very large number of iterations to converge.

We distinguish between three main classes of iterative methods for solving large sparse nonsymmetric linear systems:

- **Classical iterative methods** such as Gauss-Seidel and SOR [Var62, §3.1]. These techniques have been known for decades and are characterised by low memory requirements and smooth convergence. However, convergence is often slow, and the methods cannot be easily parallelised. Moreover, the SOR technique requires estimation of the over-relaxation parameter.
- **Krylov subspace techniques** such as Biconjugate Gradient (BiCG) [Fle76], Biconjugate Gradient Stabilised (BiCGSTAB) [Vor92] and Conjugate Gradient Squared (CGS) [Son89]. These algorithms are based on the original Conjugate Gradient method [HS52] which is a popular solution technique for linear systems involving symmetric positive definite coefficient matrices. These methods are parameter free and provide rapid, if somewhat erratic, convergence. The methods are easily parallelised.
- **Decomposition-based techniques** such as the Aggregation-Isolation (AI) algorithm and the Aggregation-Isolation Relaxed (AIR) [Tou95] algorithm. These algorithms are based on divide-and-conquer techniques and are characterised by low memory requirements and rapid smooth convergence behaviour.

We will now consider the various direct and iterative methods available for solving systems of form $Ax = 0$.

4.2 Direct methods

4.2.1 Gaussian Elimination

Gaussian elimination [EWK90, §8.3] [GL89, §3.2] [Ste94, §2.2.1] is probably the most well-known technique for solving the system $Ax = b$ in the case where A is nonsingular. By adding multiples of rows to each other, it systematically reduces the system $[A \mid b]$ to produce an equivalent upper triangular system $[U \mid c]$. This upper triangular system can then be solved by *back substitution.*

Reducing $[A \mid b]$ to $[U \mid c]$ involves $n-1$ steps. At the kth reduction step, the elements in the kth row which lie below the pivot element a_{kk} are eliminated by subtracting multiples of the kth row (the pivot row) from every row below the kth row. After the kth step, the system will have been reduced to:

$$\left(\begin{array}{cccccc|c}
a_{11} & a_{12} & & & \cdots & a_{1n} & b_1 \\
& a_{22} & & & \cdots & a_{2n} & b_2 \\
& & \ddots & & & \vdots & \vdots \\
& & & a_{kk} & a_{k,k+1} \cdots & a_{kn} & b_k \\
& & & \vdots & \vdots & \vdots & \vdots \\
& & & 0 & a'_{i,k+1} \cdots & a'_{in} & b'_i \\
& & & \vdots & \vdots & \vdots & \vdots \\
& & & 0 & a'_{n,k+1} \cdots & a'_{nn} & b'_n
\end{array}\right)$$

where the modified elements are given by:

$$\begin{array}{rcll}
a'_{ij} & = & a_{ij} - m_{ik}a_{kj} & j = k+1, k, \ldots, n \quad i = k+1, k, \ldots, n \\
b'_i & = & b_i - m_{ik}b_k & i = k+1, k, \ldots, n
\end{array}$$

with

$$m_{ik} = \frac{a_{ik}}{a_{kk}} \quad i = k+1, k, \ldots, n.$$

After $n-1$ reduction steps, the original system will have been reduced to the upper triangular system $[U \mid c]$. A simple back substitution process can then be used to find the solution vector x as follows:

$$x_n = c_n / u_{nn}$$

$$x_i = (c_i - \sum_{j=i+1}^{n} u_{ij} x_j)/u_{ii} \quad i = n-1, n-2, \ldots, 1$$

This standard Gaussian elimination cannot, however, be applied to our matrix A directly since A is singular. To handle the singularity, there are two main approaches [Ste94, pg. 73–74]:

- The "replace an equation" approach. Here, an equation (usually the last) is replaced by $\sum_i x_i = 1$ which removes the singularity. Even though any equation could be replaced, the last is chosen because this reduces fill-in and the operation count (for example, replacing the first row leads to massive fill-in since multiples of the first row are added to every other row).

  This approach is generally not used in computer implementations, however, since any accumulated rounding error in the calculation of x_n will be propagated through the back-substitution phase.

- The "remove an equation" approach. Since A is singular of rank $(n-1)$, one of the equations is redundant and may be removed; this yields $n-1$ equations in n unknowns. If we now set $x_n = 1$ and solve the remaining non-singular system of order $n-1$ for its probability vector $\hat{x}$, the final probability vector is given by normalising $(\hat{x}, 1)$.

To prevent loss of accuracy, a system of row interchanges called partial pivoting is usually applied when solving general systems of equations [EWK90]. At the kth step, the kth column from the pivot element downwards is searched for the element of the largest modulus; i.e. we determine row r such that:

$$|a_{rk}| = \max_{i=k,k+1,\ldots,n} |a_{ik}|$$

Rows r and k are then interchanged. However, pivoting is unnecessary in our case since an error analysis of Gaussian elimination as applied to irreducible Markov Chains [Ste94, §2.7.4] shows that Gaussian elimination without pivoting is already a stable way to calculate the stationary probability vector of the irreducible Q-Matrix A (once the singularity has been removed as outlined above, of course).

The algorithm for solving the singular system $Ax = 0$ by Gaussian elimination is:

GAUSSIAN ELIMINATION

1. Reduce A to upper triangular form
   - **for** $k = 1$ **to** $n-1$ **do**

     for $i = k+1$ **to** n **do** $m_{ik} = a_{ik}/a_{kk}$

     for $i = k+1$ **to** n **do**

     for $j = k+1$ **to** n **do** $a_{ij} = a_{ij} - m_{ik} * a_{kj}$
2. Back substitute
   - $sum = 1,\ x_n = 1$
   - **for** $i = n-1, n-2, \ldots, 1$ **do**
   - $x_i = -\left(\sum_{j=i+1}^{n} a_{ij} x_j\right) / a_{ii}$
   - $sum = sum +\ x_i$
3. Normalize
   - **for** $i = 1$ **to** n **do** $x_i = x_i/sum$

4.2.2 *LU* Decomposition

LU Decomposition [EWK90, §8.6] [GW89, §2.5], [GL89, §3.2.5] [Ste94, §2.2.2] is the process of factorizing a matrix A into the product of a lower diagonal matrix L an upper triangular matrix U, i.e.

$$A = LU$$

where it is usual to assume that one of L and U are unit diagonal matrices i.e. either

$$l_{ii} = 1 \quad \text{for} \quad i = 1, 2, \ldots, n$$

in which case the process is known as Doolittle decomposition, or

$$u_{ii} = 1 \quad \text{for} \quad i = 1, 2, \ldots, n$$

in which case the process is known as Crout decomposition.

LU Decomposition is closely related to Gaussian elimination; in fact, Gaussian elimination can be used to find an *LU* factorization of A, where L is given by a unit lower diagonal matrix of multipliers and U is the upper diagonal matrix at the end of the reduction phase.

The general formulas for the elements of L and U (Crout Reduction) are:

$$\begin{array}{lcllll} l_{ij} & = & a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj} & j \leq i & i = 1, 2, \ldots, n \\ u_{ij} & = & a_{ij} - (a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{jk})/l_{ii} & i \leq j & j = 2, 3, \ldots, n \end{array} \tag{15}$$

An algorithm to perform a memory-efficient in-place LU factorisation of an arbitrary matrix A may be found in [BDMC+94, pg. 31].

Once we have a *LU* decomposition of A, we can quickly solve for x by forward and backward subsitution (an $O(n^2)$ process):

1. Solve $Ly = b$ for y.
2. Solve $Ux = y$ for x.

It can be shown that an *LU* decomposition exists for any matrix A derived from an irreducible Markov Chain; further, for such matrices, no pivoting is necessary to ensure stability [Ste94, pg. 66].

In the context of solving Markov Chains, full *LU* Decomposition has two important advantages over standard Gaussian elimination:

- The inner products of Eq. (15) may be accumulated in double precision [GW89, pg. 109–110]; this yields improved accuracy.
- Storing the information about L allows for the application of *iterative refinement* [EWK90, §8.12]. This is a technique used to obtain maximum machine accuracy in the face of representation and rounding errors introduced by a floating point system. If we have an inexact decomposition $\hat{L}\hat{U}$ with solution $\hat{x}$, we can improve the solution using the residual vector $r = b - A\hat{x}$. The procedure is:

  repeat until convergence {

  set $r = b - A\hat{x}$ (usually calculated in double precision)

solve $\hat{L}y = r$
solve $\hat{U}z = y$
set $x = \hat{x} + z$
}

Iterative refinement is computationally quite cheap, since each iteration is $O(n^2)$ (L and U are triangular) and usually only one iteration is required to obtain an answer to full machine precision.

4.2.3 Grassman's Algorithm

Grassmann's algorithm [GTH85] [KGB87] [BDMC$^+$94, pg. 32-34] [Ste94, §2.5] is a variant of Gaussian elimination which appears to be even more stable since the algorithm does not involve any subtraction operations (or negative numbers). This means that problems such as loss of significance and the accumulation of rounding errors are minimized.

The algorithm is based on two key ideas which take advantage of the special structure of the matrix A:

- The properties of A (i.e. $a_{ii} > 0$, $a_{ij} \leq 0$, and $\sum_{i=1}^{n} a_{ij} = 0$) are invariant under the row operations of Gauss elimination.
- Subtractions which could lead to loss of significance occur only during the calculation of diagonal pivot elements.

Now, since A always has zero column sums, diagonal pivot elements do not have to be calculated by subtraction. Instead, the off-diagonal elements in the column can be summed and the result negated; this produces a more accurate result at the cost of slightly more numerical operations. The full version of Grassmann's algorithm is given below:

GRASSMANN'S ALGORITHM [GTH85]

1. **for** $k = n, n-1, \ldots, 2$ **do**
   - $s = \sum_{j=1}^{k} a_{kj}$
   - $a_{in} = a_{ik}/s, \forall i < k$

- $a_{ij} = a_{ij} + a_{ik}a_{kj}, \forall i, j < k$

2. $sum = 1$, $x_1 = 1$
3. **for** $j = 2, 3, \ldots, N$ **do**
   - $x_j = \sum_{k=1}^{j} x_k a_{kj}$
   - $sum = sum + x_j$
4. $x_j = x_j / sum$, $j = 1, 2, \ldots, N$

Note that both column and row access to the matrix A is required. For smaller problems, a dense two-dimensional matrix representation suffices, but for larger problems, a sparse matrix scheme supporting column links is required.

Since Gaussian elimination is already stable, the extra time and space demands of Grassmann's algorithm mean that its extra precision is only really necessary when the problem is very ill-conditioned.

4.3 Classical iterative methods

In this section we consider some of the oldest and most well-known iterative methods for solving linear system of form $Ax = b$. These methods are based around matrix splittings of form $A = M - N$ where M is non-singular. This splitting is used to define simple iterative schemes of form:

$$x_{k+1} = M^{-1}N\, x_k + c$$

where neither the iteration matrix $M^{-1}N$ nor c depends on k.

In our case of solving $Ax = 0$, the schemes reduce to the form:

$$x_{k+1} = M^{-1}N\, x_k$$

From this equation, the desired solution can be seen to be the eigenvector of the iteration matrix $M^{-1}N$ corresponding to the eigenvalue 1. Thus the convergence of these methods depends on the eigenvalues of the iteration matrix $M^{-1}N$. In particular, the rate of convergence is inversely proportional to the ratio $|\lambda_2|/|\lambda_1|$ where λ_1 and λ_2 are the dominant and the subdominant eigenvalues of the iteration matrix respectively [Bar89]. Consequently, these methods are only guaranteed to converge if the iteration matrix is *primitive*, i.e. if it has one and only one eigenvalue λ_i with $|\lambda_i| = 1$.

The methods presented here have very modest memory requirements, using only two vectors ($x^{(k+1)}, x^{(k)}$) and requiring only row (but not column) access to the matrix A.

The three most commonly used classical iterative methods of this form are presented below.

4.3.1 Jacobi's Method

Jacobi's method [Var62, §3.1] [Ste94, §3.2.2] [HY81, §2.3] is a simple iterative method based on the observation that solving $Ax = b$ is equivalent to finding the solution to the n equations:

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad i = 1, 2, \ldots, n$$

Now, solving the ith equation for x_i yields:

$$x_i = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j)$$

which suggests the iterative method:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}) \tag{16}$$

where $k \geq 0$ and $x^{(0)}$ is an initial guess at the solution vector.

If we write $A = D - L - U$ where $D = \mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn})$ and L and U are strictly lower and upper triangular matrices respectively, Eq. (16) can be written in matrix form as:

$$x^{(k+1)} = D^{-1}(L + U)x^{(k)} + D^{-1}b$$

where $D^{-1}(L + U)$ is the iteration matrix characterising the convergence behaviour of the algorithm.

Note that the calculation of the $x_i^{(k)}$'s are independent of one another which means equation updates can be performed in parallel.

4.3.2 Gauss-Seidel

The Jacobi method can be improved on by using the computed results for x_i as soon as they are available within an iteration. The resulting method is known as the Gauss-Seidel method [Var62, §3.1], [Ste94, §3.2.3] which is given by:

$$x_i^{(k+1)} = (b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)})/a_{ii} \tag{17}$$

In matrix form, Eq. (17) can be written as:

$$x^{(k+1)} = (D - L)^{-1}(Ux^{(k)} + b)$$

where $(D - L)^{-1}$ is the iteration matrix characterising the convergence behaviour of the algorithm.

Note that the computations of Eq. (17) appear to be serial in nature since the calculations of the $x_i^{(k)}$'s now depend on one another. However, if A is sparse and several coefficients are zero, then elements of the new iterate are not necessarily dependent on previous elements. By reordering the equations in this situation, it is possible to make updates to groups of components in parallel [BBC$^+$94, §3, §4.4].

4.3.3 Successive Overrelaxation (SOR)

Successive Overrelaxation (SOR) [Var62, §3.1] [Ste94, §3.2.4] [BDMC$^+$94, §3.1.3] is an extrapolation technique for accelerating the convergence of the Gauss-Seidel algorithm. The extrapolation works by successively taking a weighted average of each element of the previous iterate and each element of the newly-computed Gauss-Seidel iterate, i.e.

$$x_i^{(k+1)} = \omega \overline{x}_i^{(k+1)} + (1 - \omega) x_i^{(k)} \tag{18}$$

where $\overline{x}_i^{(k+1)}$ is the ith element of the newly-computed Gauss-Seidel iterate and $x_i^{(k)}$ is ith element of the previous iterate.

In matrix form, Eq. (18) can be written as:

$$x^{(k+1)} = (D - \omega L)^{-1}(\omega U + (1 - \omega)D)x^{(k)} + \omega(D - \omega L)^{-1} b$$

with iteration matrix

$$L_\omega = (D - \omega L)^{-1}(\omega U + (1 - \omega)D)$$

Note that if $\omega = 1$, the method reduces to the Gauss-Seidel algorithm. In the case $\omega > 1$, we speak of over-relaxation and in the case $\omega < 1$, we speak of under-relaxation. The Gauss-Seidel method converges only for values of ω in the range $0 \leq \omega \leq 2$.

In the case of solving $Ax = 0$, the optimal value of ω is that value which maximizes the difference between the dominant and subdominant values of L_ω, thus resulting in the fastest covergence rate. Unfortunately, methods for choosing this optimal value of ω are only known for very restricted classes of matrices [HY81]. Consequently, implementations usually use heuristic adaptive parameter estimation schemes to try to home in on the appropriate value of ω by guessing a value which is adjusted every few iterations according to the rate at which the method is converging.

4.4 Krylov subspace techniques

Krylov subspace techniques [Wei95] [SW95] [FGN92] [Ste94, §4.3] are a popular class of iterative methods for solving large systems of linear equations. They derive their name from the fact that they generate their iterates using a shifted Krylov subspace associated with the coefficient matrix of the system. Many conjugate-gradient type algorithms and their variants fall into this category. Before defining a Krylov subspace formally, we will first provide an overview of the advantages of Krylov subspace techniques.

Krylov subspace techniques have proved useful for solving systems of linear equations arising from a wide range of scientific and engineering applications such as fluid dynamics, atmospheric modelling, structural analysis and finite element analysis. There are three main reasons for this widespread use of Krylov subspace techniques:

- The methods are parameter free, yet still provide good rates of convergence. The original conjugate gradient algorithm, for example, provides the same order of convergence rate as optimal SOR but without the need for dynamic parameter estimation.

- Krylov subspace techniques have become increasingly competitive with classical iterative methods in terms of memory utilization. This is because the most recently developed conjugate gradient-type algorithms for non-symmetric matrices (e.g. CGS, BiCGSTAB, TFQMR) do not require storage of large sequences of vectors (as does

GMRES), nor do they require multiplication with the transpose of the coefficient matrix (as do BiCG, QMR and CGNR/CGNE).

- The methods are well suited to implementation on parallel and vector computers. Most Krylov subspace methods compute one or two matrix-vector products and several vector inner products every iteration; the methods are thus easily parallelised by distributing the matrix among processing nodes and by using the inner products as synchronisation points. For a general discussion of the issues involved see [Saa89] and [GKS95] and for case studies see [ME90], [Bou95] and [Taf95]. In practice, superlinear speedups (corresponding to efficiencies of over 100%) have been achieved in both symmetric multiprocessing environments and high-speed distributed environments [Bou95]. This can probably be attributed to efficient cache utilization.

The development of Krylov subspace techniques began in the early 1950s with the conjugate gradient (CG) algorithm of Hestenes and Stiefel [HS52]. This algorithm is used to solve $n \times n$ linear systems of form $Ax = b$ where A is a symmetric positive definite (SPD) coefficient matrix. The CG method is regarded as an attractive algorithm for two main reasons. Firstly, the algorithm has very modest memory requirements because it uses simple three-term recurrences. Secondly, the algorithm has good convergence properties since the residual is minimized with respect to some norm at each step. The generated residuals are also mutually orthogonal, which guarantees finite termination.

Several algorithms have since been devised to generalise the CG algorithm to allow for arbitrary (i.e. not necessarily symmetric or positive definite) coefficient matrices. Unfortunately, algorithms for non-symmetric coefficient matrices cannot maintain both the short recurrence formulation and the minimization property (see Faber and Manteuffel's paper [FM84] for proof). Thus, by trading off certain optimality conditions against the amount of memory required, three main classes of CG variants have been developed:

- Algorithms which attempt to preserve both properties by transforming a linear system based on a non-symmetric coefficient matrix A to an equivalent system based on the symmetric positive definite matrix $A^T A$ (CGNR) or AA^T (CGNE). This approach is known as conjugate gradient applied to the normal equations.

- "Pure" algorithms for non-symmetric A which are based on maintaining either the short recurrence formulation (e.g. BiCG [Fle76]) or the minimization property (e.g. GMRES [SS86]) but not both.

- "Hybrid" methods for non-symmetric A which seek to combine elements of the short recurrence formulation with minimization properties that are either heuristic (e.g. CGS), localized (e.g. BiCGSTAB) or quasi-optimal (e.g. QMR). This class includes most of the more recently developed CG-type methods such as CGS [Son89], QMR [FN91], BiCGSTAB [Vor92], BiCGSTAB(l) [SF93], and TFQMR [Fre93]).

Many authors have attempted to resolve the confusion resulting from the development of all these methods by using unifying mathematical frameworks to explore the relationships between them (see e.g. [Wei95], [Wei94], [Gut93a] and [AMS90]). Fig. 4.4 presents a more conceptual overview of the most important techniques. The arrows show the relationships between the methods i.e. how the methods have been generalised from their underlying basis-generating algorithms and also how key concepts have been inherited from one algorithm to the next. Readers unfamiliar with Krylov subspace techniques might like to use figure 4.4 as a top-level reference throughout the next three sections.

A more rigorous mathematical framework and classification scheme is presented in the next section.

4.4.1 Principles of Krylov Subspace Techniques

We consider the linear system

$$Ax = b \tag{19}$$

where A is a real $n \times n$ matrix and $x, b \in \mathbb{R}^n$; in general A is non-symmetric and is not positive definite. Let x_0 be an initial guess at the solution vector x with corresponding residual $r_0 = b - Ax_0$. Then, using the notation of Weiss [Wei95], Krylov subspace techniques generate subsequent iterates x_k according to the formula:

$$x_k = x_{k-\sigma_k} + d_k, \quad d_k \in \mathrm{span}(q_{k-\sigma_k,k}, \ldots, q_{k-1,k}) \quad \text{for } k = 1, 2, \ldots \tag{20}$$

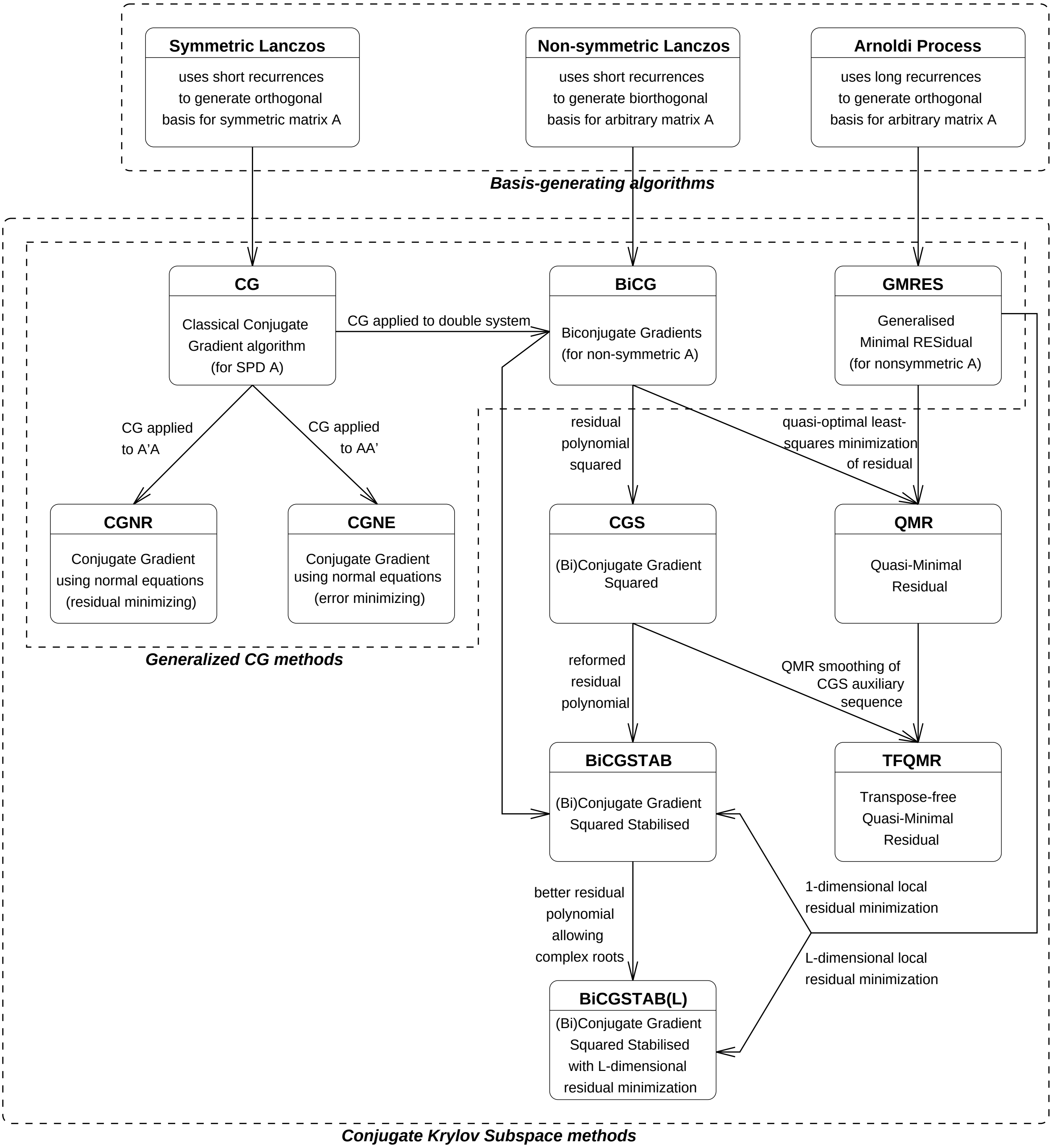


Figure 14: An overview of Krylov Subspace Techniques

where $q_{k-i,k} \in \mathbb{R}^n$ and σ_k denotes the number of previous q vectors used in the calculation of new iterates. Usually all previous q vectors are used i.e. $\sigma_k = k$, in which case we speak of an *exact* method. However, sometimes methods are restarted every σ_{res} steps to limit their memory consumption, i.e. $\sigma_k = (k-1) \bmod \sigma_{res} + 1$. In this case, we speak of a *restarted* method.

