Faculty of Natural Sciences and Technology I

Department of Computer Science

Bachelorthesis

Android App for Minimization of Acyclic Phase-Type Distributions

Submitted by:
Michael Heiner Bungert

Reviewers:
Prof. Dr. Holger Hermanns
Dr. Reza Pulungan

Submitted:
December 19, 2013

Acyclic phase-type distributions are probability distributions that mainly consist of combinations of exponential distributions and can be represented by continuous-time Markov chains. The computation time when analyzing these distributions strongly depends on the size of their representations. However, because their representations aren't unique, an algorithm was developed by Reza Pulungan and Holger Hermanns to minimize their state space [5]. They also developed a calculus which is able to construct these distributions.

In this thesis, an Android application to build and display these distributions according to this calculus is developed. It uses an already implemented webservice to compute their minimal representations and makes this service reachable from any Android device.

Contents

1 Introduction

Model-based analysis on systems forms one of the basis aspects of research in computer science. We are able to transform systems into appropriate models that provide the possibility to explicitly analyse the important internal interactions inside the system and its interaction with the environment. *Markov models* are state-based models which are used often for this purpose because of their property not depending on previous visited states. A special area of Markov models are *acyclic phase-type distributions* (*APH*). These are Markov chains representing the distribution of the time until an absorbing state is reached. Different operators are commonly used to connect different distributions and to build more complex models.

The problem that comes along with these models and especially the operators is the *state-explosion problem*. Many of the the operators build cross products of the state spaces of the operands which leads to a fast growth of the resulting state space. This leads to long computation times and high memory consumption when running algorithms on these models. Minimized representations and algorithms to compute them are needed. For acyclic phase-type distributions, Reza Pulungan and Holger Hermanns developed an algorithm with polynomial complexity [4] to compute minimized representations and implemented this algorithm in the tool suite *APHMIN* [5]. The tool uses the *Cox & Convenience Calculus* to generate expressions representing acyclic phase-type distributions. The structure of the calculus provides the ability to connect simple exponential distributions with different operators.

The omnipresence of mobile devices today and the simple Drag and Drop mechanism they provide lead to the first aim of this thesis. We want to implement a mobile application which uses the *APHMIN* tool to build expressions and compute their minimized representation.

The operators that are used in the Cox & Convenience Calculus rely on several mathematical operators frequently used for all kinds of distributions. They are normally used as n-ary operators to connect any number of distributions. At the moment, *APHMIN* is only able to use them as binary operators. But the app should also be able to build expressions with