The q vectors are generated to fulfill two conditions:

- Firstly, each $q_{k-i,k}$ is a member of the *Krylov subspace*:

$$\mathcal{K}_{k-i+1}(B, z) \quad = \quad \mathrm{span}(z, Bz, B^2z, \ldots, B^{k-i}z) \tag{21}$$

  where B is an arbitrary $n \times n$ matrix and $z \in \mathbb{R}^n$. For almost all Krylov subspace methods of practical interest (and for all the methods discussed here), $B = A$ and $z = r_0$, i.e. each $q_{k-i,k}$ lies in the Krylov subspace

$$\mathcal{K}_{k-i+1}(A, r_0) \quad = \quad \mathrm{span}(A, Ar_0, A^2r_0, ..., A^{k-i}r_0) \tag{22}$$

  Equation (22) characterises a class of methods known as Conjugate Krylov Subspace (CKS) techniques. From the definition of $q_{k-i,k}$ and equation (20), it follows that

$$x_k \quad \in \quad x_{k-\sigma_k} + \mathcal{K}_k(A, r_0) \tag{23}$$

  i.e. the iterates lie in a shifted Krylov subspace associated with the coefficient matrix of the system.

- Secondly, the q vectors satisfy the orthogonality condition:

$$r_k^T Z_k q_{k-i,k} = 0 \quad \text{for } i = 1, \ldots, \sigma_k \tag{24}$$

  where the Z_k are auxiliary non-singular matrices. Methods characterised by constant Z_k, i.e.

$$Z_k = Z$$

  are known as generalised CG methods and correspond to methods derived from applying the CG algorithm to the normal equations and "pure" methods closely related to the original basis construction algorithms of Lanczos and Arnoldi.

Note that the choice of σ_k reflects the depth to which the subspace (21) is constructed and also the depth to which the orthogonality condition (24) is maintained.

4.4.2 Basis construction algorithms

Many Krylov methods involve the construction an orthogonal or biorthogonal basis for the Krylov subspace of equation (21). Algorithms for doing this have been known since the 1950s, and it is these algorithms which formed the foundation of modern Krylov subspace techniques. The three most important basis-generating techniques include the symmetric Lanczos algorithm, the non-symmetric Lanczos algorithm and Arnoldi's method.

Symmetric Lanczos

The symmetric Lanczos algorithm [Ste94, §4.5.1] was originally devised by Cornelius Lanczos as a means of determining the eigenvectors and eigenvalues of a symmetric $n \times n$ matrix A. At the kth iteration, the algorithm constructs an orthonormal basis $(v_1, v_2, \ldots, v_k)$ and an $k \times k$ symmetric tridiagonal matrix

$$T_k = \begin{pmatrix} \alpha_1 & \beta_2 & & & & \\ \beta_2 & \alpha_2 & \beta_3 & & & \\ & \beta_3 & \alpha_3 & \beta_4 & & \\ & & \ddots & & \ddots & \\ & & & \ddots & & \beta_k \\ & & & & \beta_k & \alpha_k \end{pmatrix}$$

such that $T_k = V_k^T A V_k$ where V_k is the $n \times k$ matrix with columns $v_1, v_2, \ldots, v_k$. T_k is constructed such its eigenvalues are approximations to a subset of the eigenvalues of A. Given the initial conditions $\|v_1\|_2 = 1$, $\beta_1 = 0$ and $v_0 = 0$, the orthonormal sequence v_k is computed using the short recurrence:

$$\begin{aligned} \tilde{v}_{k+1} &= Av_k - \alpha_k v_k - \beta_k v_{k-1} \\ v_{k+1} &= \frac{\tilde{v}_{k+1}}{\|\tilde{v}_{k+1}\|_2} \end{aligned}$$

where

$$\begin{aligned} \alpha_k &= v_k^T A v_k \\ \beta_k &= \|v_{k-1}\|_2 \end{aligned}$$

The basis $(v_1, v_2, \ldots, v_k)$ so generated spans the Krylov subspace generated by A and v_1 i.e.

$$\mathrm{span}(v_1, v_2, \ldots, v_k) \;=\; \mathcal{K}_k(v_1, A) \;=\; \mathrm{span}(v_1, Av_1, \ldots, A^{k-1}v_1)$$

In exact arithmetic, the algorithm terminates after n steps with the eigenvalues of T_n the same as those of A (assuming no breakdown). When finite precision arithmetic is used, however, the Lanczos algorithm has poor numerical properties. In particular, the orthogonality of the vectors v_k is often lost, which leads to inaccurate eigenvalue estimates.

There is a close relationship between symmetric Lanczos and classical Conjugate Gradient algorithm; in fact, the Conjugate Gradient algorithm may be derived from the Lanczos algorithm and vice versa [GL89, §9.3.1 and §10.2.6].

Non-symmetric Lanczos

The non-symmetric Lanczos algorithm [FGN92, §3.1] is a generalization of the symmetric Lanczos algorithm to non-symmetric A. At the kth iteration, the matrix A is reduced to a tridiagonal system

$$T_k = \begin{pmatrix} \alpha_1 & \beta_2 & & & & \\ \gamma_2 & \alpha_2 & \beta_3 & & & \\ & \gamma_3 & \alpha_3 & \beta_4 & & \\ & & \ddots & & \ddots & \\ & & & \ddots & & \beta_k \\ & & & & \gamma_k & \alpha_k \end{pmatrix}$$

with the eigenvalues of matrix A eventually given by those of the tridiagonal system T_n. However, since A is non-symmetric, it is now impossible to use a single short recurrence to generate an orthonormal basis for A. Instead, the non-symmetric Lanczos algorithm constructs a pair of vector sequences $v_1, v_2, \ldots, v_k$ and $w_1, w_2, \ldots, w_k$ such that

$$\mathrm{span}(v_1, v_2, \ldots, v_k) \;=\; \mathcal{K}_k(v_1, A) \tag{25}$$

$$\mathrm{span}(w_1, w_2, \ldots, w_k) \;=\; \mathcal{K}_k(w_1, A^T) \tag{26}$$

and such that the biorthogonality condition

$$w_i^T v_j = v_i^T w_j = 0 \quad \text{for } i \neq j \tag{27}$$

is satisfied. Given vectors $v_1, w_1 \in \mathbb{R}^n$ with $\|v_1\|_2 = 1$ and $\|w_1\|_2 = 1$, the sequences w_k and v_k can be calculated using simple three-term recurrences:

$$\begin{aligned}
\tilde{v}_{k+1} &= Av_k - \alpha_k v_k - \beta_k v_{k-1} \\
\tilde{w}_{k+1} &= A^T w_k - \alpha_k w_k - \gamma_k w_{k-1} \\
v_{k+1} &= \frac{\tilde{v}_{k+1}}{\|\tilde{v}_{k+1}\|_2} \\
w_{k+1} &= \frac{\tilde{w}_{k+1}}{\|\tilde{w}_{k+1}\|_2}
\end{aligned}$$

where

$$\begin{aligned}
\alpha_k &= \frac{w_k^T A v_k}{w_k^T v_k} \\
\beta_k &= \frac{w_{k-1}^T A v_k}{w_{k-1}^T v_{k-1}} \\
\gamma_k &= \frac{v_{k-1}^T A^T w_k}{v_{k-1}^T w_{k-1}}
\end{aligned}$$

Unfortunately, the calculation of the coefficients used in the construction of v_{k+1} and w_{k+1} involves division by $w_k^T v_k$, which may be zero or close to zero even if $w_k \neq 0$ and $v_k \neq 0$. Breakdowns of this type are known as *serious breakdowns* and can be avoided by using block look-ahead algorithms which relax the biorthogality condition [Nac91, §3].

Fletcher's Bi-Conjugate Gradient algorithm [Fle76] is a reformulation of the non-symmetric Lanczos algorithm. As we shall see, it suffers from possible breakdowns in its underlying Lanczos process.

Arnoldi's method

Arnoldi's method [Ste94, §4.4.1] is another generalization of the symmetric Lanczos method to non-symmetric matrices. However, instead of constructing a biorthonormal basis for A by using short recurrences, Arnoldi's method uses long recurrences to generate a single orthonormal basis $(v_1, v_2, \ldots, v_k)$ spanning the Krylov subspace generated by A and v_1 i.e.

$$\text{span}(v_1, v_2, \ldots, v_k) = \mathcal{K}_k(v_1, A)$$

Since short recurrences cannot be used, the matrix A is no longer reduced to a tridiagonal system, but one based on an upper Hessenberg matrix H_k (upper Hessenberg means that

$H_k(i,j) = 0$ for $i > j+1$). The core of Arnoldi's method which actually constructs the orthonormal basis is known as the Arnoldi process and is given by the following algorithm:

1. Initialise
   - Choose vector v_1 with $\|v_1\|_2 = 1$
2. Iterate
   - for $j = 1, 2, \ldots$, k

     for $i = 1, 2, \ldots, j$

     $h_{i,j} = {v_i}^T A v_j$

     $\hat{v}_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j} v_i$

     $h_{j+1,j} = \|\hat{v}_{j+1}\|_2$

     $v_{j+1} = \hat{v}_{j+1}/h_{j+1,j}$

Note that after k steps the algorithm has generated:

- an $n \times k$ orthonormal system V_k with columns $v_1, v_2, \ldots, v_k$.
- a $(k+1) \times k$ upper Hessenberg matrix $\bar{H}_k$. The first k rows of this matrix are given by $H_k = {V_k}^T A V_k$ and represent the matrix A in the basis $(v_1, v_2, \ldots, v_k)$; the last row has only one non-zero element which is $h_{j+1,j} = \|\hat{v}_{j+1}\|_2$.

The algorithm cannot break down, but is expensive because calculation of v_k at the kth iteration requires the use of vectors $v_1, v_2, \ldots, v_{k-1}$. The Arnoldi process was central to the development of the Generalised Minimum Residual (GMRES) algorithm.

4.4.3 Generalised CG Techniques

Classical Conjugate Gradient Algorithm

The classical conjugate gradient algorithm provides an efficient means of solving linear systems of form (19) when A is symmetric positive definite (SPD). The central idea is to minimize the function:

$$f(x_k) = \frac{1}{2} x_k^T A x_k - x^T b \tag{28}$$

which has a unique minimum (given SPD A) when its gradient

$$\frac{\partial f}{\partial x_k} = A x_k - b = -r_k$$

is zero, so the value of x_k minimizing equation (28) is also the solution to equation (19). To perform the function minimization, a sequence of search directions p_k are generated starting with $p_0 = r_0$; these are used to improve the iterates according to the recurrence:

$$x_{k+1} = x_k + \alpha_k p_k \tag{29}$$

$$r_{k+1} = r_k - \alpha_k A p_k \tag{30}$$

$$p_{k+1} = r_{k+1} + \beta_k p_k \tag{31}$$

where

$$\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$$

is chosen to minimize $f(x_{k+1})$ over the subspace $(p_0, p_1, \ldots, p_k)$ and

$$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$$

is chosen to update the p vectors such they are A-conjugate to one another, i.e. such that the conjugacy condition

$$p_k^T A p_j = 0 \quad \text{for } j < k \tag{32}$$

holds. Note that, since the p_k are non-zero and non-zero A-conjugate vectors are linearly independent, the algorithm should terminate in $m \leq n$ steps (given exact arithmetic).

Multiplying equation (29) on the left by $-A$ and adding b yields the update formula for the residuals given in equation (30). The residuals satisfy the orthogonality conditions:

$$r_k^T r_j = 0 \quad \text{and} \quad r_k^T p_j = 0 \quad \text{for } j < k \tag{33}$$

An inductive proof of the first orthogonality condition (i.e. $r_k^T r_j = 0$) is given in [GL89, §10.2.5]; the proof may also be derived by specialising a similar proof for the biconjugate gradient algorithm given [Fle76, §5]. The second orthogonality condition (i.e. $r_k^T p_j = 0$) follows by rewriting equation (31) as

$$p_k = r_k + \beta_{k-1} r_{k-1} + \beta_{k-1}\beta_{k-2} r_{k-2} + \ldots + (\beta_{k-1}\beta_{k-2}\ldots\beta_0) r_0 = \sum_{i=0}^{k} \gamma_i r_i \qquad (34)$$

Changing the index from k to j and multiplying on the left by r_k yields:

$$r_k^T p_j = \sum_{i=0}^{j} \gamma_i r_k^T r_i = 0 \qquad (35)$$

by the first orthogonality condition.

From equation (34), equation (29) can be rewritten as:

$$x_k = x_{k-1} + \alpha_{k-1} p_{k-1} = x_0 + \sum_{i=0}^{k-1} \gamma_i r_i \qquad (36)$$

Multiplying on the left by $-A$ and adding b gives:

$$r_k = r_0 - \sum_{i=0}^{k-1} \gamma_i A r_i \qquad (37)$$

Applying this equation to itself to yield an expression for r_k in terms of r_0 yields

$$\begin{aligned} r_k &= r_0 - \gamma_0 A r_0 - \gamma_1 A r_1 - \ldots - \gamma_{k-1} A r_{k-1} \\ &= r_0 - \gamma_0 A r_0 - \gamma_1 A(r_0 - \gamma_0 A r_0) - \gamma_2 A(r_0 - \gamma_1 A(r_0 - \gamma_0 A r_0)) - \ldots \\ &\quad - \gamma_{k-1} A(r_0 - \gamma_{k-2} A(r_0 - \ldots - \gamma_1 A(r_0 - \gamma_0 A r_0)) \ldots) \\ &= r_0 - \sum_{i=0}^{k} \delta_i A^i r_0 \qquad (38) \end{aligned}$$

i.e. $r_k \in \mathrm{span}(r_0, Ar_0, A^2 r_0, \ldots, A^k r_0)$, which is the Krylov subspace spanned by A and r_0. It now follows from equation (36) that:

$$x_k \in x_0 + \mathrm{span}(r_0, Ar_0, A^2 r_0, \ldots, A^{k-1} r_0) \qquad (39)$$

so the iterates lie in a shifted Krylov subspace spanned by A and r_0. This property holds for all exact conjugate Krylov subspace methods (cf. Eq. (23)).

Note that equation (38) can be used to express r_k in polynomial form, i.e.

$$r_k = \Psi_k(A) r_0 \tag{40}$$

where

$$\Psi_k(A) = (I - \sum_{i=1}^{k-1} \delta_i A^i)$$

This representation is just a formal way of expressing r_k as a polynomial in A applied to a starting residual; $\Psi_k(A)$ is not explicitly computed but is rather implicitly computed as the algorithm proceeds. The importance of this residual polynomial representation will become apparent when considering the development of variants of the Biconjugate Gradients algorithm such as CGS and BiCGSTAB.

Relating these results to the framework presented in Sec. 4.4.1, we see the CG algorithm is an exact Krylov subspace method (i.e. $\sigma_k = k$) with $q_{k-i,k} = r_{k-i}$ and $Z_k = I$. Substituting these parameters into equation (24) yields the first orthogonality condition of equation (33) i.e. $r_k^T r_j = 0$ for $k \neq j$. Also, from equation (39), the $q_{k-i,k}$ are equivalently given by $q_{k-i,k} = A^{k-i} r_0$, so $B = A$ and $z = r_0$. Since $Z_k = I$ is constant, CG falls into the class of generalised CG methods.

CG ALGORITHM:

1. Initialise
    - $r_0 = b - Ax_0$
    - $p_0 = r_0$
2. Iterate
    - **for** $k = 1, 2, \ldots$

      $\alpha_{k-1} = r_{k-1}^T r_{k-1} / p_{k-1}^T A p_{k-1}$

      $x_k = x_{k-1} + \alpha_{k-1} p_{k-1}$

      $r_k = r_{k-1} - \alpha_{k-1} A p_{k-1}$

      $\beta_k = r_k^T r_k / r_{k-1}^T r_{k-1}$

      $p_k = r_k + \beta_k p_{k-1}$

The algorithm performs one-matrix vector multiplication with A, three vector updates and two inner products per iteration. It requires storage for four vectors (one each for p, r and x and one for the product Ap_{k-1}).

The convergence rate of the conjugate gradient algorithm depends on the spectral condition number $\kappa = \kappa_2(A) = \lambda_{max}/\lambda_{min}$ where λ_{max} and λ_{min} are the largest and smallest eigenvalues of A respectively [GL89, §10.2.8]. In particular, the error at iteration k is bounded by:

$$\|e\|_A = \|x - x_k\|_A \leq 2\|x - x_0\|_A \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k \tag{41}$$

where $\|e\|_A$ denotes the A-norm given by $\sqrt{e^T A e}$. Like optimal SOR, the rate of convergence is proportional to $\kappa^{-\frac{1}{2}}$. More complex convergence results taking into account the entire spectrum of A are given in [SV86].

CGNR/CGNE: Conjugate Gradient using the normal equations

An obvious approach to generalising the CG method to non-SPD A matrices is to find ways of applying the CG method to the matrices $A^T A$ and AA^T, since these matrices will be SPD for non-singular A. One way is to multiply equation (19) by A^T on both sides which yields:

$$A^T A x = A^T b = y$$

This leads to a technique for minimizing the two-norm of the residuals at each step (CGNR). Alternatively, one can solve the system:

$$AA^T z = b$$

for z and compute the desired solution as $x = A^T z$. This leads to technique for minimizing the two-norm of the error at each step (CGNE).

CGNR generates a Krylov space spanned by $A^T A$ and r_0, while CGNE generates a Krylov space spanned by AA^T and r_0. The products $A^T A$ and AA^T do not have to be calculated but can be incorporated into the algorithm implicitly.

Like the original CG algorithm, CGNR and CGNE are exact generalised CG methods that can be formulated using short recurrences. For both methods, $q_{k-i,k} = A^T r_{k-i}$. $Z_k = A^{-T}$ for CGNR and $Z_k = A$ for CGNE [Wei94].

CGNR ALGORITHM [Mei94, §2.5]

1. Initialise
   - $r_0 = A^T(b - Ax)$
   - $p_0 = r_0$
2. Iterate
   - **for** $k = 1, 2, \ldots$

$$\alpha_{k-1} = r_{k-1}^T r_{k-1} / p_{k-1}^T A^T A p_{k-1}$$
$$x_k = x_{k-1} + \alpha_{k-1} p_{k-1}$$
$$r_k = r_{k-1} - \alpha_{k-1} A^T A p_{k-1}$$
$$\beta_k = r_k^T r_k / r_{k-1}^T r_{k-1}$$
$$p_k = r_k + \beta_k p_{k-1}$$

CGNE ALGORITHM [Mei94, §2.5]

1. Initialise
   - $r_0 = b - Ax$
   - $\tilde{p}_0 = A^T r_0$
2. Iterate
   - **for** $k = 1, 2, \ldots$

$$\alpha_{k-1} = r_{k-1}^T r_{k-1} / \tilde{p}_{k-1}^T \tilde{p}_{k-1}$$
$$x_k = x_{k-1} + \alpha_{k-1} \tilde{p}_{k-1}$$
$$r_k = r_{k-1} - \alpha_{k-1} A \tilde{p}_{k-1}$$
$$\beta_k = r_k^T r_k / r_{k-1}^T r_{k-1}$$
$$\tilde{p}_k = A^T r_k + \beta_k \tilde{p}_{k-1}$$

The algorithms perform two matrix vector multiplications (one with A and one with A^T), three vector updates and 2 inner products per iteration. Both algorithms require storage for 5 vectors (r, p, x, and two for the matrix-vector products).

CGNR and CGNE have the same theoretical convergence properties as classical CG and given exact arithmetic, the algorithm should converge in fewer than n steps. However, there is a drawback: the condition number of $A^T A$ or AA^T is given by the square of the condition number of A, so from equation (41) it can be seen that the convergence rate is actually much slower than that of CG. In fact, given the finite precision arithmetic available on a computer, this poor conditioning sometimes leads to incorrect results.

The poor convergence and accuracy of these methods, together with the increased storage cost incurred in providing both row and column access to A, mean that CGNR and CGNE are only useful in practice when memory is not at a premium and a good preconditioner is available to improve the spectrum of A.

GMRES: Generalised Minimum RESidual

The GMRES method [SS86] aims to generalise the CG method to the non-symmetric case by maintaining the orthogonality of the residual vectors at the expense of losing the three term recurrence. The kth GMRES iterate is given by:

$$x_k = x_0 + z_k$$

where the correction z_k is chosen from the Krylov subspace

$$\mathcal{K}_k(A, r_0) = \mathrm{span}(r_0, Ar_0, A^2 r_0, ..., A^{k-1} r_0) \tag{42}$$

such that z_k minimizes the two-norm of the kth residual,

$$||r_k||_2 = ||b - A(x_0 + z_k)||_2 = ||r_0 - Az_k||_2$$

Determining the correction z_k involves constructing a basis for $\mathcal{K}_k(A, r_0)$ and then solving an k-dimensional least-squares problem for the coefficients of that linear combination of the basis elements which minimizes the sum of squares of the elements of the residual vector.

The original paper by Saad and Schultz [SS86] uses a Gram Schmidt-based algorithm known as the Arnoldi process (see Sec. 4.4.2) to construct an orthonormal basis $(v_1, v_2, \ldots, v_k)$ for the subspace of equation (42). Note that other ways of forming the orthonormal basis are possible. Walker, for example, presents a procedure based on Householder transformations which is slightly more expensive but more numerically stable than the Arnoldi process [Wal88a].

Now, from the formula for $\hat{v}_{j+1}$ given in the algorithm for the Arnoldi process in Sec. 4.4.2, the relationship:

$$AV_k = V_{k+1}\bar{H}_k \tag{43}$$

can be derived (see [Ste94, pg. 191] for proof). If we write $z_k = V_k y$, where y denotes the desired coefficients of the linear combination of basis elements, then the original least squares problem:

$$\min_{z \in \mathcal{K}_k(A, r_0)} \|r_0 - Az_k\|_2$$

can be rewritten in terms of a minimization of the function:

$$J(y) = \|\beta v_1 - AV_k y\|_2$$

where $\beta = \|r_0\|_2$ and $v_1 = r_0/\|r_0\|_2$. Using equation (43) and using the fact that V_{k+1} is orthonormal, it follows that

$$J(y) = \|V_{k+1}(\beta e_1 - \bar{H}_k y)\|_2 = \|\beta e_1 - \bar{H} y\|_2 \tag{44}$$

where $e_1 = (1, 0, 0, \ldots, 0)^T$ is a $(k+1)$-vector. Thus the problem of finding the correction z_k which minimizes the residual has been reduced to the problem of finding the vector y which minimizes $J(y)$. The simple structure of $\bar{H}$ means that y can be found efficiently if, for example, a QR factorization of $\bar{H}$ is maintained; in this case y_m can be determined as a solution to an upper triangular system.

GMRES is optimal in that it provides the smallest residual for a fixed number of iterations steps. However, the cost of maintaining this optimality increases with each iteration step. At the kth iteration:

- The space required is $\mathcal{O}(k)$ since the k vectors making up the orthonormal basis have to be stored.
- The time required is $\mathcal{O}(k^2)$; this corresponds to the effort required to solve the $(k+1) \times k$ least squares system used to minimize $\|r_k\|_2$.

In practical implementations, GMRES is therefore usually restarted every m iterations. This restarted form is called GMRES(m).

In its full (non-restarted) form, the GMRES algorithm is an exact generalised CG method with $q_{k-i,k} = r_{k-i}$ and $Z_k = A$ [Wei94]. GMRES(m) is a restarted procedure with $\sigma_k = (k-1) \bmod \sigma_m + 1$.

GMRES(m) ALGORITHM [SS86]

1. Initialize
   - Choose vector x_0
   - Calculate $r_0 = b - Ax_0$ and $v_1 = r_0/\|r_0\|_2$
2. Arnoldi Process
   - **for** $j = 1, 2, \ldots,$ m

     for $i = 1, 2, \ldots, j$

     $h_{i,j} = {v_i}^T A v_j$

     $\hat{v}_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j} v_i$

     $h_{j+1,j} = \|\hat{v}_{j+1}\|$

     $v_{j+1} = \hat{v}_{j+1}/h_{j+1,j}$
3. Form approximate solution and restart
   - Calculate $x_k = x_0 + V_k y_k$ where y_k minimizes equation (44).
   - Calculate $r_k = b - Ax_k$
   - If satisfied stop, else set $x_0 = x_m$ and $v_1 = r_m/\|r_m\|_2$ and restart at step (2)

The algorithm performs one matrix vector multiplication (with A) per iteration. At iteration k, the algorithm performs $i = (k-1) \bmod m + 1$ inner products and $i + 1$ vector updates. The algorithm uses $i + 4$ vectors of storage.

Like the classical conjugate gradient algorithm, full GMRES terminates in $m \leq n$ steps given exact arithmetic. Since the residual norm is minimized at each step, the convergence of full GMRES is monotonic i.e. $\|r_k\|_2 \leq \|r_i\|_2$ for all $k > i$. In addition, it can be shown that the algorithm cannot break down unless the solution has already been found [SS86].

GMRES(m), on the other hand, usually converges more slowly than non-restarted GMRES. In fact, GMRES(m) is not guaranteed to converge for general A, and may continue indefinitely. The likelihood of non-convergence decreases with increasing m and vanishes when $m = n$. Like full GMRES, GMRES(m) will not break down unless it has already converged.

GMRES(m) with a large value of m is often used as a basis of comparison for new algorithms because of its robustness and good convergence properties. However, for reasonable values of m (say $m \geq 20$), GMRES(m) uses large amounts of memory and it is thus generally not competitive with other algorithms in terms of space.

BiCG: Biconjugate Gradient Algorithm

The biconjugate gradient algorithm [Fle76] is a reformulation of the classical non-symmetric Lanczos algorithm. It attempts to generalise the CG algorithm to the non-symmetric case by maintaining the three term recurrence while sacrificing the orthogonality of the residuals. In order to ensure finite termination, the algorithm makes use of a "shadow" system based on A^T to construct a sequence of "pseudo-residuals" $\tilde{r}_k$, which satisfies the biorthogonality condition:

$$\tilde{r}_i^T r_j = {r_i}^T \tilde{r}_j = 0 \quad \text{for } j < i$$

and a sequence of "pseudo-directions" $\tilde{p}_k$, which satisfies the biconjugacy condition:

$$\tilde{p}_i^T A p_j = {p_i}^T A^T \tilde{p}_j = 0 \quad \text{for } j < i$$

The vectors r_k and $\tilde{r}_k$ generated by BiCG are scalar multiples of the vectors v_k and w_k generated by the non-symmetric Lanczos algorithm started with $v_1 = r_0$ and $w_1 = \tilde{r_0}$ [FGN92, pg. 17].

Weiss [Wei95] shows that BiCG is equivalent to CG applied the double system:

$$\hat{A}\hat{x} = \hat{b}$$

where

$$\hat{A} = \begin{pmatrix} A & 0 \\ 0 & A^T \end{pmatrix}, \quad \hat{x} = \begin{pmatrix} x \\ \tilde{x} \end{pmatrix}, \quad \hat{b} = \begin{pmatrix} b \\ \tilde{b} \end{pmatrix} \tag{45}$$

and $\tilde{b}$ is arbitrary. The residuals are given by:

$$\hat{r}_k = \begin{pmatrix} r_k \\ \tilde{r}_k \end{pmatrix} = \begin{pmatrix} b \\ \tilde{b} \end{pmatrix} - \begin{pmatrix} A & 0 \\ 0 & A^T \end{pmatrix} \begin{pmatrix} x \\ \tilde{x} \end{pmatrix} = \begin{pmatrix} b - Ax_k \\ \tilde{b} - A^T \tilde{x}_k \end{pmatrix} \tag{46}$$

The iterates $\tilde{x}_k$ of the shadow system $A^T \tilde{x} = \tilde{b}$ converge at about the same speed as the true solution [Son89]; however, and this is one of the main criticisms of BiCG, BiCG does not exploit this convergence.

BiCG is an exact generalised CG method which is formulated using a short recurrence. Using the notation of equations (45) and (46),

$$q_{k-i,k} = \hat{r}_{k-i} \quad \text{and} \quad Z_k = Z = \begin{pmatrix} 0 & I \\ I & 0 \end{pmatrix}.$$

BiCG Algorithm [Fle76]

1. Initialise
   - $r_0 = b - Ax$
   - $\tilde{r}_0 = b - A^T x$
   - $p_0 = r_0$
   - $\tilde{p}_0 = \tilde{r}_0$
2. Iterate
   - **for** $k = 1, 2, \ldots$

$$\alpha_{k-1} = (\tilde{r}_{k-1}^T r_{k-1})/(\tilde{p}_{k-1}^T A p_{k-1})$$
$$x_k = x_{k-1} + \alpha_{k-1} p_{k-1}$$
$$r_k = r_{k-1} - \alpha_{k-1} A p_{k-1}$$
$$\tilde{r}_k = \tilde{r}_{k-1} - \alpha_{k-1} A^T \tilde{p}_{k-1}$$
$$\beta_k = (\tilde{r}_k^T r_k)/(\tilde{r}_{k-1}^T r_{k-1})$$
$$p_k = r_k + \beta_k p_{k-1}$$
$$\tilde{p}_k = \tilde{r}_k + \beta_k \tilde{p}_{k-1}$$

The algorithm basically performs twice the work of the CG algorithm for each iteration because it needs to perform two matrix multiplications (one with A, one with A^T). Note, however, that the matrix multiplications are independent and can be done in parallel. The algorithm requires storage for 7 vectors (p, $\tilde{p}$, r, x, $\tilde{r}$, and two for the matrix-vector products Ap and $A^T\tilde{p}$).

Like the CG algorithm, BiCG should terminate in fewer than n steps if convergence occurs. However, since the residual minimizing property of CG has been lost, BiCG can produce highly oscillating residuals. In addition, the algorithm can even break down should a zero or near-zero denominator occur in the computation of α_k or β_k. Some breakdowns can be fixed by simply restarting the algorithm if a zero denominator is detected. Breakdowns in the computation of β_k which occur because $\tilde{r}_{k-1}^T r_{k-1} \approx 0$ with $r_{k-1} \neq 0$ and $\tilde{r}_{k-1} \neq 0$ correspond to a *serious breakdown* in the underlying Lanczos process and may be avoided through the use of lookahead Lanczos algorithms.

Quantitative analytical results characterizing the convergence of BiCG are either non-existent or extremely scarce in the literature; however, some bounds on the error for methods closely related to BiCG are available; see e.g. [FN91] for error bounds on QMR.

Despite its erratic convergence behaviour and the need to perform matrix-vector multiplications with both A and A^T, BiCG is still particularly significant because it lead directly to the development of several more efficient techniques with faster and/or smoother convergence, such as CGS, BiCGSTAB and QMR.

4.4.4 Conjugate Krylov Subspace Techniques

CGS: Conjugate Gradient Squared

The Conjugate Gradient Squared algorithm [Son89] aims to remedy two weaknesses of BiCG. Firstly, BiCG ignores the convergence of the pseudo-residuals $\tilde{r}_k$ of the shadow system $A^T\tilde{x} = \tilde{b}$, even though the pseudo-residuals can be expected to converge at about the same rate as the true residuals. Secondly, BiCG involves matrix-vector products with A^T; this means that both row and column access must be provided to A.

At iteration k of the BiCG algorithm we have:

$$r_k = \Psi_k(A)r_0 \quad \text{and} \quad \tilde{r}_k = \Psi_k(A^T)\tilde{r}_0$$

where Ψ_k is a matrix polynomial of degree k (cf. Eq. (40)). Now the only time the BiCG algorithm makes use of A^T is when computing $\tilde{r}_k$, which is itself only used in calculating the inner product $\tilde{r}_k^T r_k$. Sonneveld observed that this inner product may be instead computed in terms of A only as follows [Ste94, pg. 221]:

$$\tilde{r}_k^T r_k = (\Psi(A^T)\tilde{r}_0)^T \Psi(A) r_0 = \tilde{r}_0^T(\Psi(A^T))^T \Psi(A) r_0 = \tilde{r}_0^T(\Psi^2(A)) r_0$$

This suggests an algorithm which generates its residuals as $r_k = \Psi^2(A)r_0$ instead of the standard form $r_k = \Psi(A)r_0$; the inner product $\tilde{r}_k^T r_k$ can then be calculated as $\tilde{r}_0^T r_k$ where

$r_k = \Psi^2(A)r_0$. Doing this leads to the CGS algorithm, which not only removes the need for multiplications with A^T, but also converges faster than BiCG. This is because $\Psi(A)$ behaves like a contraction operator; by applying $\Psi(A)$ twice, the contraction effect on r_k is increased, resulting in faster convergence.

CGS is an exact conjugate Krylov subspace method that can be formulated using terms of a short recurrence. In terms of the classification of Sec. 4.4.1, $q_{k-i,k} = r_{k-i}$. Z_k depends on k and its exact determination is very complex [Wei94].

CGS Algorithm [Vor92]

1. Initialise
   - $r_0 = b - Ax_0$
   - $\hat{r}_0$ is an arbitrary vector such that ${r_0}^T \hat{r}_0 \neq 0$ e.g. $\hat{r}_0 = r_0$
   - $\rho_0 = 1$
   - $p_0 = q_0 = 0$
2. Iterate
   - **for** $k = 1, 2, \ldots$

$$\rho_k = \hat{r}_0^T r_{k-1}$$
$$\beta = \rho_k / \rho_{k-1}$$
$$u = r_{k-1} + \beta q_{k-1}$$
$$p_k = u + \beta(q_{k-1} + \beta p_{k-1})$$
$$v = Ap_k$$
$$\alpha = \rho_k / \hat{(r)}_0^{\;T} v$$
$$q_k = u - \alpha v$$
$$w = u + q_k$$
$$x_k = x_{k-1} + \alpha w$$
$$r_k = r_{k-1} - \alpha A w$$

Sometimes the update to the residual vector in the last line is replaced by the computation of the true residual $r_k = b - Ax_k$; this prevents accumulated cancellation effects which can occur in finite precision arithmetic when using the updated residual method.

CGS performs two matrix vector multiplications (both with A), 6 vector updates and 2 inner products per iteration. Thus each iteration of CGS involves a similar amount of effort to an iteration of BiCG. Note, however, that the CGS algorithm is not as easily parallelisable as the BiCG algorithm because the two matrix products are dependent. Storage for 8 vectors is required.

While CGS generally converges at rate faster than BiCG, it is still susceptible to the same erratic convergence behaviour and breakdown possibilities as BiCG. The convergence of CGS is in fact sometimes more erratic than that of BiCG because the contraction effect of $\Psi(A)$ depends on it being applied to r_0 (see [Vor92]); applying $\Psi(A)$ to $\Psi(A)r_0$ (as in $\Psi^2(A)r_0$) can sometimes result in the norm of $\Psi^2(A)r_0$ being larger than the norm of $\Psi(A)r_0$. As a result, large local peaks are often observed in convergence graphs of CGS. These peaks do not appear to delay convergence but can cause troublesome cancellation effects in the calculation of the updated residuals. The true residual version of the algorithm does not suffer from this problem but can take longer to converge.

BiCGSTAB: CGS Stabilised

Van der Vorst's BiCGSTAB algorithm [Vor92] attempts to improve the CGS algorithm by retaining the attractive convergence speed while stabilising the convergence behaviour. The central idea is to replace the residual matrix polynomial $\Psi_k^2(A)$ with one of form $\Phi_k(A)\Psi_k(A)$ where $\Phi_k(A)$ will have a more stable contraction effect on $\Phi_k(A)r_0$ than $\Psi_k(A)$. Ideally, one would like $\Phi_k(A)$ to be related to a class of polynomials with good optimality properties, such as the Chebyshev polynomials. However, doing this would require complex parameter estimation (see e.g. [HY81, §6]). Instead, Van der Vorst uses a polynomial which has a simple recurrence relation; this polynomial is built up in factored form given by

$$\Phi_k(A) = (I - \omega_k A)\Phi_{k-1}(A)$$

or, equivalently,

$$\Phi_k(A) = (I - \omega_1 A)(I - \omega_2 A)\dots(I - \omega_k A)$$

where ω_k is calculated at the kth iteration step to minimize the two-norm of the residual $r_k = \Phi_k(A)\Psi_k(A)$. Note that at step k only ω_k needs to be determined; the other omega's have been defined already in steps 1 through $k-1$ of the algorithm.

From the form of $r_k = \Phi_k(A)\Psi_k(A)$, we can see that BiCGSTAB is a hybrid combination of of BiCG and GMRES(1), since it combines the residual polynomial $\Psi_k(A)$ of the BiCG method with the one-dimensional residual minimising effect of GMRES(1) through $\Phi_k(A)$.

A weakness of BiCGSTAB is that Φ_k has only real roots. However, it is known that, for matrices with complex spectra, optimal reduction polynomials may also have complex roots [Vor93, §5.4.5]. Gutknecht's BiCGSTAB2 method [Gut93b] extends BiCGSTAB by making use of a quadratic polynomial to expand Φ_k by a quadratic factor on even-numbered steps. In this scheme, a two-dimensional minimization is performed. BiCGSTAB2 can thus be seen as a combination of BiCG and GMRES(2).

The BiCGSTAB(l) algorithm of Sleijpen and Fokkema [SF93] takes the generalisation of BiCGSTAB to its logical conclusion. This is a robust method which combines GMRES(l) with BiCG. BiCGSTAB(1) computes the same iterates as BiCGSTAB and BiCGSTAB(2) is mathematically equivalent to BiCGSTAB2 in exact arithmetic. However BICGSTAB(2) is more efficient and more robust than BiCGSTAB2.

BiCGSTAB is an exact conjugate Krylov subspace method that can be formulated using terms of a short recurrence. In terms of the classification of Sec. 4.4.1, $q_{k-i,k} = r_{k-i}$. As for CGS, Z_k varies with k but its exact determination is very complex.

BiCGSTAB Algorithm [Vor92]

1. Initialise
   - $r_0 = b - Ax_0$
   - $\hat{r}_0$ is an arbitrary vector such that ${r_0}^T \hat{r}_0 \neq 0$ e.g. $\hat{r}_0 = r_0$
   - $\rho_0 = \alpha = \omega_0 = 1$
   - $v_0 = p_0 = 0$
2. Iterate
   - **for** $k = 1, 2, \ldots$

     $\rho_k = \hat{r}_0^T r_{k-1}$

     $\beta = (\rho_k / \rho_{k-1}) / (\alpha / \omega k - 1)$

     $p_k = r_{k-1} + \beta(p_{k-1} - \omega_{k-1} v_{k-1})$

     $v_k = A p_k$

$$\alpha = \rho_k / \hat{r}_0^T v_k$$
$$s = r_{k-1} - \alpha v_k$$
$$t = As$$
$$\omega_i = t^T s / t^T t$$
$$x_k = x_{k-1} + \alpha p_k + \omega_k s$$
$$r_k = s - \omega_k t$$

BiCGSTAB performs two matrix vector multiplications (both with A), 6 vector updates and 4 inner products (2 more than CGS) per iteration. Thus each iteration of BiCGSTAB is slightly more expensive than an iteration of CGS. Storage for 7 vectors is required (1 vector less than CGS).

BiCGSTAB(2) Algorithm [Vor93]

1. Initialise
   - $r_0 = b - Ax_0$
   - $\hat{r}_0$ is an arbitrary vector such that ${r_0}^T \hat{r}_0 \neq 0$ e.g. $\hat{r}_0 = r_0$
   - $\rho_0 = \omega_2 = 1$
   - $u = \alpha = 0$
2. Iterate
   - **for** $k = 0, 2, 4, 6, \ldots$

$$\rho_0 = \omega_2 \rho_0$$
$$\rho_1 = \tilde{r}_0^T r_k; \beta = \alpha \rho_1 / \rho_0; \rho_0 = \rho_1$$
$$u = r_k - \beta u$$
$$v = Au$$
$$\gamma = v^T \tilde{r}_0$$
$$\alpha = \rho_0 / \gamma$$
$$r = r_k - \alpha v$$
$$s = Ar$$
$$x = x_k + \alpha u$$

$$\rho_1 = \tilde{r}_0^T s; \beta = \alpha\rho_1/\rho_0; \rho_0 = \rho_1$$
$$v = s - \beta v$$
$$w = Av$$
$$\gamma = w^T \tilde{r}_0$$
$$\alpha = \rho_0/\gamma$$
$$u = r - \beta u$$
$$r = r - \alpha v$$
$$s = s - \alpha w$$
$$t = As$$
$$\omega_1 = r^T s; \mu = s^T s; \nu = s^T t; \tau = t^T t$$
$$\omega_2 = r^T t; \tau = \tau - \nu^2; \omega_2 = (\omega_2 - \nu\omega_1/\mu)/\tau$$
$$\omega_1 = (\omega_1 - \nu\omega_2)/\mu$$
$$x_{k+2} = x + \omega_1 r + \omega_2 s + \alpha u$$
$$r_{k+2} = r - \omega_1 s - \omega_2 t$$
$$u = u - \omega_1 v - \omega_2 w$$
$$s = r_{k-1} - \alpha v_k$$
$$t = As$$
$$\omega_i = t^T s/t^T t$$
$$x_k = x_{k-1} + \alpha p_k + \omega_k s$$
$$r_k = s - \omega_k t$$

BICGSTAB(l) requires $2l + 10$ vector updates, $l + 7$ inner products and 4 matrix vector multiplications per two iteration cycle. Storage for $2l + 5$ vectors is required [SV95].

BiCGSTAB generally converges slightly faster and more smoothly than CGS; in addition, the updated residual is generally more accurate than CGS. This behaviour can be attributed to the residual minimising effect of Φ_k. However, there are cases where CGS converges well, but where BiCGSTAB converges slowly, stagnates or even breaks down. Such situations can occur when ω_k is close to zero; this is not uncommon in matrices with complex eigenvalues with large imaginary parts. In finite precision arithmetic $\omega_k \approx 0$ leads to inaccurate BiCG coefficients (i.e. inaccurate α and β) which can upset the convergence [SF93].

BiCGSTAB(l) generally converges better than BiCGSTAB because it performs a better local minimization through Φ_k and maintains a more stable underlying BiCG process [SV95]. A value of $l = 2$ is usually adequate to achieve good convergence; sometimes, however, a larger value of $l = 4$ or $l = 8$ is necessary.

QMR: Quasi-minimal Residual

The QMR algorithm of Freund and Nachtigal [FN91] attempts to stabilise the irregular convergence behaviour of BiCG by introducing a relaxed residual minimization property which is less optimal than that of GMRES, but which can still be implemented using short recurrences. In addition, look-ahead versions of the QMR algorithm are available [FN94] which aim to address the problem of BiCG breakdown.

Like BiCG, the QMR algorithm is based on the non-symmetric Lanczos algorithm. The recurrence formula of equations (25) and (26) can be written in matrix form as:

$$AV_k = V_{k+1}\bar{H}_k \tag{47}$$

$$A^T W_k = W_{k+1}\tilde{H}_k \tag{48}$$

where V_k is the $n \times k$ matrix with columns $v_1, v_2, \ldots, v_k$ and W_k is the $n \times k$ matrix with columns $w_1, w_2 \ldots, w_k$. $\bar{H}_k$ and $\tilde{H}_k$ are tridiagonal $(k+1) \times k$ matrices involving the recurrence coefficients α, β and γ and the scaling factors $\nu_k = 1/\|\tilde{v}_k\|_2$ and $\mu_k = 1/\|\tilde{w}_k\|_2$ (see [Wei95, §3.1] for their full form).

Now if we let $v_1 = r_0/\|r_0\|_2$, then the Lanczos algorithm will generate $v_1, v_2, \ldots, v_k$ spanning $\mathcal{K}_k(A, r_0)$ and $w_1, w_2, \ldots, w_k$ spanning $\mathcal{K}_k(A^T, r_0)$. As is the case for GMRES, the kth iterate of the QMR algorithm is given by:

$$x_k = x_0 + z_k$$

where the correction z_k is chosen from the Krylov subspace $\mathcal{K}_k(A, r_0)$. Since $v_1, v_2, \ldots, v_k$ forms a basis for the subspace, z_k may be written as $z_k = V_k y$ where y denotes the coefficients of the linear combination of basis elements. Using this fact and equation (47), the kth residual is given by:

$$r_k = r_0 - Az = r_0 - AV_k y = r_0 - V_{k+1}\bar{H}_k y = V_{k+1}(\beta e_1 - \bar{H}_k y) \tag{49}$$

where $\beta = \|r_0\|_2$ and $e_1 = (1, 0, 0, \ldots, 0)^T$ is $(k+1)$-vector. Note that the development so far has been very similar to that of GMRES, except now we cannot proceed by using the fact that the columns V_{k+1} are orthonormal, since the short recurrences of the non-symmetric Lanczos algorithm generate biorthonormal vectors, and not mutually orthonormal vectors like Arnoldi's algorithm. In fact, attempting to minimize the two-norm of the residual as given above would lead to an algorithm equivalent to GMRES, but would involve solving an $n \times (k+1)$ least-squares problem requiring $\mathcal{O}(nk^2)$ work and $\mathcal{O}(nk)$ storage [Nac91].

Instead, Freund and Nachtigal introduce a $(k+1) \times (k+1)$ diagonal scaling matrix given by

$$\Omega_n = \mathrm{diag}(\omega_1, \omega_2, \ldots, \omega_{k+1}), \quad \omega_j > 0, \quad j = 1, 2, \ldots, k+1$$

into equation (49) which gives:

$$r_k = V_{k+1}\Omega_k^{-1}\Omega_k(\beta e_1 - \bar{H}_k y) = V_{k+1}\Omega_k^{-1}(\omega_1 \beta e_1 - \Omega_k \bar{H}_k y)$$

Note that there is no known optimality condition for choosing the weights; usually $\omega_j = 1$ for $j = 1, 2, \ldots, k+1$. Now if we are willing to sacrifice the optimality of a true residual minimization, we can ignore the $V_{k+1}\Omega_k^{-1}$ term and solve a much smaller $(k+1) \times k$ least squares problem which involves only the bracketed term of r_k, i.e.

$$\min_{y \in \mathbb{R}^k} \|(\omega_1 \beta e_1 - \Omega_k \bar{H}_k y)\|_2$$

Thus the expensive true residual minimizing property has been replaced with a cheaper quasi-optimal property.

QMR is an exact Krylov subspace method that can be formulated with a short recurrence. In this case, $q_{k-i,k} = r_{k-i}$ and

$$Z_k = W_k D_k^{-T} \Omega_k^T \Omega_k D_k^{-1} W_k^T A$$

where W_k is the matrix of shadow Lanczos vectors from equation (48) and D_k is a $k \times k$ diagonal matrix derived by rewriting equation (27) in matrix form as:

$$W_k^T V_k = D_k.$$

QMR ALGORITHM [BBC+94]

1. Initialise
   - $r_0 = b - Ax_0$
   - $\tilde{w}_1$ is an arbitrary vector such that $\tilde{w}_1^T \tilde{r}_0 \neq 0$ e.g. $\tilde{w}_1 = r_0$
   - $\tilde{v}_1 = r_0$
   - $\rho_1 = \|\tilde{v}_1\|_2$
   - $\xi = \|\tilde{w}_1\|$
   - $\gamma_0 = 1$
   - $\eta_0 = -1$
   - $\tau_0 = \|r_0\|$
   - $\theta_0 = \eta_0 = 0$
2. Iterate
   - **for** $k = 1, 2, \ldots$

     if $\rho_k = 0$ or $\xi_k = 0$ **method fails**

     $v_k = \tilde{v}_k / \rho_k$

     $w_k = \tilde{w}_k / \xi_k$

     $\delta_k = w_k^T V_k$; **if** $\delta_k = 0$ **method fails**

     if $k = 1$

     $p_1 = v_1; q_1 = w_1$

     else

     $p_i = v_k - (\xi_k \delta_k / \epsilon_{k-1}) p_{k-1}$

     $q_i = w_k - (\rho_k \delta_k / \epsilon_{k-1}) q_{k-1}$

     endif

     $\tilde{p} = Ap_k$

     $\epsilon_k = q_k^T \tilde{p}$; **if** $\epsilon_k = 0$ **method fails**

     $\beta_k = \epsilon_k / \delta_k$; **if** $\beta_k = 0$ **method fails**

     $\tilde{v}_{k+1} = \tilde{p} - \beta_k v_k$

     $\rho_{k+1} = \|\tilde{v}_{k+1}\|_2$

     $\tilde{w}_{k+1} = A^T q_k - \beta_k w_i$

$\xi_{k+1} = \|\tilde{w}_{k+1}\|_2$

$\theta_k = \rho_{k+1}/(\gamma_{k-1}|\beta_k|)$; $\gamma_k = 1/\sqrt{1+\theta_k^2}$; **if** $\gamma_k = 0$ **method fails**

$\eta_k = -\eta_{k-1}\rho_k\gamma_k^2/(\beta_k\gamma_{k-1}^2)$

if $k = 1$

$d_1 = \eta_1 p_1$; $s_1 = \eta_1 \tilde{p}$

else

$d_k = \eta_k p_k + (\theta_{k-1}\gamma_k)^2 d_{k-1}$

$s_k = \eta_k \tilde{p} + (\theta_{k-1}\gamma_k)^2 s_{k-1}$

endif

$x_k = x_{k-1} + d_k$

$r_k = r_{k-1} - s_k$

QMR performs 2 matrix-vector multiplications (one with A and one with A^T) per iteration. Storage for 8 vectors is required.

Freund and Nachtigal [FN91] give general error bounds showing that the upper bound for the kth residual norm of QMR is greater than that of GMRES by a factor of $\tau\sqrt{k+1}$, where τ is a constant related to, among other things, the conditioning of A and H_k. Thus GMRES and QMR have similar upper bounds on their errors.

Practical experience suggests that QMR converges more smoothly than BiCG, but it is not necessarily faster. If implemented, the look-ahead steps of QMR make it more robust then BiCG since they prevent all but so-called "incurable" breakdowns in the underlying Lanczos process.

TFQMR: Transpose free Quasi-minimal Residual

Since QMR is the result of applying quasi-minimal smoothing to the BiCG algorithm, it may be also be beneficial to apply quasi-minimal smoothing to the CGS algorithm. Doing this leads to Freund's TFQMR algorithm [Fre93] which, like CGS, has the advantage that it does not involve multiplications with A^T.

CGS and TFQMR are closely related since TFQMR may be derived from CGS by changing only a few lines in the algorithm and the CGS iterates may be easily recovered from the TFQMR process.

TFQMR ALGORITHM [Fre93]

1. Initialise
   - $r_0 = b - Ax_0$
   - $\tilde{r}_0$ is an arbitrary vector such that ${r_0}^T \tilde{r}_0 \neq 0$ e.g. $\hat{r}_0 = r_0$
   - $w_1 = y_1 = r_0$
   - $v_0 = Ay_1$
   - $d_0 = 0$
   - $\tau_0 = \|r_0\|$
   - $\theta_0 = \eta_0 = 0$

2. Iterate
   - **for** $k = 1, 2, \ldots$

     $\sigma_{k-1} = \tilde{r}_0^T v_{k-1}$

     $\alpha_{k-1} = \rho_{k-1}/\sigma_{k-1}$

     $y_{2k} = y_{2k-1} - \alpha_{k-1} v_{k-1}$

     for $m = 2k-1, 2k$

     $w_{k+1} = w_k - \alpha_{k-1} Ay_m$

     $\theta_m = \|w_{m+1}\|/\tau_{m-1}$

     $c_m = 1/\sqrt{1+\theta_m^2}$

     $\tau_m = \tau_{m-1}\theta_m c_m$

     $\eta_m = c_m^2 \alpha_{k-1}$

     $d_m = y_m + (\theta_{m-1}^2 \eta_{m-1}/\alpha_{k-1}) d_{m-1}$

     $x_m = x_{m-1} + \eta_m d_m$

     if x_m has converged **stop**

     $\rho_k = \tilde{r}_0^T w_{2k+1}$

     $\beta_k = \rho_k/\rho_{k-1}$

     $y_{2k+1} = w_{2k+1} + \beta_k y_2 k$

     $v_n = Ay_{2k+1} + \beta_k(Ay_{2k} + \beta_k v_{n-1})$

TFQMR performs 2 matrix-vector multiplications (with A). 8 vectors of storage are required.

TFQMR does not calculate or update a residual explicitly; however, Freund gives an upper bound for the kth residual given by:

$$\|r_k\|_2 \leq \tau_k\sqrt{k+1}$$

This is very similar to that for QMR and shows that the two methods can be expected to show similar convergence behaviour.

Like CGS, TFQMR can break down unless look-ahead steps are incorporated into the algorithm.

4.5 Decomposition-based techniques

4.5.1 Principles of Decomposition-based Techniques

Decomposition [Cou85] is a divide-and-conquer technique for simplifying the analysis of complex systems. It involves breaking a complex system up into simpler subsystems, analysing the subsystems individually and then constructing a global solution by analysing how the subsystems interact.

Decompositional techniques are best applied to structures where interactions *within* subsystems are strong and more frequent than interactions *between* subsystems. Such systems consisting of loosely-coupled nearly-independent subunits are referred to as being *nearly completely decomposable* (NCD). The analysis of NCD systems is based on the ideal assumption that interactions *within* subsystems can be analysed without reference to interactions *between* subsystems and vice versa. In practice, this assumption is hardly ever met exactly, so decomposition will only yield approximate results. Iterative decompositional techniques are therefore often used to reduce the error in the results to an acceptable level by successive approximations.

In the context of solving large-scale Markov chains, a chain is referred to as NCD if its states can be partitioned into disjoint subchains, with strong interactions among the states of a subchain but with weak interactions among the subchains themselves [Ste94]. Given this structure, the probability transition matrix P of an NCD chain can be partitioned into block form as follows:

$$P = \begin{pmatrix} P_{11} & P_{12} & \cdots & P_{1N} \\ P_{21} & P_{22} & \cdots & P_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ P_{N1} & P_{N2} & \cdots & P_{NN} \end{pmatrix} \tag{50}$$

where the magnitudes of the elements in off-diagonal blocks (which represent the interactions between subchains) are assumed to be small in comparison to the magnitude of elements in the diagonal blocks (which represent interactions within subchains).

We wish to solve for the steady state probability vector π given by

$$\pi = \pi P \quad \text{subject to} \quad \|\pi\|_1 = 1.$$

Partitioning the steady state vector according to the block structure of Eq. (50), i.e. $\pi = (\pi_1, \pi_2, \ldots, \pi_N)$, we can obtain an approximate solution for the steady state probability vector π by ignoring the off-diagonal blocks and solving for the steady state distribution of each diagonal block P_{ii}. However, we cannot solve directly for

$$\pi_i = \pi_i P_{ii}$$

since each P_{ii} is a substochastic matrix. Instead, we take the normalised eigenvector u_i corresponding to the Perron root λ_i (the eigenvalue closest to one) as the probability vector of block i. That is, for each block i we solve for u_i in

$$u_i P_{ii} = \lambda_i u_i \quad \text{subject to} \quad \|u_i\|_1 = 1$$

Each u_i is a conditional probability vector in the sense that u_i is the probability vector of the states within a block, given that the system is in one of the states in the block.

To construct the full steady state solution, we also need to find the probabilities of transitions between blocks; these probabilities are given by the $N \times N$ aggregation matrix A with entries

$$a_{ij} = \phi_i P_{ij} e$$

where $e^T = (1, 1, \ldots, 1)^T$ and

$$\phi_i = \frac{\pi_i}{\|\pi_i\|_1}$$

Unfortunately, the exact steady state distribution π is not known; however, we can estimate ϕ_i by:

$$\phi_i \approx \frac{u_i}{\|u_i\|_1}$$

Given an irreducible stochastic matrix P partitioned as in Eq. (50), the resulting aggregation matrix A will also be stochastic and irreducible [Ste94, pg. 290–291]; thus A has a unique steady-state solution ξ given by:

$$\xi = \xi A \quad \text{subject to} \quad \|\xi\| = 1$$

Finally, the approximate stationary probability distribution can be calculated as:

$$\hat{\pi} = (\xi_1 u_1, \xi_2 u_2, \ldots, \xi_N u_N)$$

This approximation procedure can be transformed into an iterative algorithm having the general form given below [Ste94]:

1. Initialise $\pi^{(0)} = (\pi_1^{(0)}, \pi_2^{(0)}, \ldots, \pi_N^{(0)})$ as the initial approximation to π and set the iteration number $m = 1$.

2. For $i = 1, 2, \ldots, N$ set
$$\phi_i^{(m-1)} = \frac{\pi_i^{(m-1)}}{\|\pi_i^{(m-1)}\|}$$

3. Construct the $N \times N$ aggregation matrix $A^{(m-1)}$ with elements:
$$(A^{m-1})_{ij} = \phi_i^{(m-1)} P_{ij} e$$

4. Determine the steady state distribution $\xi^{(m-1)}$ of matrix $A^{(m-1)}$ by solving:
$$\xi^{(m-1)} A^{(m-1)} = \xi^{(m-1)} \quad \text{subject to} \quad \|\xi^{(m-1)}\|_1 = 1$$

5. Use a block Gauss-Seidel operation to compute a new approximation to the steady state distribution $\pi_i^{(m)}$ using $\phi^{(m-1)}$, $\xi^{(m-1)}$ and the blocks of matrix P.

6. Test $\pi^{(m)}$ for convergence. If $\pi^{(m)}$ has not converged, set $m = m + 1$ and go to step 2. Otherwise $\pi^{(m)}$ is the solution vector.

There are many such iterative aggregation-disaggregation (IAD) algorithms in the literature [CS85, CS84, KMS84, Sch86], all of which are closely related to one another. Two of the most popular IAD algorithms are the KMS (Koury, McAllister and Stewart) algorithm [KMS84] and the Takahashi IAD algorithm [Ste94, pg. 314-315]. These two methods essentially differ from one another in their 5th step, i.e. how the new approximation to the steady state distribution is computed. The KMS algorithm uses a block Gauss-Seidel operation which involves solving for $\pi^{(m)}$ from the set of equations given by

$$\pi_k^{(m)} = \pi_k^{(m)} P_{kk} + \sum_{j<k} \pi_j^{(m)} P_{jk} + \sum_{j>k} z_j^{(m)} P_{jk}$$

where $k = 1, 2, \ldots N$ and

$$z^{(m)} = (\xi_1^{(m-1)} \phi_1^{(m-1)}, \xi_2^{(m-1)} \phi_2^{(m-1)}, \ldots, \xi_N^{m-1} \phi_N^{(m-1)}).$$

The Takahashi algorithm is based on the idea of isolating each block k and lumping all states outside the block into a single state. Here the 5th step involves solving for z_k in

$$z_k^{(m)} = z_k^{(m)} P_{kk} + \sum_{j<k} \xi_j^{(m-1)} \phi_j^{(m)} P_{jk} + \sum_{j>k} \xi_j^{(m-1)} \phi_j^{(m-1)} P_{jk},$$

where $k = 1, 2, \ldots N$ and

$$\phi_k = \frac{z_k^{(m)}}{\|z_k^{(m)}\|_1}$$

Then $\pi^{(m)}$ can be computed as

$$\pi^{(m)} = (\xi_1^{(m-1)} \phi_1^{(m)}, \xi_2^{(m-1)} \phi_2^{(m)}, \ldots, \xi_N^{(m-1)} \phi_N^{(m)}).$$

4.5.2 Aggregation-Isolation algorithm

Abderezak Touzene's Aggregation-Isolation (AI) algorithm [Tou95] and its relaxed variant Aggregation-Isolation Relaxed (AIR) are recent algorithms for solving large-scale Markov Chains. AI and AIR are based on decompositional techniques, which makes them suited to solving NCD chains. However, they are also applicable to solving general chains.

The AI and AIR algorithms have two characteristics which distinguish them from the host of other steady state solution methods:

- **Low memory requirements**. Given n states, the algorithm requires storage for the one-step transition matrix P and only 3 n-vectors, compared to storage for P and between 6 and 12 n-vectors for conjugate gradient-like methods. Furthermore, the algorithm can be implemented using only column access to P (row access is not required); this allows for the use of a space-efficient sparse matrix representation.

- **Good convergence behaviour**. AI exploits the advantages offered by decompositional techniques and the so-called "Gauss-Seidel" effect (making use of values as soon as they are available) to achieve rapid smooth convergence. AIR also takes advantage of overrelaxation techniques to further accelerate the convergence. Numerical experiments show that AI and AIR are competitive with and often outperform even the best classical and Krylov subspace methods.

Given n states, each iteration of the AI algorithm consists of $n-2$ steps. We consider the general step i:

1. The states of a Markov chain are partitioned into three classes: a left macrostate (L) consisting of states $(1, 2, \ldots, i)$, a single "isolated" state $(i+1)$ and a right macrostate R consisting of states $((i+2), \ldots, n)$. These three state classes are used to form a 3×3 aggregration matrix A giving the transition probabilities between these classes. This transition probability matrix takes the form:

$$A = \begin{array}{c} \\ (L) \\ (i+1) \\ (R) \end{array} \begin{array}{c} \begin{array}{ccc} (L) & (i+1) & (R) \end{array} \\ \left(\begin{array}{ccc} 1-a-c & c & a \\ d & p_{i+1,i+1} & b \\ f & e & 1-e-f \end{array} \right) \end{array}$$

This system is completely specified by the six parameters a, b, c, d, e and f where:

- a is the transition probability from (L) to (R),
- b is the transition probability from $(i+1)$ to (R),
- c is the transition probability from (R) to $(i+1)$,
- d is the transition probability from $(i+1)$ to (L),
- e is the transition probability from (R) to $(i+1)$,
- f is the transition probability from (R) to (L).

Calculating these system parameters from scratch at every step i would be a tedious task involving much computation. This turns out to be unnecessary since we can take advantage of the coherence both between iterations and within steps. In the following, we will use subscripted variables such as $a_i, b_i, c_i, d_i, e_i, f_i$ to indicate the value of the system parameters at step i.

The values of b_i and d_i are given by

$$b_i = \sum_{k=i+2}^{n} p_{i+1,k} \quad \text{and} \quad d_i = \sum_{k=1}^{i} p_{i+1,k} = 1 - p_{i+1,i+1} - b_i$$

Note that b_i and d_i remain constant through iterations since they do not depend on π; they can thus be calculated once and then stored in a vector b.

The values of c_i and e_i are easily computed as:

$$c_i = cc_i/(l_{i-1} + \pi_i) \quad \text{and} \quad e_i = ee_i/(l_{i-1} - \pi_{i+1})$$

where

$$cc_i = (\sum_{j=1}^{i} \pi_j p_{j,i+1}) \quad \text{and} \quad ee_i = (\sum_{j=i+2}^{n} \pi_j p_{j,i+1})$$

The values of a_i and f_i are more complex and are given by

$$a_i = aa_i/(l_{i-1} + \pi_i) \quad \text{and} \quad f_i = ff_i/(u_{i-1} - \pi i + 1)$$

where

$$\begin{aligned} aa_i &= \left(\sum_{j=1}^{i-1} \sum_{k=i+2}^{n} \pi_j p_{jk} \right) + \sum_{k=i+2}^{n} \pi_i p_{ik} \\ ff_i &= \left(\sum_{j=i+1}^{n} \sum_{k=1}^{i} \pi_j p_{jk} \right) - \sum_{k=1}^{i} \pi_{i+1} p_{i+1,k} \end{aligned}$$

Touzene has derived simple update formulas which allow aa_i and ff_i to be expressed in terms of the previous step's parameters as follows:

$$aa_i = aa_{i-1} - cc_i - \pi_{i-1} b_{i-1} \quad \text{and} \quad ff_i = ff_{i-1} - \pi_{i+1} d_i + ee_{i-1}$$

For the full derivation of these update formulas, see [Tou95].

2. Once the system parameters have been calculated, the aggregation matrix A is then solved for its steady state distribution to determine:

   - l_i, the approximate steady-state probability of being in the left macrostate (L),
   - π_{i+1}, the approximate probability of being in state $(i+1)$,
   - u_i, the approximate probability of being in the right macrostate (R).

   The values of l_i, π_{i+1} and u_i are determined from solving:

$$\begin{pmatrix} l_i & \pi_{i+1} & u_i \end{pmatrix} = \begin{pmatrix} l_i & \pi_{i+1} & u_i \end{pmatrix} A \quad \text{subject to} \quad l_i + \pi_{i+1} + u_i = 1$$

   In his paper, Touzene does not dictate what method should be used to solve this system. Since the system is so small, a direct method such as Gaussian elimination or Grassmann's method is appropriate. In fact, using Grassmann's method leads to this accurate subtraction-free algorithm:

$$\begin{aligned}
a' &= a/(e+f) \\
b' &= b/(e+f) \\
\pi'_{i+1} &= (c + a' * e)/(d + b' * f) \\
u' &= a' + \pi_{i+1} * b' \\
l &= 1/(1 + \pi'_{i+1} + u') \\
\pi_{i+1} &= \pi'_{i+1} * l \\
u &= u' * l
\end{aligned}$$

   To make this solution as fast as possible in real implementations, a, b, c, d, e and f can be stored in registers and the reduction can be carried out in place.

3. Set $i = i+1$ and go to step 1. In the next step, the state $(i+1)$ will be absorbed into (L) and state $(i+2)$ will be removed from (R) and isolated.

This general step can now be incorporated into an iterative algorithm:

1. Initialise

   - Compute vector b
   - Choose initial probability vector $\pi^{(0)}$

- Set $m = 1$

2. Iterate

   (a) First step ($i = 1$):

   - Isolate $(L), (2), (R)$ with $(L) = 1$ and $(R) = (3 \dots n)$
   - Compute the parameters of the 3×3 aggregation matrix. Shortcut parameter calculations for the first step are:

$$c_1 = p_{12}, \;\; a_1 = \sum_{k=3}^{n} p_1 k \quad \text{and} \quad f_1 = \frac{\sum_{k=3}^{n} \pi_k^{(m-1)}}{1 - \pi_1^{(m-1)} - \pi_2^{(m-1)}}$$

   - Solve for $l_1 = \pi_1^{(m)}, pi_2^{(m)}, u_1$

   (b) General step ($i = 2, \dots, n-3$)

   - Set $u_i = u_i - \pi_{i+1}^{(m)}$
   - Isolate $(L), (i+1), (R)$
   - Compute the parameters of the 3×3 aggregation matrix
   - Solve for $l_i, \pi_{i+1}^{(m)}, u_i$
   - Set $l_i = l_i + \pi_{i+1}^{(m)}$

   (c) Last step ($i = n-2$)

   - Isolate $(L), (n-1), (n)$
   - Compute the parameters of the 3×3 aggregation matrix. Shortcut parameter calculations for the last step are

$$e_{n-1} = p_{n,n-1} \quad \text{and} \quad f_{n-1} = 1 - p_{n,n} - e_{n-1}$$

   - Solve for $\pi_{n-1}^{(m)}$ and $\pi_n^{(m)}$

   (d) Normalise $\pi^{(m)}$ so that $\|\pi^{(m)}\|_1 = 1$ and test $\pi^{(m)}$ for convergence. If $\pi^{(m)}$ has converged then stop, else set $m = m + 1$ and iterate

The AI algorithm can be adapted to incorporate an SOR-like relaxation step of form:

$$\pi_i^{(m)} = \omega \pi_i^{(m)} + (1 - \omega) \pi_i^{(m-1)} \quad \text{for} \quad i = 1, \dots n$$

where $1 \leq \omega < 2$. The method is then known as Aggregation-Isolation Relaxed or AIR; setting $\omega = 1$ corresponds to the straightforward AI algorithm. Unfortunately, just as is the case for SOR, there is no known way of calculating the optimal value of ω in the general case.

Touzene recommends a simple adaptive scheme were ω is set to 1 and then increased in small steps (say 0.01). Every few iterations, the convergence rate is checked to see if the new value of ω is an improvement. If so, ω is further increased; otherwise ω is set to:

$$\omega = 1 + \frac{(\omega - 1)}{2}.$$

A table-driven scheme can be used to provide a more effective relaxation technique. We maintain a small table T of n entries $T[0], T[1], \ldots, T[n-1]$, where entry $T[k]$ corresponds to the observed improvement in convergence rate obtained using a relaxation parameter value of

$$\omega_k = 1 + k/n.$$

At regular intervals - say the beginning of every sth iteration - the table is used to select the value of ω_k which has yielded the best improvement in convergence rate so far, i.e. we choose

$$\omega_k = 1 + k/n \quad \text{with } k \text{ such that} \quad T[k] = \max_{0 \le k \le n} T[k]$$

Then s iterations of algorithm are performed using ω_k as the relaxation parameter, after which the convergence rate at the current iteration m is calculated as

$$c_m = \frac{\|r^{(m-s)} - r^{(m)}\|_\infty}{\|r^{(m-s)}\|_\infty}$$

where $r^{(m)}$ is the residual vector as calculated at iteration m. $T[k]$ is then updated to reflect the new value of c_m, using an exponentially weighted moving average of form:

$$T[k] = \alpha c_m + (1 - \alpha) T[k]$$

where $0 < \alpha \le 1$.

The algorithm requires a startup phase to seed the table T with initial convergence rates for the entries. Once this has been done, the algorithm is effective at finding values of ω which maintain a good convergence rate.

Chapter 5

Interface Language Specification

5.1 Introduction

An interface language for a Markov chain analyser must meet design criteria relating to *power of expression* and *ease of use*. By examining the facilities provided by existing Markov chain analysers (such as USENUM [Scz87, MCS88, SMC90] and MARCA [Ste91, KS95]) and by considering the needs of likely user applications (such as the Petri net tool DNAnet [ABK95]), an interface language has been designed to meet the following general requirements:

- There should be a flexible high-level *model description* which can be used by a *state space generator* as a basis for constructing a Markov chain; this model description should be powerful enough to support a variety of formalisms such as Generalised Stochastic Petri nets, queueing networks etc. The language given here meets this requirement by making use of general C/C++ constructs for the description of model components which govern the generation and solution processes.

- It should be possible to verify *functional* properties which should hold on the model. The interface language allows the user to specify functional properties which should be checked during the state generation process, such as system invariants and the existence of deadlocks.

- There should be provision for a variety of *performance* results; these include *state measures* which compute the value of a real expression at every state (such as buffer occupancy), and *count measures* which measure the occurrence rate of events (such as transition throughput). The language enables the user to specify both types of performance result using the power of general C/C++ expressions.

- The user should have control of aspects of the state space generation process and the solution process used to find results. The language makes provision for user guidance of both the state generation and steady state solution processes and also allows the user to select the desired level of feedback.

- The language should use concepts and constructs likely to be familiar to target users. Since the language presented here has a simple TEX-like syntax and uses elementary C/C++ expressions, it should be familiar to users in academic environments.

Note that, while some syntax checking can be done during parsing of the input, the syntax of C/C++ expressions etc. can only be checked by the C++ compiler when an attempt is made to compile the self-analysing C++ file generated by the parser.

The following symbols are used in the definition:

`{ X }*`	denotes one or more occurrences of X
`|`	separates alternatives

As in TEX, comments begin with `%`; the remainder of the input line is ignored.

5.2 Language elements

5.2.1 Model Description

The underlying Markov chain of a system is likely to involve many thousands of states and transitions. To avoid explicit enumeration of these states and transitions, a high-level model description is necessary. This model description specifies the components of a general *state* of the system, the conditions on and effects of *transitions* between states and an *initial state* of the system.

A Markov chain *generator* maps this high-level model description onto a low-level system representation consisting of the state space and transitions between states.

```
model_description = \model {
  {
    state_vector | initial_state | transition_declaration |
    constant | help_value | invariant | state_output_function |
    primary_hash_function | secondary_hash_function | additional_headers
  }*
}
```

State Descriptor Vector

The *state descriptor vector* consists of discrete components which, when taken together, describe a state of the system; each unique assignment to these components corresponds to one state.

An arbitrary vector of elementary C++ variables (`int`, `long`, `short`, `char` etc.) is ideal for this purpose; note that elements with `float` or `double` types are not allowed since the state space must be discrete. Variables are declared just as they are in C/C++:

```
state_vector = \statevector{
  { <type> <identifier> {, <identifier> }*; }*
}

type = basic C/C++ variable type;
identifier = valid C/C++ identifier;
```

Initial State

An initial state must be specified for reachability analysis purposes; this can be done using simple C/C++ assignments to the elements of the state vector.

```
initial_state = \initialstate {
  { <assignment> }*
}

assignment = C/C++ assignment to elements of the state vector
```

Transition declarations

Transitions describe how the system moves from state to state (via updates to the current state vector). Since it would be virtually impossible to enumerate successor transitions for every individual reachable state, a more general scheme (similar to USENUM) is used. Possible transitions from the current state are specified by describing:

- one or more *enabling conditions* involving elements of the state vector corresponding to the *current state*.
- an *action* to be taken if the transition is executed; this will involve an assignment to the state vector elements of the *next state*.
- an indication of whether the transition from the current to the next state is *timed* or *instantaneous* (i.e. the transition takes no time to execute).
- a *rate* (for timed transitions) or *relative weight* (for instantaneous transitions) should also be specified; note that these rates and weights may be denoted by (possibly state-dependent) arbitrary expressions. If a non-positive rate is encountered during state exploration, it will be ignored during analysis.
- an optional *priority* which allows transitions of a higher priority to preempt lower priority transitions of the same type (i.e. timed or instantaneous).

Transitions from the *current* to the *next* state descriptor vector can be achieved through C/C++ assignment statements, while enabling conditions can be given using C/C++ boolean expressions. Since the conditions and actions will form part of transition code encapsulated in a C++ `State` object, elements of the *current* state descriptor (as declared in Sec. 5.2.1) can be referred to directly while elements of the *next* state descriptor can be accessed via a `next` pointer.

```
transition_declaration = \transition{<identifier>}{
  \condition{<boolean expression>}
  \action{ { <assignment> }* }
  \rate{<real expression>} | \weight{<real expression>}
  \priority{<non-negative integer>}
```

```
}
```

```
boolean expression = C/C++ boolean expression
real expression = C/C++ real expression
assignment = C/C++ assignment
```

Constants and Help Values

It is convenient to allow for *constant* declarations and complicated formulae which are used repeatedly during the evaluation of transition conditions and rates/weights. Such values are called *help values*; this is a concept adopted from USENUM.

```
constant = \constant{<identifer>}{value}
```

```
help_value = \helpvalue{<type>}{<identifier>}{<expression>}
```

Invariants

Depending on the application domain, there may be invariant conditions which should not be violated during the generation of the state space; these invariant conditions can be expressed as C/C++ expressions. The state generator will issue a warning if it encounters any state which violates an invariant.

```
invariant = \invariant{<expression>}
```

Custom state output function (optional)

If a deadlock or a violation of a user-specified invariant occurs, the state generator reports the event and outputs the state responsible for the error. A simple default output function is provided; however, the user can also provide his/her own output function if desired.

```
state_output_function = \output {
  { <statements> }*
}
```

```
statements = C++ statements to output elements of the state vector
```

Custom state hash functions (optional)

The state generator uses a method of probabilistic state space storage which requires the computation of two hash keys for each state. The primary hash key is a 14-bit positive integer (from 0 to 16383), while the secondary hash key is a 32-bit integer. The functions which perform the key computations should be designed such that, in the event of a primary key collision, a secondary key collision is very unlikely. Default functions are provided, but the user may wish to use application-specific knowledge to write better functions.

```
primary_hash function = \primaryhash {
  <C++ function body returning an integer from 0 to 16383>
}

primary_hash function = \secondaryhash {
  <C++ function body returning a 32-bit integer>
}
```

Additional headers

Should the user require any C/C++ functions which are not usually included by default (such as the advanced mathematical functions to be found in `math.h`), the necessary `#include` statements can be placed in a `header` declaration. Class definitions of user-defined classes can also be placed here.

```
additional_headers = \header {
  <C++ include statements and/or class definitions>
}
```

5.2.2 Generation Control

The user is able to control aspects of the state generation process, such as the *maximum number* of states to be generated or the *maximum cpu time* that should be spent on the generation. The user can also specify the level of feedback by specifying the *report style* and the *report interval*.

```
generation_control = \generation {
  { \maxstates{<long int>} | \maxcputime{<seconds>} |
    \reportstyle{full | short | none} | \reportinterval{<long int>}
  }*
}
```

5.2.3 Solution Control

Once the state space has been generated (using the model description), the resulting state transition matrix must be solved for its steady state distribution. The user is able to guide this steady state solution process through parameters such as:

- *Solution Method.* Possible solution methods include:
  - Direct Methods (Gaussian Elimination, Grassmann)
  - Classical Iterative Methods (Gauss-Seidel, fixed SOR, dynamic SOR)
  - Krylov Subspace Techniques (BiCG, CGNR, CGS, BiCGSTAB, BiCGSTAB2, TFQMR)
  - Decomposition-based Methods (AI (Aggregation-Isolation), AIR)

  Choice of which algorithm to use will depend on the characteristics of the generator matrix Q e.g. for very small state spaces direct methods are generally more efficient than iterative methods, while decompositional methods are useful when the Markov chain is nearly completely decomposable (NCD). An automatic algorithm selection (based on the number of states in the model) is also available.

- *Accuracy.* This specifies the convergence criterion for the iterative methods. These methods will terminate after i iterations with an "accuracy" of ϵ if:

$$\frac{\|x^{(i)} - x^{(i-k)}\|_\infty}{\|x^{(i)}\|_\infty} < \epsilon$$

where k depends on the particular method and ϵ can vary between 10^{-2} and $2.22045 * 10^{-16}$ (IEEE-754 machine epsilon for double precision). Reported performance results are rounded to reflect this accuracy.

- *Maximum Iterations* (within which iterative methods should converge)
- *Relaxation Parameter* (SOR). Parameter estimation can be either fixed or dynamic.
- *Start Vector* (useful when performing a sequence of experiments)

As with the generation of the state space, the user is able to set the required level of reporting feedback.

```
solution_control = \solution {
  { \method{gauss | grassman | gauss_seidel | sor | bicg | cgnr |
        bicgstab | bicgstab2 | cgs | tfqmr | ai | air | automatic} |
    \accuracy{<real>} |
    \maxiterations{<long int>} |
    \relaxparameter{<real> | dynamic} |
    \startvector{ <filename> } |
    \reportstyle{full | short | none} |
    \reportinterval{<long int>}
  }*
}
```

5.2.4 Performance Measures/Results

Performance results provide a backward mapping from low-level results like probabilities of states and rates of transitions to higher-level quantities like throughput or mean buffer occupancy. Performance measures can generally be classified as *state* or *count* measures; the concept of state and count measures originated in the HIT-tool [BS87] and has been adopted by other tools such as USENUM.

```
\performance_measures = \performance {
  { state_measure | count_measure }*
}
```

State measures

A state measure is used to determine the mean and variance of a real expression which is defined at every state in the system. e.g. the average number of tokens on a particular place of a Petri net or some transition's enabling probability. The mean, variance, standard deviation and distribution of state measures can be computed.

```
state_measure = \statemeasure{identifier}{
  \estimator{ {mean | variance | stddev | distribution}* }
  \expression{<real_expression>}
}
```

Count measures

A count measure is used to determine the mean rate at which a particular event occurs e.g. the rate at which a transition fires yields transition throughput.

The occurrence of an event is specified according to three conditions:

- a *precondition* on the current state that must be true.
- a *postcondition* on the next state that must be true.
- *transitions* which must be fired during the transition from the current to the next state.

The conditions can be specified as C++ expressions while the transitions can be given in a list. Note that only the mean of count measures is available, since computation of higher moments requires transient analysis.

```
count_measure = \count_measure{identifer}{
  \estimator{mean}
  \precondition{<boolean expression>}
  \postcondition{<boolean expression>}
  \transition{ all | {<identifier>}* }
}
```

5.2.5 Output options

Besides the desired performance results, the user is able to control the level of intermediate output detail, including a list of the final states with their steady state probabilities and the generator matrix Q.

```
output_options = \outputoptions {
  { \statelist{<filename>} |
    \steadystatevector{<filename>} |
    \transitionmatrix{<filename>} |
    \performanceresults{<filename>}
  }*
}
```

Chapter 6

The DNAmaca Performance Analyser

6.1 Introduction

The concepts discussed in the previous chapters have been implemented in the DNAmaca (pronounced *"dee-nam-ack-a"*) performance analyser. DNAmaca provides a complete performance analysis sequence including model specification, state space generation, functional analysis, steady state solution and the computation of performance statistics.

6.2 DNAmaca Components

Fig. 15 illustrates the major modules of DNAmaca. Control is passed from module to module as follows:

- The **parser** translates the user's high-level model description into a C++ class which describes the same model.
- The C++ class is then compiled and linked with common library routines to form a standalone **state space generator** for the model. The state space generator uses a probabilistic exploration algorithm incorporating on-the-fly vanishing state elimination to generate all reachable tangible states. The infinitesimal generator matrix Q

Figure 15: Main Modules of the DNAmaca Markov chain analyser

which describes the transition rates between tangible states is also generated during this process.

- The **functional analyser** examines the state transition matrix Q to check if the Markov Chain is irreducible, i.e. if the states form a single strongly connected component. If the chain is irreducible, it is possible to solve for its stationary distribution and control is passed to the **steady state solver**.

- Given a chain of n states, the **steady state solver** determines the stationary distribution $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ by solving the set of n steady state equations given by

$$\pi Q = 0 \quad \text{subject to} \quad \sum_{i=1}^{n} \pi_i = 1.$$

- Finally, the user code is linked with common library routines to form a **performance analyser**. The performance analyser uses the steady state solution in combination with state space information to produce performance results.

The following sections describe each component of DNAmaca in detail.

6.2.1 The Parser

A simple recursive descent parser [ASU86] implements the interface language described in Chapter 5. The parser accepts a user data file containing:

- A **model description** including the format of the state descriptor, an initial state and rules governing transitions between states.

- A description of **performance statistics** to be computed in the form of state or count measures.

- User **options** relating to state space generation and steady state solution, such as desired accuracy or choice of solution method.

If there are no syntactic errors, the backend of the parser generates the user code necessary for state space exploration and performance analysis. The user code is encapsulated in a

C++ `State` class. This mechanism ensures that every model presents a uniform high-level interface to external program modules such as the state space generator and performance analyser. In particular, the generated `State` class includes methods to:

- setup the current state as the **initial** state of the system.
- determine the set of **enabled transitions** at the current state.
- **fire** any enabled transition to determine the successor states of the current state.
- determine the (possibly state-dependent) **rate or weight** of any transition.
- compute the current state's primary and secondary **hash keys** needed by the probabilistic dynamic hash compaction technique (cf. Chapter 3).
- check that any user-specified **invariants** apply to the current state.
- compute **performance statistics** for the current state in the form of state and count measures.

Since this high-level interface does not change from model to model, the relatively small amount of code found in the model-specific `State` class can now be compiled and linked with pre-compiled external modules to produce a state space generator and a performance analyser for the model. This reduces compilation time considerably.

6.2.2 The State Space Generator

The probabilistic dynamic hash compaction technique described in Chapter 3 has been implemented in DNAmaca as a `Generator` class. The `Generator` class interfaces with the user code through the `State` class and contains three main data structures:

- A **hash table** which is used to store and to search for states according to their primary and secondary hash keys. The hash table has 2^{14} rows and uses 32-bit secondary keys.
- A **tangible state stack** for storing unexplored tangible states.
- A **vanishing state stack** for temporary storage of vanishing states during vanishing state elimination.

The `Generator` class also contains an `explore()` method which implements the state space exploration algorithm given in Fig. 11.

We will demonstrate the effectiveness of this state space generation technique using the *benchprod* Stochastic Petri Net model shown in Fig. 16 [CCM95]. *Benchprod* is a scalable model of the Oki Electric Company (Japan) production line and is a popular test case for state exploration algorithms.

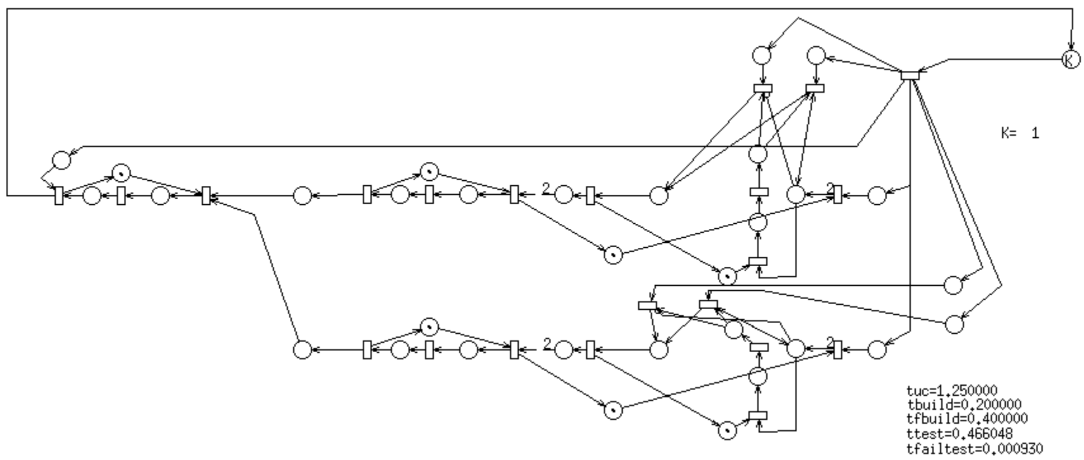
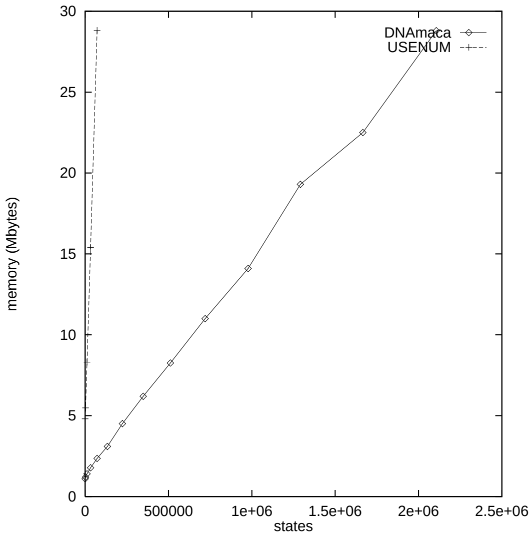


Figure 16: The *benchprod* scaleable stochastic Petri net model

The graph on the left of Fig. 17 compares the dynamic probabilistic hash compaction technique used by DNAmaca with that of the exhaustive dynamic storage technique used by the USENUM analyser [Scz87] in terms of memory needed to generate the state space of the *benchprod* model. The results were obtained on a Sun SPARCclassic with 64MB memory and memory utilization was measured with the UNIX *top* utility. The space saving advantages of using a probabilistic technique are clear.

The table on the right of Fig. 17 presents the corresponding state space generation times (CPU and system time, as given by the `clock()` system call) for systems of up to 2 million states. It is interesting to note that, even on a SPARCclassic (a machine only approximately 1.5 times as powerful as a 33MHz 486), our state exploration method outperforms a parallel exploration technique [CCM95] running on a CM-5 with 32 nodes, each of which corresponds to a SPARC2 workstation with 32 Mbytes RAM. For a 511 588 state *benchprod* model, the CM-5 generates the state space at a rate of 1.507 milliseconds per state, while we measured



k	number of states	generation time (seconds)	time/state (millisecs)
1	172	0.12	0.709
2	2359	1.51	0.641
3	11386	8.05	0.707
4	32653	24.17	0.740
5	71560	55.01	0.769
6	133507	108.26	0.811
7	223894	186.01	0.831
8	348121	292.60	0.841
9	511588	442.14	0.864
10	719695	645.47	0.897
11	977842	887.32	0.907
12	1291429	1215.89	0.942
13	1665856	1625.18	0.976
14	2106523	2147.10	1.019

Figure 17: Comparative memory use between exhaustive (USENUM) and probabilistic (DNAmaca) state space exploration techniques (left) and state space generation times for DNAmaca (right) for the *benchprod* model

0.864 milliseconds per state.

DNAmaca also implements on-the-fly elimination of vanishing states, which is of particular use for those variants of stochastic Petri nets which include timeless transitions. We will illustrate the effect of vanishing state elimination using an SDL-net model of the *InRes* (Initiator-Responder) communication protocol [Hog89]. SDL-nets [BKKK95] are a subclass of queueing Petri nets [Bau93], which are themselves coloured Generalised Stochastic Petri nets with special timed and immediate queueing places. Note that it is not necessary for the reader to be familiar with SDL-nets or QPNs to understand what follows; the example is used only to illustrate the effectiveness of on-the-fly vanishing state elimination on a timed transition system representation which supports timeless transitions. Fig. 18 presents an overview of the SDL-net model [Kab95] of the InRes protocol which was constructed using the QPN-Tool [BK94]. The model has 208 702 states, 73 735 of which are tangible and 134 967 of which are vanishing.

Table 4 shows the effect of on-the-fly vanishing state elimination applied to the *InRes* model. As expected, vanishing state elimination leads to a decrease in the number of states generated, memory usage and transition matrix size. There is also an increase in the number of transition firings owing to the repeated exploration of clusters of vanishing

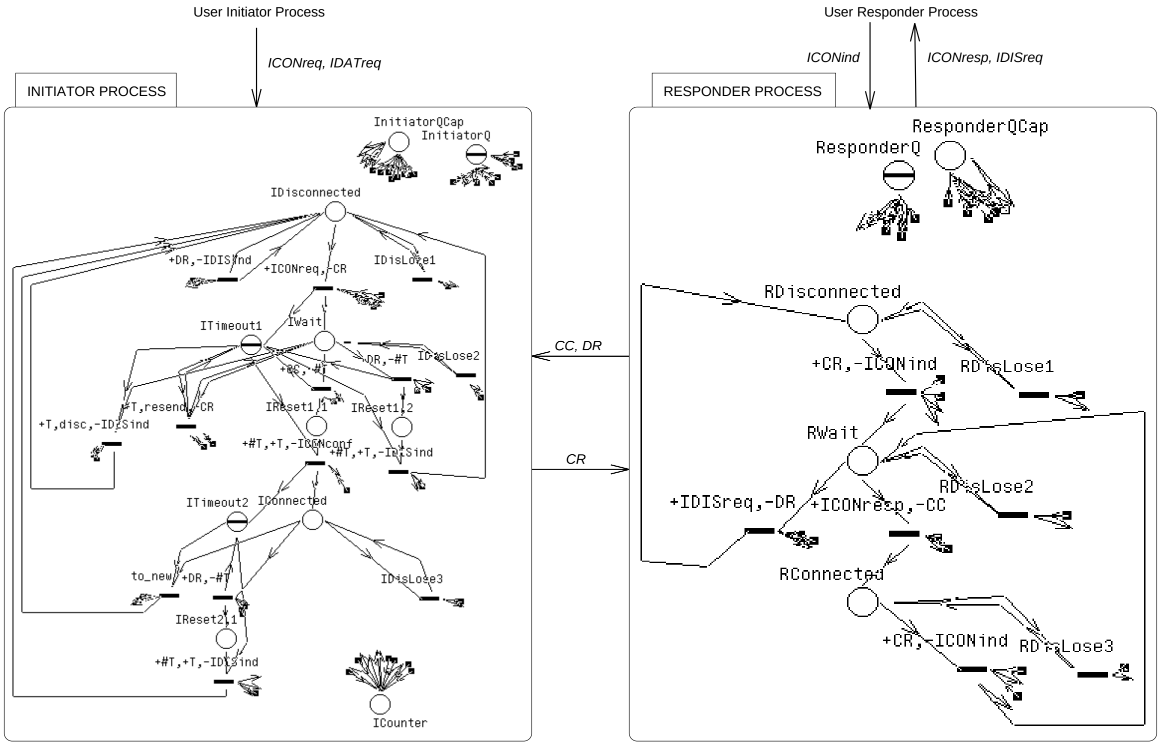


Figure 18: The *InRes* SDL-net model

	Without on-the-fly vanishing state elimination	With on-the-fly vanishing state elimination
States generated	208702	73735
Generation time (CPU seconds)	143.51	132.53
Memory used (KBytes)	4992	2804
Non-zero entries in transition matrix	427651	295571
Transitions fired	427651	520699

Table 4: The effect of on-the-fly vanishing state elimination on the *InRes* queueing Petri net model

states. Remarkably, even though more transitions are fired, the generation time with on-the-fly elimination is less since vanishing states need not be stored in the hash table, nor do they need to be written to secondary storage.

6.2.3 The Functional Analyser

It possible to solve for the stationary distribution of a continuous-time Markov chain if and only if the chain is irreducible, i.e. if every state communicates with every other state (cf. Chapter 2). DNAmaca thus includes a functional analyser which checks that the Markov chain given by the infinitesimal generator matrix Q is irreducible. If the chain is not irreducible, but can be made to be so by eliminating transient states, the analyser performs a remapping of the states.

The analyser uses a strongly connected components algorithm [Baa88, pg. 193–197] to divide the states into recurrent and transient state classes. There are three possible outcomes of the analysis:

- The state graph consists of one recurrent state class only. In this case, the states in Q form a single strongly connected component of states and the Markov chain formed from Q is irreducible. The functional analyser takes no further action.
- The state graph consists of one recurrent state class and one or more transient state classes. In this case, the states in Q consist of several transient states and a single final strongly connected component of states. Since the stationary probability of being in each of the transient states is 0 and the transient states have no effect on the transitions between recurrent states, the transient states may be eliminated from Q. The functional analyser performs this remapping to leave a state graph consisting of 1 recurrent state class only.
- The state graph consists of more than one recurrent state class. In the case, the Markov chain is reducible and it is not possible to solve for the chain's stationary distribution. In this case, the solution process is abandoned.

The strongly connected components algorithm is based on a depth-first search. It has time complexity $O(e)$ where e is the number of edges in the state graph, or, equivalently, the number of non-zero entries in Q. Since the algorithm has linear complexity and the algorithm can be carried out on a copy of Q stored in main memory, the functional analysis phase is substantially faster than the state space generation phase or the steady state solution phase. The space complexity of the algorithm is $O(n+e)$ since space is needed for storing the e non-zero entries in Q, as well as a DFS stack of maximum size n.

6.2.4 The Steady State Solver

DNAmaca implements the steady state solvers described below (cf. Chapter 4):

- **Direct Methods**: Sparse Gaussian Elimination and Grassmann's method. These methods are very accurate but are only suitable for the solution of small models since they have time complexities of $O(n^3/3)$ and $O(2n^3/3)$ respectively. Grassmann's method is thus the default solution method for models of up to 250 states while sparse Gaussian elimination is the default solver for models of up to 500 states.
- **Classical Iterative Methods**: Gauss-Seidel, fixed SOR and dynamic SOR. Dynamic SOR is the most effective of these methods and is used as the default solver for models of up to 20000 states.
- **Krylov subspace techniques**: BiCG, CGNR, CGS, BiCGSTAB, BiCGSTAB2 and TFQMR. CGS is used as the default solver for models of up to 50000 states since it exhibits rapid convergence and has the lowest memory requirements of the methods in this class.
- **Decompositional techniques**: AI and AIR. AIR with table driven relaxation has very low memory requirements and achieves rapid and smooth convergence once its initialisation phase has completed. It is thus the default solver for models with more than 50000 states.

For all of these techniques, it is critical to use an efficient scheme to store the transition matrix Q in memory, since the storage of this matrix usually dominates the space requirements of the entire performance analysis sequence. Two observations are helpful in designing an efficient storage structure:

- Most iterative methods (including SOR, CGS, BiCGSTAB, BiCGSTAB2, TFQMR, AI and AIR) require column access, but not row access, to the transition matrix Q. This is because these methods access the elements of Q in the same fashion as a matrix multiplication operation of the form $Q^T x$ where x is the current solution vector.

- Since transition rates are very often fixed and are seldom state-dependent, many of the entries in Q will have the same value. Furthermore, all iterative methods do not modify the entries of Q. Thus, instead of using a double precision floating point number to denote each entry in Q, a smaller pointer into a table of common transition rates can be used.

Fig. 19 shows DNAmaca's transition matrix data structure for storing the $n \times n$ transition matrix Q, as used by most of DNAmaca's iterative methods. The structure consists of three components:

- A **sparse matrix** consisting of n dynamic vectors. The ith dynamic vector stores the elements found in the ith column of Q.

- A **store** of transition rates. The elements stored in each dynamic vector of the sparse matrix point to entries in this store.

- An **AVL tree** (a height-balanced binary tree) of all the items in the store. As each transition rate entry is added to the sparse matrix, a search mechanism is needed to establish whether the entry is already in the store. An AVL tree is thus maintained to rapidly search for store items; this reduces the search complexity from $O(n)$ for a linear search of the store to $O(\log_2 n)$ for a search of the tree. The AVL tree is destroyed once all items have been inserted into the matrix since it is then no longer needed.

This transition matrix structure has been encapsulated in a `TransitionMatrix` class. For solution methods requiring both row and column access to Q (i.e. BiCG and CGNR), a `TransitionMatrixWithRowColumn` class inherits from the base `TransitionMatrix` class and includes row access information.

An instance of the transition matrix data structure is in turn encapsulated within class known as the `SteadyStateAnalyser` class. This class contains methods to implement the

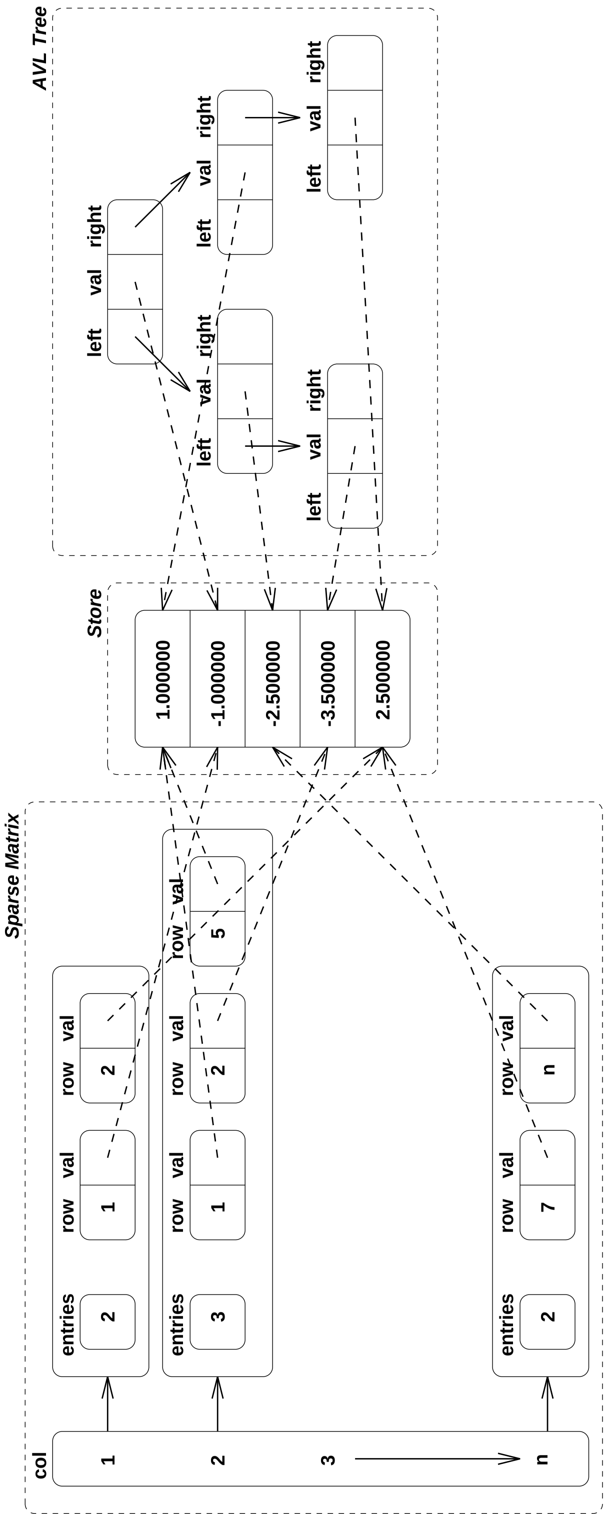


Figure 19: DNAmaca's transition matrix data structure

various steady state solvers. The class also includes a method for verifying that the steady state solutions produced by each of the steady state solvers are plausible. In particular, given a steady state solution vector x, DNAmaca calculates:

- The norm of the final residual vector, as given by $\| - Q^T x\|_\infty$. The closer the norm is to 0, the more accurate the result.
- The sum of the elements of the steady state vector $\sum_{i=1}^{n} x_i$, which should be 1.
- The range of elements in the steady state vector. All elements should lie in the range $0 \leq x_i \leq 1$.

Fig. 20 shows the observed convergence behaviour of several steady state methods for the 73 735 state *InRes* queueing Petri net model of Fig. 18. Notice the fairly smooth but slow convergence of Gauss-Seidel and SOR, the erratic but superlinear convergence of the Krylov subspace methods, the smooth convergence of the AI method and the rapid convergence of the AIR method.

6.2.5 The Performance Analyser

The last stage in the performance analysis sequence is to combine the low-level results given by the steady state distribution and the state space information to form more meaningful higher-level performance measures such as throughput or mean buffer occupancy. DNAmaca includes a performance analyser which calculates two types of performance measures: *state* and *count* measures. The concept of state and count measures originated in the HIT-tool [BS87] and has been adopted by other tools such as USENUM [Scz87].

A state measure is used to determine the mean and variance of a real expression which is defined at every state in the system, e.g. the average number of tokens on a particular place of a Petri net or some transition's enabling probability. The mean, variance, standard deviation and distribution of state measures can be computed. Given n states, the steady state distribution $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ and a vector of expression values $v = (v_1, v_2, \ldots, v_n)$ where v_i is a function of the elements of the state descriptor of state i, the mean of a state measure m can be calculated as:

$$\mathrm{E}[m] = \sum_{i=1}^{n} \pi_i v_i$$

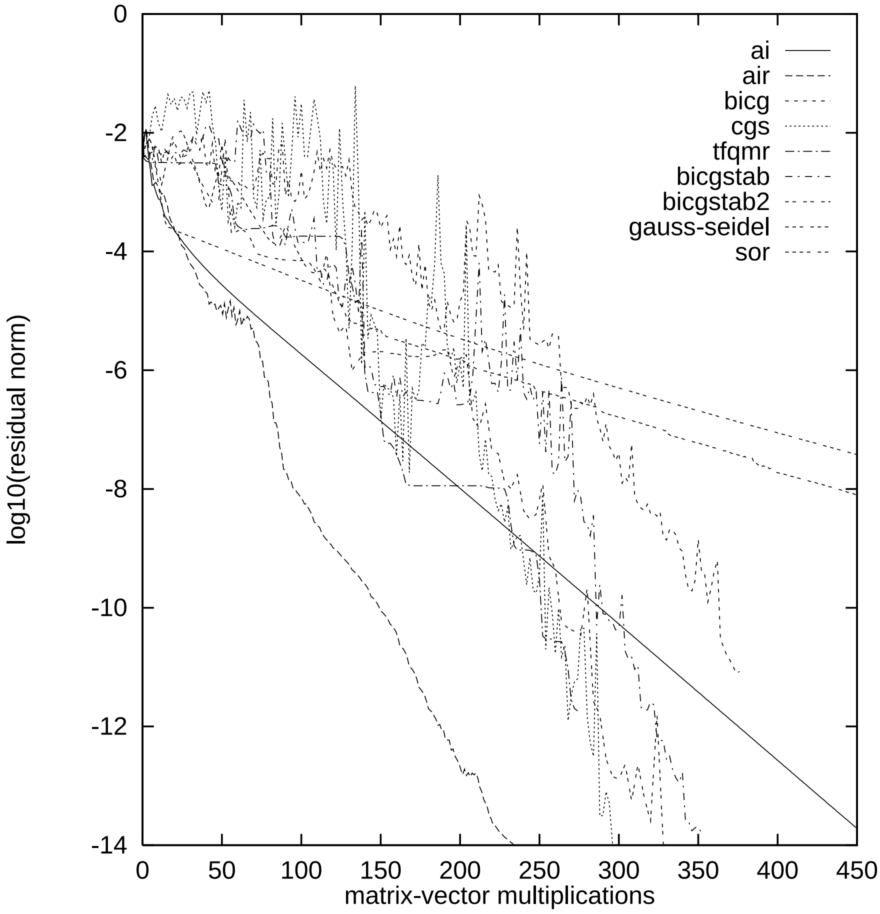


Figure 20: Convergence behaviour of some steady state algorithms for the *InRes* model

We can regard v_i as the ith state's contribution towards the value of the state measure. If, for example, we are interested in the mean number of tokens on a place p in a Petri net, v_i would represent the number of tokens on place p in state i.

The second moment of a state measure m is given by:

$$\mathrm{E}[m^2] = \sum_{i=1}^{n} \pi_i v_i^2$$

and the variance of m by:

$$\begin{aligned}\mathrm{Var}[m] &= \mathrm{E}[m^2] - (\mathrm{E}[m])^2 \\ &= \sum_{i=1}^{n} \pi_i v_i^2 - \left(\sum_{i=1}^{n} \pi_i v_i\right)^2\end{aligned}$$

A count measure is used to determine the mean rate at which a particular event occurs e.g. the mean rate of transition firing gives transition throughput. Given a system with n states, the steady state distribution $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ and a function r_i which returns the rate at which the event occurs when the system is in state i, the mean of a count measure m is given by:

$$\mathrm{E}[m] = \sum_{i=1}^{n} \pi_i r_i$$

The calculation of the variance of a count measure requires transient analysis, which is not supported by DNAmaca.

The computation of performance measures is facilitated in the user code which includes methods for calculating the values of v_i and r_i for each state. The performance analyser reads the steady state distribution into memory and then reads the state space state-by-state, using these methods to calculate the contribution of the state towards each performance measure.

Chapter 7

Example Timed Transition Systems and Solutions

7.1 Introduction

In this chapter we will demonstrate the effectiveness of DNAmaca as a performance analysis tool by applying it to three models of timed transition systems: a queueing model of a multimedia traffic switch, a queueing network model of an interactive computer system and a Generalised Stochastic Petri net model of a telecommunications protocol.

7.2 Multimedia teletraffic switch

The schematic diagram in Fig. 21 is of a multimedia teletraffic switch designed to handle delay-sensitive voice traffic and delay-insensitive data traffic [AK93, pg. 133–137].

The switch has a capacity for s calls and is designed to give priority to voice calls. If the switch is full and the number of data calls in the system exceeds a certain threshold n, an arriving voice call may preempt a data call. If there are less than n data calls and no free circuits in the switch, arriving voice calls will be blocked. Waiting or preempted data calls are stored in a buffer with capacity b.

There are v potential sources of voice calls. Each of these sources is governed by a two-state Markov process which alternates between a silence phase and a talkspurt phase. The mean

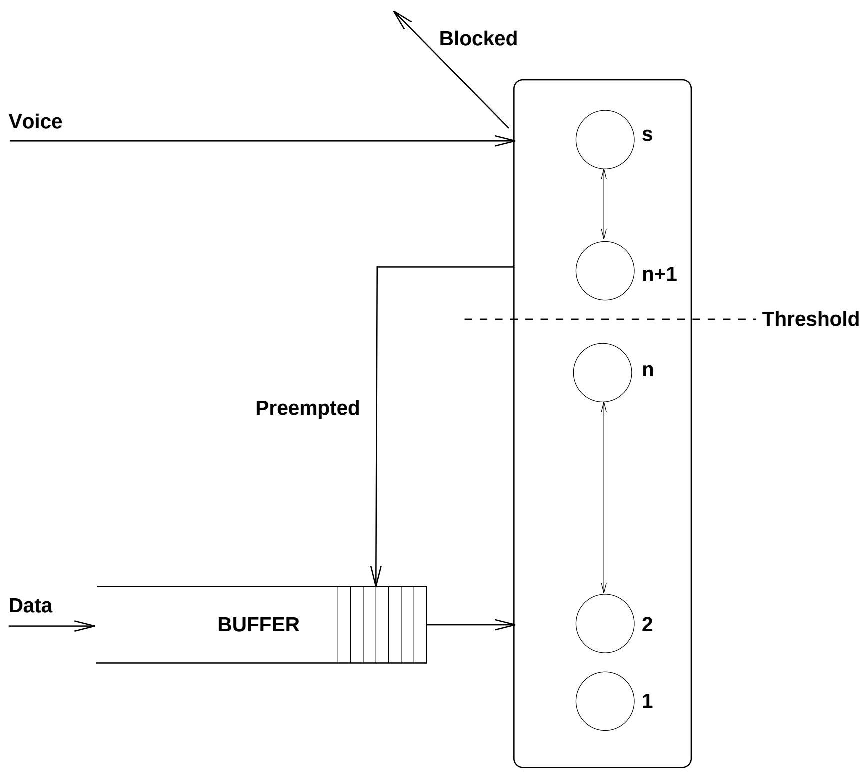


Figure 21: A multimedia switch for handling voice and data traffic

duration of the silence phase is $1/\lambda_1$ and the mean duration of a talkspurt phase is $1/\mu_1$. The data arrival process is simpler, being Poisson with parameter λ_2. Data calls are served at a rate of μ_2 per server.

The performance analysis of this switch, with blocking, preemptive service and a two-phased voice arrival process, is not easily accomplished using conventional queueing theory. Equally, the problem cannot be tackled with a Petri net model or any other such formalism. Thus, apart from simulation, the only other alternative is to model the system as a Markov chain and solve that.

We used DNAmaca to model a switch with capacity $s = 72$ and buffer size $b = 200$. There were $v = 1000$ voice sources, with $\lambda_1 = 0.04$ and $\mu_1 = 1.0$. The data arrival rate was $\lambda_2 = 43.0$, and the data service rate was $\mu_2 = 1.2$ per server. The DNAmaca input file which specifies this model is given in Appendix A.

Three runs were then conducted to test the effect of varying the preemption threshold value n. The first run used a low value of $n = 20$, the second a medium value of $n = 32$ and the last a high value of $n = 50$. Each run generated a Markov chain of 17301 tangible states.

Fig. 22 presents the resulting distributions for the number of voice and data calls in the system and the corresponding distribution for the number of data calls in the buffer for each of the preemption threshold values. The lowest preemption value $n = 20$ allows voice calls to aggressively preempt data calls; thus it makes sense that the mean number of voice calls in the system should be higher than the mean number of data calls. The high level of data call preemption, however, leads to a data call buffer overflow, suggesting that a higher threshold is needed. A preemption value of $n = 32$ leads to a more balanced distribution of voice and data calls, with the number of voice calls dropping off sharply once the preemption limit is reached. The buffer no longer overflows and the distribution of calls in the buffer shows that a buffer size of $b = 150$ should be more than adequate to deal with almost all calls. The highest preemption value of $n = 50$ allows only restricted preemption of data calls and leads to the blocking of a large number of voice calls; consequently the mean number of data calls is higher than the mean number of voice calls. Since data calls are seldom preempted, a buffer size of only about $b = 15$ would be adequate to deal with almost all calls.

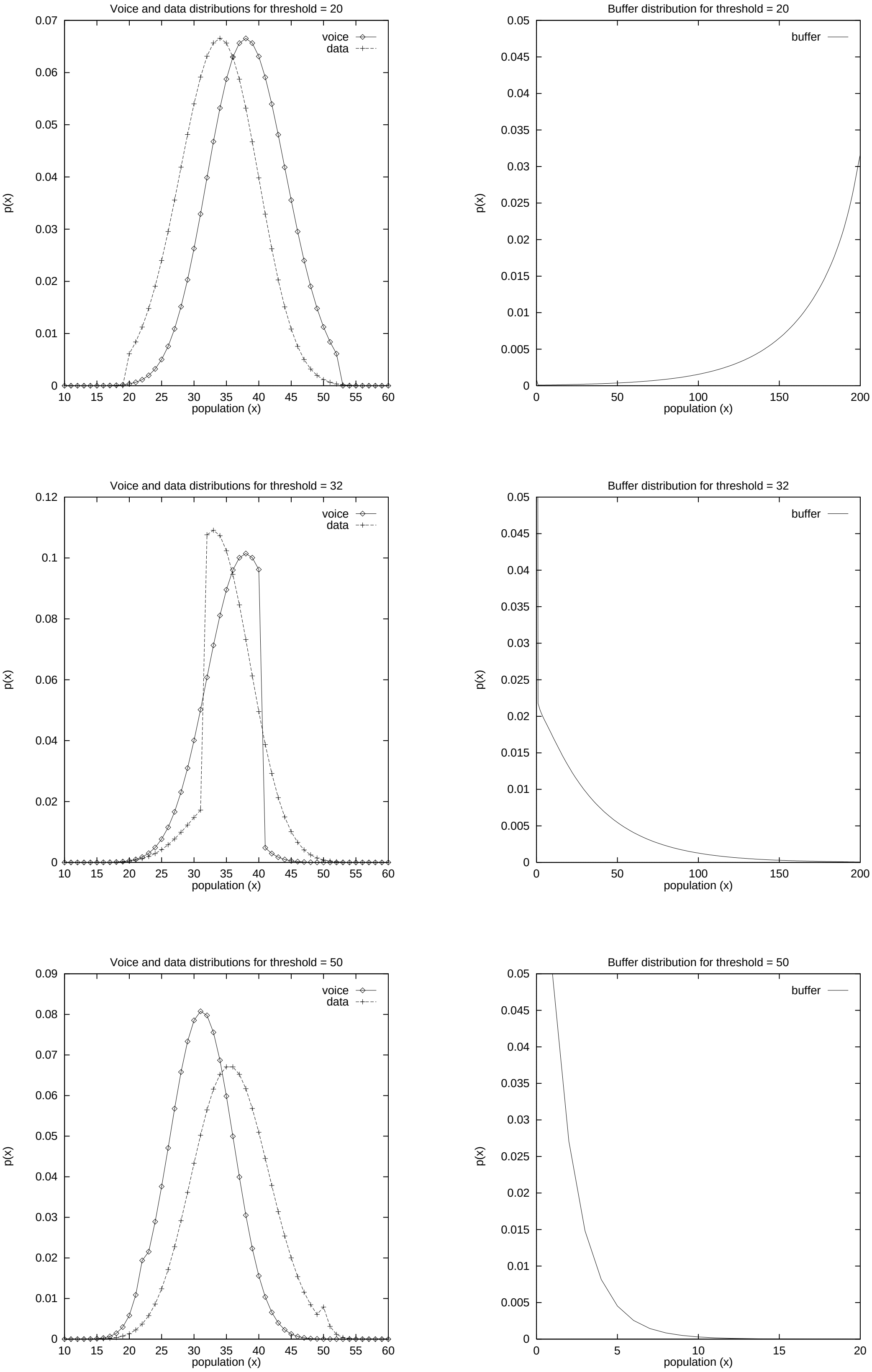


Figure 22: Distributions for the number of voice and data calls in the switch (left) and the number of data calls in the buffer (right) for various threshold values.

7.3 Interactive computer system

Fig. 23 presents a queueing network model of a time-shared interactive computer with a paged virtual memory system. The model is the same as that discussed in Stewart [Ste94, pg. 326–327].

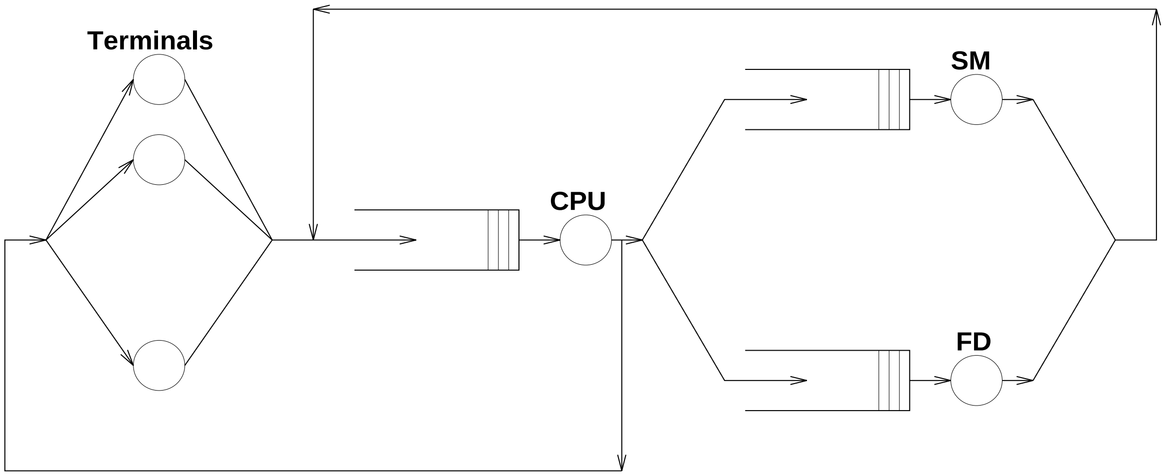


Figure 23: Model of an interactive computer system

The system consists of a set of N terminals, a central processing unit (CPU), a secondary memory device (SM) and a filing device (FD). The CPU, SM and FD each have an associated FCFS queue of pending requests.

Users at the terminals generate jobs which are submitted into the CPU queue for processing. It is assumed that users are inactive between job submissions. Jobs being processed at the CPU may either complete and return to the terminals, or they can be interrupted by an I/O request or a page fault. In the case of an I/O request, the job enters the FD queue and in the case of a page fault, the job enters the SM queue. After completion of service at the SM or FD device, interrupted jobs return to the CPU queue for further processing.

The state descriptor of the system is given by (n_0, n_1, n_2, n_3) where n_0 is the number of idle terminals and n_1, n_2 and n_3 denote the number of jobs in the CPU, SM and FD queues respectively. The total number of jobs executing in the system at any one moment is given by $\eta = n_1 + n_2 + n_3$. Jobs issue I/O requests at a rate given by $r(\eta)$ where $(r(\eta))^{-1}$ is the mean compute time between I/O requests. Similarly, jobs page fault at a rate given by $q(\eta)$ where $(q(\eta))^{-1}$ is the mean time between page faults. The page fault rate of a process executing in memory m is modelled as $q(\eta) = \alpha/m^k$, where α and k are constants which depend on CPU speed, program characteristics and memory management strategy. Here we assume that the main memory of size M is equally shared between the number of processes currently executing in the system, so $q(\eta) = \alpha(\eta/M)^k$. Processes depart from the CPU queue and return to the terminals at a rate given by $c(\eta)$ where $(c(\eta))^{-1}$ is the mean compute time of a process.

While a conventional MVA queueing network analysis of this system fails on account of the state-dependent page fault rate, a Markov chain representation allows this phenomenon to be modelled accurately.

DNAmaca was used to study the effect of varying N and M on the utilisation of the CPU, FD and SM for a systems with particular numerical parameters. The page fault rate $q(\eta)$ was obtained by setting $\alpha = 100$ and $k = 1.5$ so that $q(\eta) = 100(\eta/M)^{1.5}$. The mean time between I/O requests was taken to be 20 milliseconds, so $r(\eta) = 0.05$, and the mean compute time of a process was taken to be 500 milliseconds, so $c(\eta) = 0.002$. The mean think time of a user at a terminal was estimated to be about $\lambda^{-1} = 10$ seconds. The mean service time of the FD was taken as $\mu_1^{-1} = 12$ milliseconds and the mean service time of the SM was taken as $\mu_2^{-1} = 5$ milliseconds. The DNAmaca input file which specifies this model is given in Appendix A.

Two sets of runs were conducted. In the first, the effects of increasing the number of users in a system with a fixed memory capacity of $M = 1200$ was studied. In the second, the effect of varying the amount of available memory in a system with a workload of $N = 30$ users was studied.

Fig. 24 presents the results of the experiments. The graph on the left of the figure shows that a system with a memory capacity of $M = 1200$ can support up to about 32 users before the system begins to thrash, while the graph on the right shows that a memory capacity of about $M = 1100$ should suffice for system which has to support a maximum of 30 users.

7.4 TFTP telecommunications protocol

Fig. 25 presents a Generalised Stochastic Petri net (GSPN) model of the internet Trivial File Transfer Protocol (TFTP). TFTP is a simple protocol for the transfer of files which was designed to be implemented on top of the User Datagram Protocol (UDP).

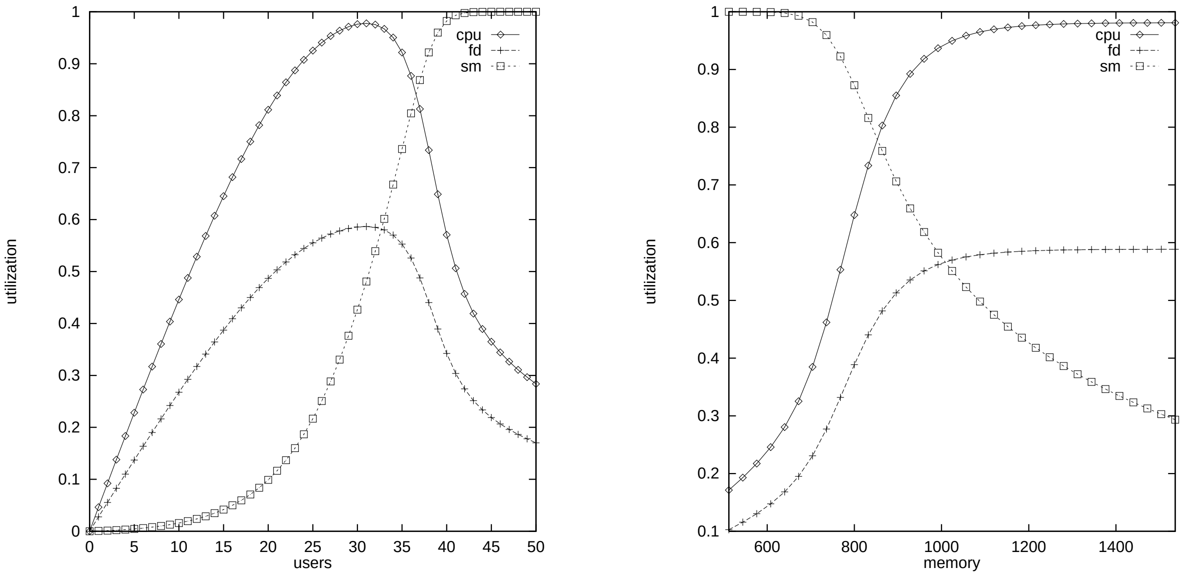


Figure 24: CPU, filing device and secondary memory utilization for various number of users on a system with $M = 1200$ (left) and utilization for various main memory capacities system with $N = 30$ users (right)

TFTP reads and writes files from or to a remote server. Each data packet is acknowledged separately – thereby ensuring that all the previous data packets have arrived at their destination before the next one is sent. If, for instance, the server is waiting for a data packet and the packet is lost on the channel, the server may timeout and retransmit the previous acknowledgement. The client, on receiving the acknowledgement, will retransmit the lost data packet. This process continues until the server receives the data packet.

The GSPN of Fig. 25 models a write-request from a sender to a receiver over an unreliable channel with a limited capacity. At the top level, the GSPN can be considered to have three columns of net elements representing the sender, the channel and the receiver from left to right, respectively. Tokens in the channel column represent packets in transit, while tokens in the column of places on the left and right represent the state of the sender and receiver.

The performance of the model was studied by examining the throughput of the protocol as a function of channel quality and various timeout values. Throughput is defined here as the number of useful data packets delivered per unit time. Average throughput was measured by determining the length of time required to send a 100 packet file over a network which transmits from the send to the receiver at an mean rate of 5 packets per second. Note that

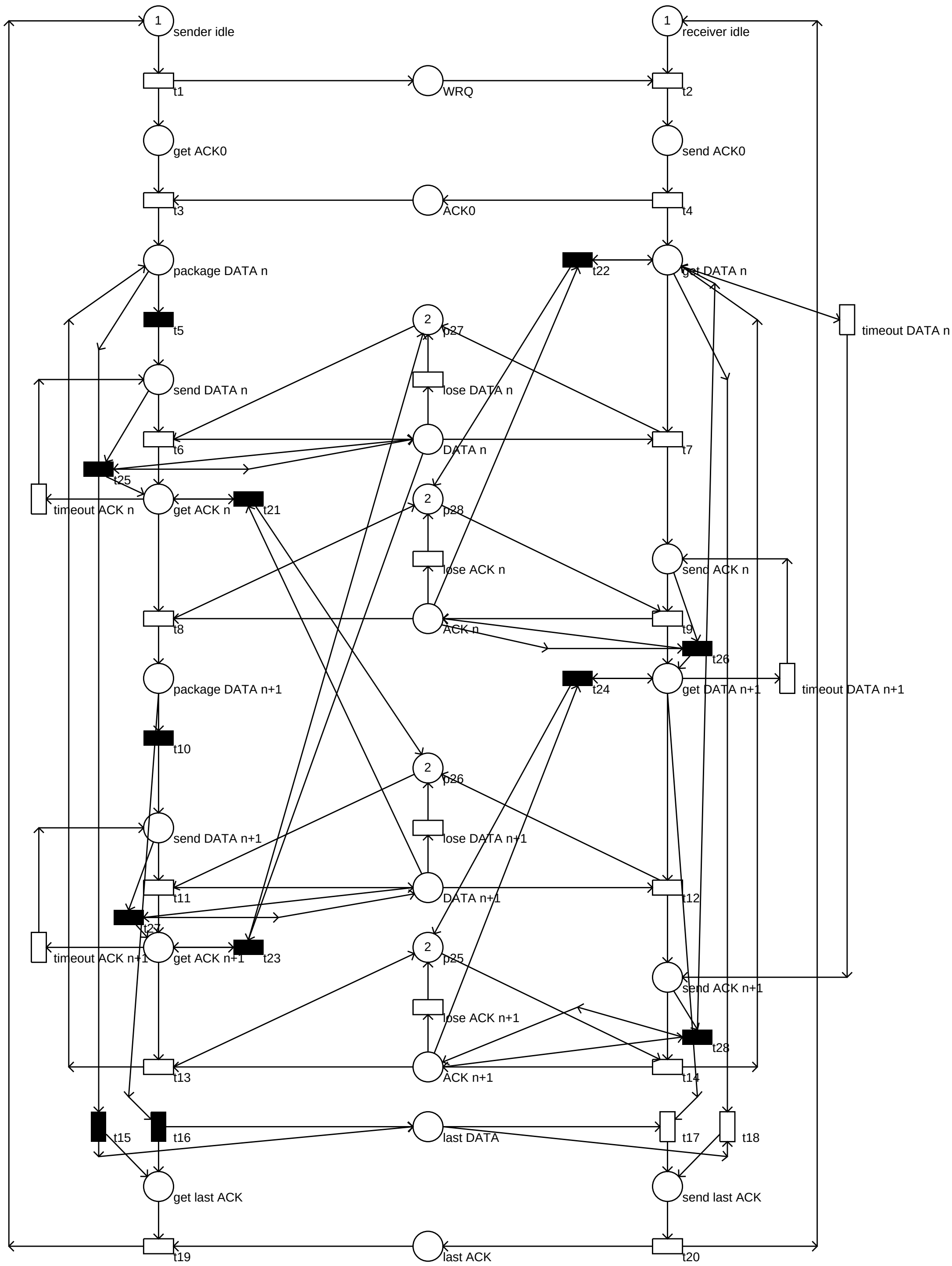


Figure 25: GSPN model of the TFTP file transfer protocol

the maximum mean throughput of the TFTP protocol is 2.5 packets per second since half the number of packets are acknowledgements. In order to simulate the transmission of a file of length 100 packets (on average) in the GSPN model the weight of the immediate transitions t5 and t10 (send another packet, see Fig. 25) are set to 99, while the weight of t15 and t16 (send last packet) are set to 1.

Channel quality is controlled by setting the mean rates on the *lose ACK* and *lose DATA* timed transitions in the channel. The length of timeouts are controlled, in turn, by setting the mean rates on the *timeout ACK* and *timeout DATA* timed transitions inside the sender and receiver respectively.

A DNAmaca model of the TFTP protocol was automatically generated by the DNAnet Petri net tool [ABK95]; this model is given in Appendix A. The model generates a Markov Chain with 200 tangible states.

Throughputs were calculated both analytically using DNAmaca and also by simulation using DNAnet. The results for various channel qualities and timeouts are presented in Fig. 26. The simulation results are plotted as points with 95% confidence intervals.

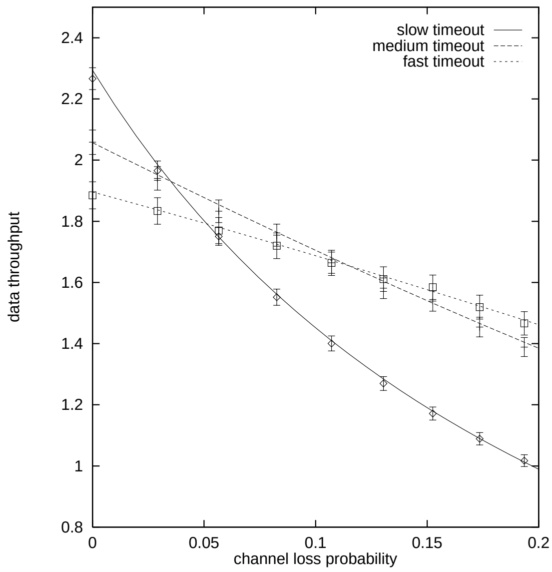


Figure 26: Throughput as a function of channel quality for various timeout values

The slow (long duration) timeout produces the best throughput for high channel qualities but has the worst throughput for poor channel qualities. This is expected since, for a highly reliable channel, there are few unnecessary timeouts, while for a poor quality channel, slow timeouts result in a long recovery time. The fast (short duration) timeout produces poor throughput for good channel qualities but better throughput for poor channel qualities. This is also expected, since the many retranmissions caused by a fast timeout are unnecessary on a good channel but reduce recovery time on a poor channel. The medium duration timeout gives intermediate results.

Chapter 8

Conclusion

8.1 Summary

This dissertation has investigated efficient techniques for assessing the performance of general timed transition systems, using Markov chains as the underlying vehicle for obtaining performance results. The process of establishing the performance of a system forms a sequence which begins with a high-level model specified using a formalism such as Generalised Stochastic Petri nets, queueing networks or Queueing Petri nets. The behaviour of the system is then characterised by enumerating all possible states that the model may enter. The states, together with temporal information about transitions between states, are then mapped onto a low-level Markov chain representation. From the Markov Chain, a set of sparse linear equations, known as the steady-state equations, are derived. These are solved to yield the steady-state distribution of the chain. This distribution indicates the long-run probability of being in each of the system's states. The basic information provided by the steady-state distribution can be synthesized into more meaningful performance statistics, such as transition throughput or mean buffer occupancy, by combining the steady-state distribution with the enumerated states.

We have investigated efficient techniques for the two major challenges encountered by the Markovian performance analyst when attempting to solve models of complex real life systems:

- **Enumerating the large number of states**. Existing techniques for solving this problem have been reviewed and a new probabilistic dynamic hash-compaction technique has been proposed. This method results in considerable memory savings over conventional static or exhaustive state space generation techniques. The reliability and space complexity of the technique has been analysed. The analysis shows that, given 64 Mb of available memory, it is possible to generate state spaces with up to 1.27×10^7 states while keeping the probability of omitting even one state to less than 0.1%.

- **Solving the large sparse set of steady-state equations**. A wide range of linear equation solvers has been reviewed, including two classes of historical methods, namely direct methods and classical iterative methods, and two classes of more recent iterative methods, namely Krylov subspace techniques and decomposition-based techniques. Particular attention has been paid to the Krylov subspace methods and a decomposition-based technique known as the Aggregation-Isolation algorithm.

The complete performance analysis sequence has been automated and implemented in the DNAmaca performance analyser. DNAmaca incorporates the new probabilistic dynamic state generation technique and uses on-the-fly elimination of vanishing states. In addition, DNAmaca implements twelve linear equation solvers, including six Krylov subspace-based solvers and two variants of Touzene's decomposition-based Aggregation-Isolation algorithm. Experiments with DNAmaca have shown that it is possible to solve models with up to 500 000 tangible states on a 64 Mb machine.

Finally, an interface language which is general enough to support the specification of general timed transition systems has been proposed and implemented in DNAmaca. Examples of the specification and solution of various timed transition systems have been presented.

8.2 Future work

With the development of memory-efficient probabilistic state space exploration algorithms, the process of solving the steady-state equations has become the major time and memory bottleneck in the performance analysis of timed transition systems. Parallel implementations of Krylov subspace techniques may provide the answer, since these techniques involve easily parallelisable operations such as matrix-vector multiplication. Indeed, experiments with Krylov subspace techniques for solving sets of linear equations with symmetric coefficient matrices have yielded superlinear speedups in both symmetric multiprocessing and high-speed distributed environments [Bou95]. There is no reason to suspect that similar results cannot be obtained for the methods applicable to solving the large unsymmetric infinitesimal generator matrices encountered in Markov chain analysis.

The time complexity of probabilistic state space exploration algorithms is another area that could benefit from parallelisation efforts. Work has already been done on the parallel generation of state spaces for Generalised Stochastic Petri nets [CCM95] which shows that less dramatic speedups can be expected here because of the high communication overhead inherent in state space generation algorithms.

Work on techniques for the parallelisation of DNAmaca has in fact begun and a version for use in distributed computing environments is expected by the middle of 1997.

Appendix A

DNAmaca model files

A.1 Multimedia teletraffic switch model

```
\model{
  \constant{ss}{72}             % servers in switch
  \constant{voice_source}{1000} % voice sources
  \constant{buffer_size}{200}   % data call buffer size
  \constant{threshold}{50}      % preemption threshold
  \constant{lambda_0}{0.04}     % voice silence -> talk spurt rate
  \constant{lambda_1}{1.0}      % voice talk spurt -> silence rate
  \constant{lambda_2}{43.0}     % data call arrival rate
  \constant{mu_2}{1.2}          % data service rate

  \statevector{
    \type{int}{data,voice,buffer}
  }

  \helpvalue{int}{idle_voice_source}{voice_source - voice}

  \invariant{ (voice + data) <= ss }

  \initial{
    data = 0;
    voice = 0;
    buffer = 0;
  }

  \transition{data_arrival}{
    \condition{buffer < buffer_size}
    \action{ next->buffer = buffer + 1; }
    \rate{lambda_2}
```

```
  }

  \transition{serve}{
    \condition{buffer > 0 && voice + data < ss}
    \action{
      next->buffer = buffer - 1;
      next->data = data + 1;
    }
    \weight{1.0}
  }

  \transition{data_service}{
    \condition{data > 0}
    \action{ next->data = data - 1; }
    \rate{(double)mu_2*data}
  }

  \transition{voice_arrival}{
    \condition{voice < ss && idle_voice_source}
    \action{
      if ( ((voice + data) >= ss) && (data > threshold)) {
        if (buffer < buffer_size)
          next->buffer = buffer + 1;
        next->data = data - 1;
        next->voice = voice + 1;
      } else if ( ((voice + data) >= ss) && (data <= threshold) ) {
        /* cannot preempt --> discard call */
      } else if ((voice + data) < ss) {
        next->voice = voice + 1;
      }
    }
    \rate{ (double) lambda_0*idle_voice_source}
  }

  \transition{voice_service}{
    \condition{voice > 0}
    \action{ next->voice = voice - 1; }
    \rate{ (double) lambda_1*voice}
  }

}

\performance{
  \statemeasure{mean voice} {
    \estimator{mean variance distribution}
    \expression{voice}
  }
```

```
  \statemeasure{mean data} {
    \estimator{mean variance distribution}
    \expression{data}
  }

  \statemeasure{mean buffer} {
    \estimator{mean variance distribution}
    \expression{buffer}
  }

  \countmeasure{blocking rate} {
    \estimator{mean}
    \precondition{1}
    \postcondition{voice == next->voice}
    \transition{voice_arrival}
  }

  \countmeasure{voice throughput}{
    \estimator{mean}
    \transition{voice_service}
  }

  \countmeasure{data throughput}{
    \estimator{mean}
    \transition{data_service}
  }
}

\solution{
  \method{bicgstab2}
  \accuracy{1e-10}
}
```

A.2 Interactive computer system model

```
\model{

  \header{#include<math.h>}

  % N       = total number of users
  % lambda  = mean job arrival rate from each idle terminal
  % compute = mean compute time for a process
  % M       = memory size
  % io      = mean compute time between i/o requests
  % page    = page fault rate dependent on number of processes in system
  % alpha   = page fault-related constant dependent on processing speed
  % k       = page fault-related constant dependent on program locality
```

```
% smspeed = mean service time for sm device
% fdspeed = mean service time for fd device

\constant{N}{50}
\constant{lambda}{0.0001}
\constant{compute}{500.0}
\constant{M}{1024.0}
\constant{io}{20.0}
\constant{alpha}{0.01}
\constant{k}{1.5}
\constant{page}{(alpha*pow( M/(cpu+sm+fd), k ))}
\constant{smspeed}{5.0}
\constant{fdspeed}{12.0}

% idle    = number of idle terminals
% cpu     = number of processes on cpu
% sm      = number of processes waiting on secondary memory
% fd      = number of processes waiting on filing device
\statevector{
  \type{int}{idle, cpu, sm, fd}
}

\initial{
  idle = N;
  cpu = 0;
  sm = 0;
  fd = 0;
}

\invariant{idle + cpu + sm + fd == N}

\transition{job_start}{
  \condition{idle > 0}
  \action{
    next->idle = idle - 1;
    next->cpu = cpu + 1;
  }
  \rate{ (double) idle*lambda}
}

\transition{job_finish}{
  \condition{cpu > 0}
  \action{
    next->cpu = cpu - 1;
    next->idle = idle + 1;
  }
  \rate{ (double) 1.0/compute}
}
```

```
  \transition{job_enters_fd}{
    \condition{cpu > 0}
    \action{
      next->cpu = cpu - 1;
      next->fd = fd + 1;
    }
    \rate{ (double) 1.0/io}
  }

  \transition{job_enters_sm}{
    \condition{cpu > 0}
    \action{
      next->cpu = cpu - 1;
      next->sm = sm + 1;
    }
    \rate{ (double) 1.0/page}
  }

  \transition{job_leaves_sm}{
    \condition{sm > 0}
    \action{
      next->sm = sm - 1;
      next->cpu = cpu + 1;
    }
    \rate{ (double) 1.0/smspeed}
  }

  \transition{job_leaves_fd}{
    \condition{fd > 0}
    \action{
      next->fd = fd - 1;
      next->cpu = cpu + 1;
    }
    \rate{ (double) 1.0/fdspeed}
  }
}

\solution{
  \method{air}
  \maxiterations{2000}
}

\performance{
  \statemeasure{idle terminals}{
    \estimator{mean variance stddev distribution}
    \expression{idle}
  }
```

```
  \statemeasure{processes on cpu}{
    \estimator{mean variance stddev distribution}
    \expression{cpu}
  }
  \statemeasure{cpu utilization}{
    \estimator{mean variance stddev}
    \expression{ (cpu > 0) ? 1 : 0}
  }
  \statemeasure{processes waiting for filing device}{
    \estimator{mean variance stddev distribution}
    \expression{fd}
  }
  \statemeasure{filing device utilization}{
    \estimator{mean variance stddev}
    \expression{ (fd > 0) ? 1 : 0}
  }
  \statemeasure{processes waiting for secondary memory}{
    \estimator{mean variance stddev distribution}
    \expression{sm}
  }
  \statemeasure{secondary memory utilization}{
    \estimator{mean variance stddev}
    \expression{ (sm > 0) ? 1 : 0}
  }

  \countmeasure{jobs entering}{
    \estimator{mean}
    \transition{job_start}
  }
  \countmeasure{jobs leaving}{
    \estimator{mean}
    \transition{job_finish}
  }
}
```

A.3 TFTP communications protocol

This DNAmaca model file was automatically generated by the GSPN tool DNAnet from the TFTP model presented in Fig. 25. The mapping from the places and transitions reflected in the diagram to the places and transitions used in this model file is given at the top of the file.

```
% p0 = 'main.sender idle'        p1 = 'main.receiver idle'
% p2 = 'main.get ACK0'           p3 = 'main.send ACK0'
```

```
% p4 = 'main.WRQ'                p5 = 'main.ACK0'
% p6 = 'main.package DATA n'     p7 = 'main.wait DATA n'
% p8 = 'main.get ACK n'          p9 = 'main.send ACK n'
% p10 = 'main.last DATA'         p11 = 'main.send last ACK'
% p12 = 'main.get last ACK'      p13 = 'main.last ACK'
% p14 = 'main.send DATA n'       p15 = 'main.package DATA n+1'
% p16 = 'main.send DATA n+1'     p17 = 'main.get ACK n+1'
% p18 = 'main.get DATA n+1'      p19 = 'main.send ACK n+1'
% p20 = 'main.DATA n'            p21 = 'main.ACK n'
% p22 = 'main.DATA n+1'          p23 = 'main.ACK n+1'
% p24 = 'main.p25'               p25 = 'main.p26'
% p26 = 'main.p27'               p27 = 'main.p28'
% t0 = 'main.t15'                t1 = 'main.t16'
% t2 = 'main.t10'                t3 = 'main.t5'
% t4 = 'main.t24'                t5 = 'main.t22'
% t6 = 'main.t23'                t7 = 'main.t21'
% t8 = 'main.t25'                t9 = 'main.t26'
% t10 = 'main.t27'               t11 = 'main.t28'
% t12 = 'main.lose ACK n+1'      t13 = 'main.lose DATA n+1'
% t14 = 'main.lose ACK n'        t15 = 'main.t18'
% t16 = 'main.t17'               t17 = 'main.t3'
% t18 = 'main.t8'                t19 = 'main.t14'
% t20 = 'main.t13'               t21 = 'main.t12'
% t22 = 'main.t1'                t23 = 'main.t2'
% t24 = 'main.t7'                t25 = 'main.t4'
% t26 = 'main.t9'                t27 = 'main.t19'
% t28 = 'main.t20'               t29 = 'main.timeout ACK n'
% t30 = 'main.lose DATA n'       t31 = 'main.timeout ACK n+1'
% t32 = 'main.t11'               t33 = 'main.t6'
% t34 = 'main.timeout DATA n+1' t35 = 'main.timeout DATA n'


\model {
  \statevector{
    \type{short}{p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11,
      p12, p13, p14, p15, p16, p17, p18, p19, p20, p21, p22, p23, p24,
      p25, p26, p27}
  }

  \initial{
    p0 = 1;    p1 = 1;    p2 = 0;    p3 = 0;    p4 = 0;    p5 = 0;
    p6 = 0;    p7 = 0;    p8 = 0;    p9 = 0;    p10 = 0;  p11 = 0;
    p12 = 0;   p13 = 0;   p14 = 0;   p15 = 0;   p16 = 0;  p17 = 0;
    p18 = 0;   p19 = 0;   p20 = 0;   p21 = 0;   p22 = 0;  p23 = 0;
    p24 = 2;   p25 = 2;   p26 = 2;   p27 = 2;
  }

  \transition{t0}{
```

```
    \condition{p6 > 0}
    \action{
      next->p6 = p6 - 1;
      next->p10 = p10 + 1;
      next->p12 = p12 + 1;
    }
    \weight{1}
  }
  \transition{t1}{
    \condition{p15 > 0}
    \action{
      next->p10 = p10 + 1;
      next->p12 = p12 + 1;
      next->p15 = p15 - 1;
    }
    \weight{1}
  }
  \transition{t2}{
    \condition{p15 > 0}
    \action{
      next->p15 = p15 - 1;
      next->p16 = p16 + 1;
    }
    \weight{99}
  }
  \transition{t3}{
    \condition{p6 > 0}
    \action{
      next->p6 = p6 - 1;
      next->p14 = p14 + 1;
    }
    \weight{99}
  }
  \transition{t4}{
    \condition{p18 > 0 && p20 > 0}
    \action{
      next->p20 = p20 - 1;
      next->p26 = p26 + 1;
    }
    \weight{1}
  }
  \transition{t5}{
    \condition{p7 > 0 && p22 > 0}
    \action{
      next->p22 = p22 - 1;
      next->p25 = p25 + 1;
    }
    \weight{1}
```

```
}
\transition{t6}{
  \condition{p16 > 0 && p23 > 0}
  \action{
    next->p23 = p23 - 1;
    next->p24 = p24 + 1;
  }
  \weight{1}
}
\transition{t7}{
  \condition{p14 > 0 && p21 > 0}
  \action{
    next->p21 = p21 - 1;
    next->p27 = p27 + 1;
  }
  \weight{1}
}
\transition{t8}{
  \condition{p14 > 0 && p20 > 1}
  \action{
    next->p8 = p8 + 1;
    next->p14 = p14 - 1;
  }
  \weight{1}
}
\transition{t9}{
  \condition{p9 > 0 && p21 > 1}
  \action{
    next->p9 = p9 - 1;
    next->p18 = p18 + 1;
  }
  \weight{1}
}
\transition{t10}{
  \condition{p16 > 0 && p22 > 1}
  \action{
    next->p16 = p16 - 1;
    next->p17 = p17 + 1;
  }
  \weight{1}
}
\transition{t11}{
  \condition{p19 > 0 && p23 > 1}
  \action{
    next->p7 = p7 + 1;
    next->p19 = p19 - 1;
  }
  \weight{1}
```

```
}
\transition{t12}{
  \condition{p23 > 0}
  \action{
    next->p23 = p23 - 1;
    next->p24 = p24 + 1;
  }
  \rate{2.5}
}
\transition{t13}{
  \condition{p22 > 0}
  \action{
    next->p22 = p22 - 1;
    next->p25 = p25 + 1;
  }
  \rate{2.5}
}
\transition{t14}{
  \condition{p21 > 0}
  \action{
    next->p21 = p21 - 1;
    next->p27 = p27 + 1;
  }
  \rate{2.5}
}
\transition{t15}{
  \condition{p7 > 0 && p10 > 0}
  \action{
    next->p7 = p7 - 1;
    next->p10 = p10 - 1;
    next->p11 = p11 + 1;
  }
  \rate{10}
}
\transition{t16}{
  \condition{p10 > 0 && p18 > 0}
  \action{
    next->p10 = p10 - 1;
    next->p11 = p11 + 1;
    next->p18 = p18 - 1;
  }
  \rate{10}
}
\transition{t17}{
  \condition{p2 > 0 && p5 > 0}
  \action{
    next->p2 = p2 - 1;
    next->p5 = p5 - 1;
```

```
    next->p6 = p6 + 1;
  }
  \rate{10}
}
\transition{t18}{
  \condition{p8 > 0 && p21 > 0}
  \action{
    next->p8 = p8 - 1;
    next->p15 = p15 + 1;
    next->p21 = p21 - 1;
    next->p27 = p27 + 1;
  }
  \rate{10}
}
\transition{t19}{
  \condition{p19 > 0 && p24 > 0}
  \action{
    next->p7 = p7 + 1;
    next->p19 = p19 - 1;
    next->p23 = p23 + 1;
    next->p24 = p24 - 1;
  }
  \rate{10}
}
\transition{t20}{
  \condition{p17 > 0 && p23 > 0}
  \action{
    next->p6 = p6 + 1;
    next->p17 = p17 - 1;
    next->p23 = p23 - 1;
    next->p24 = p24 + 1;
  }
  \rate{10}
}
\transition{t21}{
  \condition{p18 > 0 && p22 > 0}
  \action{
    next->p18 = p18 - 1;
    next->p19 = p19 + 1;
    next->p22 = p22 - 1;
    next->p25 = p25 + 1;
  }
  \rate{10}
}
\transition{t22}{
  \condition{p0 > 0}
  \action{
    next->p0 = p0 - 1;
```

```
    next->p2 = p2 + 1;
    next->p4 = p4 + 1;
  }
  \rate{10}
}
\transition{t23}{
  \condition{p1 > 0 && p4 > 0}
  \action{
    next->p1 = p1 - 1;
    next->p3 = p3 + 1;
    next->p4 = p4 - 1;
  }
  \rate{10}
}
\transition{t24}{
  \condition{p7 > 0 && p20 > 0}
  \action{
    next->p7 = p7 - 1;
    next->p9 = p9 + 1;
    next->p20 = p20 - 1;
    next->p26 = p26 + 1;
  }
  \rate{10}
}
\transition{t25}{
  \condition{p3 > 0}
  \action{
    next->p3 = p3 - 1;
    next->p5 = p5 + 1;
    next->p7 = p7 + 1;
  }
  \rate{10}
}
\transition{t26}{
  \condition{p9 > 0 && p27 > 0}
  \action{
    next->p9 = p9 - 1;
    next->p18 = p18 + 1;
    next->p21 = p21 + 1;
    next->p27 = p27 - 1;
  }
  \rate{10}
}
\transition{t27}{
  \condition{p12 > 0 && p13 > 0}
  \action{
    next->p0 = p0 + 1;
    next->p12 = p12 - 1;
```

```
    next->p13 = p13 - 1;
  }
  \rate{10}
}
\transition{t28}{
  \condition{p11 > 0}
  \action{
    next->p1 = p1 + 1;
    next->p11 = p11 - 1;
    next->p13 = p13 + 1;
  }
  \rate{10}
}
\transition{t29}{
  \condition{p8 > 0}
  \action{
    next->p8 = p8 - 1;
    next->p14 = p14 + 1;
  }
  \rate{4}
}
\transition{t30}{
  \condition{p20 > 0}
  \action{
    next->p20 = p20 - 1;
    next->p26 = p26 + 1;
  }
  \rate{2.5}
}
\transition{t31}{
  \condition{p17 > 0}
  \action{
    next->p16 = p16 + 1;
    next->p17 = p17 - 1;
  }
  \rate{4}
}
\transition{t32}{
  \condition{p16 > 0 && p25 > 0}
  \action{
    next->p16 = p16 - 1;
    next->p17 = p17 + 1;
    next->p22 = p22 + 1;
    next->p25 = p25 - 1;
  }
  \rate{10}
}
\transition{t33}{
```

```
    \condition{p14 > 0 && p26 > 0}
    \action{
      next->p8 = p8 + 1;
      next->p14 = p14 - 1;
      next->p20 = p20 + 1;
      next->p26 = p26 - 1;
    }
    \rate{10}
  }
  \transition{t34}{
    \condition{p18 > 0}
    \action{
      next->p9 = p9 + 1;
      next->p18 = p18 - 1;
    }
    \rate{4}
  }
  \transition{t35}{
    \condition{p7 > 0}
    \action{
      next->p7 = p7 - 1;
      next->p19 = p19 + 1;
    }
    \rate{4}
  }
}

\performance{
  \statemeasure{Mean tokens on place main.sender idle}{
    \estimator{mean variance distribution}
    \expression {p0}
  }
  \statemeasure{Mean tokens on place main.receiver idle}{
    \estimator{mean variance distribution}
    \expression {p1}
  }
  \statemeasure{Mean tokens on place main.get ACK0}{
    \estimator{mean variance distribution}
    \expression {p2}
  }

  (etc...)

  \statemeasure{Mean tokens on place main.p28}{
    \estimator{mean variance distribution}
    \expression {p27}
  }
```

```
  \statemeasure{Enabled probability for transition main.lose ACK n+1}{
    \estimator{mean}
    \expression {(p23 > 0) ? 1 : 0}
  }
  \countmeasure{Throughput for transition main.lose ACK n+1}{
    \estimator{mean}
    \precondition{1}
    \postcondition{1}
    \transition{t12}
  }
  \statemeasure{Enabled probability for transition main.lose DATA n+1}{
    \estimator{mean}
    \expression {(p22 > 0) ? 1 : 0}
  }
  \countmeasure{Throughput for transition main.lose DATA n+1}{
    \estimator{mean}
    \precondition{1}
    \postcondition{1}
    \transition{t13}
  }

  (etc...)

  \statemeasure{Enabled probability for transition main.timeout DATA n}{
    \estimator{mean}
    \expression {(p7 > 0) ? 1 : 0}
  }
  \countmeasure{Throughput for transition main.timeout DATA n}{
    \estimator{mean}
    \precondition{1}
    \postcondition{1}
    \transition{t35}
  }
}
```

Bibliography

[ABK95] Annitta Attieh, Mark Brady, and William Knottenbelt. Functional and temporal analysis of concurrent systems. In Jonathan Billington and Michel Diaz, editors, *Petri nets applied to Protocols: Proceedings of a Workshop of the 16th International Conference on Application and Theory of Petri nets*, pages 79–96, Torino, Italy, June 1995.

[AK93] Haruo Akimaru and Konosuke Kawashima. *Teletraffic: theory and applications*. Springer-Verlag, 1993.

[AMCB84] M. Ajmone-Marsan, G. Conte, and G. Balbo. A class of Generalised Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2:93–122, 1984.

[AMS90] Steven F. Ashby, Thomas A. Manteuffel, and Paul E. Saylor. A taxonomy for conjugate gradient methods. *SIAM Journal on Numerical Analysis*, 27(6):1542–1568, December 1990.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[B+94] J.R. Burch et al. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):250 – 263, 1994.

[B+95] Werner Backes et al. *LiDIA: A library for computational number theory*. Universität des Saarlandes, 1995.

[Baa88] Sara Baase. *Computer Algorithms*. Addison-Wesley, 1988.

[Bar89] V.A. Barker. Numerical solution of sparse singular systems of equations arising from ergodic Markov Chains. *Communications in Statistics: Stochastic Models*, 5(3):335–381, 1989.

[Bau93] F. Bause. Queueing Petri nets: A formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of the 5th International Workshop on Petri nets and Performance Models.* IEEE, October 1993.

[BB89] F. Bause and H. Beilner. Eine Modellwelt zur Integration von Warteschlangen- und Petri-Netz-Modellen. In *Proceedings of the 5th GI/ITG-Fachtagung, Messung, Modellierung und Bewertung von Rechensystemen und Netzen*, pages 190–204. Braunschweig, Gesellschaft für Informatik (GI), Germany, September 1989.

[BBC+94] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the solution of linear systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1994.

[BCMP75] F. Basket, K.M. Chandy, R.R. Muntz, and F.G. Palacios. Open, closed and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22:248 – 260, 1975.

[BDMC+94] P. Buchholz, J. Dunkel, B. Müller-Clostermann, M. Sczittnick, and S. Zäske. *Quantitative Systemanalyse mit Markovschen Ketten.* Teubner, 1994.

[BK94] F. Bause and P. Kemper. QPN-Tool for the qualitative and quantitative analysis of Queueing Petri Nets. In G. Haring and G. Kotsis, editors, *Lecture notes in Computer Science 794: Proceedings of the 7th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools, Vienna, Austria*, pages 321–334. Springer-Verlag, 1994.

[BK95] F. Bause and P.S. Kritzinger. *Stochastic Petri net theory.* Verlag Vieweg, Wiesbaden, Germany, 1995.

[BKKK95] F. Bause, H. Kabutz, P. Kemper, and P. Kritzinger. SDL and Petri net performance of communicating systems. In *Proceedings of IFIP/PSTV95: Conference on Protocol Specification, Testing and Verification, Warsaw, Poland*, pages 269–282. Chapman and Hall, June 1995.

[Bou95] Brendan Boulter. Performance evaluation of HPF for scientific computing. In *Lecture Notes in Computer Science 919*. Springer-Verlag, 1995.

[BS87] H. Beilner and F.J. Stewing. Concepts and techniques of the performance modelling tool HIT. In *Proc. European Simulation Multiconference*, Vienna, 1987.

[CCM95] S. Caselli, G. Conte, and P. Marenzoni. Parallel state exploration for GSPN models. In *Lecture Notes in Computer Science 935: Proceedings of the 16th International Conference on the Application and Theory and Petri Nets, Turin, Italy*. Springer-Verlag, June 1995.

[Cou85] P.J. Courtois. On time and space decomposition of complex structures. *Communications of the ACM*, 28(6):590–603, June 1985.

[CS84] P.J. Courtois and P. Semal. Block decomposition and iteration in stochastic matrices. *Philips Journal of Research*, 39:178–194, 1984.

[CS85] Wei-Lu Cao and William J. Stewart. Iterative aggregation/disaggregation techniques for nearly uncoupled Markov chains. *Journal of the ACM*, 32(3):702–719, July 1985.

[CW79] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[DCB93] T. Demaria, G. Chiola, and G. Bruno. Introducing a color formalism into Generalised Stochastic Petri nets. In *Proceedings of the 9th International Workshop on Application and Theory of Petri Nets*. IEEE, October 1993.

[DER86] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, 1986.

[DLPR95] Jack Dongarra, Andrew Lumsdaine, Roldan Pozo, and Karin Remington. *IML++ v. 1.1: Iterative Methods Library Reference Guide.* National Institute of Standards and Technology, Oak Ridge National Laboratory, February 1995.

[EWK90] L. Eldén and L. Wittmeyer-Koch. *Numerical Analysis: An Introduction.* Academic Press, 1990.

[FGN92] Roland W. Freund, Gene H. Golub, and Noël M. Nachtigal. Iterative solution of linear systems. *Acta Numerica*, pages 1–44, 1992.

[Fle76] R. Fletcher. Conjugate gradient methods for indefinite systems. In *Lecture Notes in Mathematics*, volume 506, pages 73–89. Springer-Verlag, 1976.

[FM84] Vance Faber and Thomas Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM Journal on Numerical Analysis*, 21(2):352–362, April 1984.

[FN91] Roland W. Freund and Noël M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.

[FN94] Roland W. Freund and Noël M. Nachtigal. QMRPACK: a package of QMR algorithms. Technical Report 4-16, AT&T Bell Laboratories, 1994.

[Fre93] Roland W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM Journal on Scientific Computing*, 14(2):470–482, March 1993.

[GKS95] Anshul Gupta, Vipin Kumar, and Ahmed Sameh. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. *IEEE Transactions on Parllel and Distributed Systems*, 6(5):455–469, May 1995.

[GL89] Gene H. Golub and Charles F. van Loan. *Matrix Computations.* John Hopkins Press, Maryland, 2nd edition, 1989.

[Goo88] Roe Goodman. *Introduction to Stochastic Models.* Benjamin/Cummings, 1988.

[GTH85] W.K. Grassmann, M.I. Taskar, and D.P. Heyman. Regenerative analysis and steady state distributions for Markov chains. *Operations Research*, 33(5):1107–1116, 1985.

[Gut93a] Martin H. Gutknecht. Changing the norm in conjugate gradient type algorithms. *SIAM Journal on Numerical Analysis*, 30(1):40–56, February 1993.

[Gut93b] Martin H. Gutknecht. Variants of BICGSTAB for matrices with complex spectrum. *SIAM Journal on Scientific Computing*, 14(5):1020–1033, September 1993.

[GW89] C.F. Gerald and P.O. Wheatley. *Applied Numerical Analysis.* Addison-Wesley, 4th edition, 1989.

[Hog89] D. Hogrefe. *Estelle, LOTOS and SDL.* Springer-Verlag, 1989.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[Hol95] Gerard J. Holzmann. An analysis of bitstate hashing. In *Proceedings of IFIP/PSTV95: Conference on Protocol Specification, Testing and Verification.* Chapman & Hall, Warsaw, Poland, June 1995.

[HS52] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–435, 1952.

[HY81] L.A. Hageman and D.M. Young. *Applied Iterative Methods.* Academic Press, 1981.

[Kab95] Heinz M. Kabutz. QPN-model of reduced InRes protocol. QPN-Tool input file, 1995. Available by email from `heinz@cs.uct.ac.za`.

[Kel79] F.P. Kelley. *Reversibility and Stochastic Networks.* Wiley and Sons, 1979.

[Kem95] P. Kemper. Numerical analysis of superposed GSPNs. In *Proc. of the Sixth International Workshop on Petri Nets and Perfromance Models*, pages 52–62. IEEE Computer Society Press, 1995.

[KGB87] Sudhir Kumar, Winfried Grassmann, and Roy Billinton. A stable algorithm to calculate steady-state probability and frequency of a Markov system. *IEEE Transactions on Reliability*, 36(1):58–62, April 1987.

[Kle75] Leonard Kleinrock. *Queueing Systems*, volume 1. John Wiley and Sons, 1975.

[KMCS90] Udo R. Krieger, Bruno Müller-Clostermann, and Michael Sczittnick. Modelling and analysis of communication systems based on computational methods for Markov chains. *IEEE Journal on Selected Areas in Communications*, 8(9):1630–1648, December 1990.

[KMS84] J.R. Koury, D.F. McAllister, and W.J. Stewart. Iterative methods for computing stationary distributions of nearly completely decomposable Markov chains. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):164–186, June 1984.

[KS60] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. Van Nostrand, 1960.

[KS95] R.L. Klevans and W.J. Stewart. From queueing networks to Markov chains: the XMARCA interface. *Performance Evaluation*, 24:23–45, 1995.

[LZGS84] E.D. Lazowska, J. Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[MCS88] Bruno Müller-Clostermann and Michael Sczittnick. A framework for the implementation of generators for stochastic finite state models. Technical report, University of Dortmund, 1988.

[ME90] Ulrike Meier and Rudolf Eigenmann. Parallelization and performance of conjugate gradient algorithms on the Cedar hierarchical-memory multiprocessor. Technical Report 1035, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1990.

[Mei94] Ulrike Meier-Yang. Preconditioned conjugate gradient-like methods for nonsymmetric linear systems. Technical Report 1210, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, July 1994.

[Nac91] Noël M. Nachtigal. *A look-ahead variant of the Lanczos algorithm and its application to the quasi-minimal residual method for non-Hermitian linear systems.* PhD thesis, Massachusetts Institute of Technology, August 1991.

[Pet81] J.L. Peterson. *Petri Nets and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Pre92] William H. Press. *Numerical recipes in C: the art of scientific computing.* Cambridge University Press, Cambridge, 2nd edition, 1992.

[PRL95] Roldan Pozo, Karin Remington, and Andrew Lumsdaine. *SparseLib v. 1.3: Sparse Matrix Class Library Reference Guide.* National Institute of Standards and Technology, March 1995.

[Rei92] W. Reisig. *A Primer in Petri Net Design.* Springer-Verlag, 1992.

[Saa89] Youcef Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, November 1989.

[Sch86] Paul J. Schweitzer. An iterative aggregation-disaggregation algorithm for solving linear equations. *Applied Mathematics and Computation*, 18:313–353, 1986.

[Scz87] Michael Sczittnick. Technicken zur funktionalen und quantitativen Analyse von Markoffschen Rechensystemmodellen. Diplomarbeit, Universität Dortmund, October 1987.

[SD95] U. Stern and D.L. Dill. Improved probabilistic verification by hash compaction. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1995.

[SF93] Gerard L. Sleijpen and Diederik R. Fokkema. BiCGSTAB(L) for linear equations involving unsymmetric matrices with complex spectrum. *Electronic Transactions on Numerical Analysis*, 1:11–32, September 1993.

[SMC90] Michael Sczittnick and Bruno Müller-Clostermann. MACOM - a tool for the Markovian analysis of communication systems. In *Proceedings of the Fourth International Conference on Data Communication Systems and their Performance*, Barcelona, Spain, June 1990.

[Son89] Peter Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 10(1):36–52, January 1989.

[SS86] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, July 1986.

[Ste91] W.J. Stewart. MARCA: Markov chain analyser. A software package for Markov modelling. In W.J. Stewart, editor, *Numerical Solution of Markov Chains*, pages 37–62. Marcel Dekker Inc., New York, 1991.

[Ste94] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[SV86] A. van der Sluis and H.A. van der Vorst. The rate of convergence of conjugate gradients. *Numerische Mathematik*, 48:543–560, 1986.

[SV95] Gerard L. Sleijpen and Henk van der Vorst. An overview of approaches for the stable computation of hybrid bicg methods. Preprint 908, University of Utrecht, Department of Mathematics, March 1995. To appear in Appl. Numer. Math.

[SW95] W. Schönauer and R. Weiss. An engineering approach to generalized conjugate gradient methods and beyond. *Appl. Numer. Math.*, 19:175–206, 1995. Special Issue on Iterative Methods for Linear Equations.

[Taf95] D.K. Tafti. A study of Krylov methods for the solution of the pressure-Poisson equation on the CM-5. *Numerical Developments in CFD*, FED-Vol. 215, August 1995.

[Tou95] A. Touzene. A new iterative method for solving large-scale Markov chains. In *Lecture Notes in Computer Science 977: Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Heidelberg*. Springer-Verlag, September 1995.

[Ulr95] Roya Ulrich. *Reservoir-based resource management for slotted high-speed networks*. PhD thesis, Universität Erlangen Nürnberg, September 1995.

[Var62] R.S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, 1962.

[Vor92] Henk van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, March 1992.

[Vor93] Henk van der Vorst. Lecture notes on iterative methods. Lecture notes, University of Utrecht, December 1993.

[Wal88a] Homer F. Walker. Implementation of the GMRES method using householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 9(1):152–163, January 1988.

[Wal88b] Jean Walrand. *An Introduction to Queueing Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Wei94] R. Weiss. Orthogonalization methods. Interner Bericht 52, Universität Karlsruhe, 1994.

[Wei95] R. Weiss. A theoretical overview of Krylov subspace methods. *Appl. Numer. Math.*, 19:207–233, 1995. Special Issue on Iterative Methods for Linear Equations.

[Wil65] J.H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, 1965.

[WL93] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In *Lecture Notes in Computer Science 697*, pages 59–70. Springer-Verlag, 1993.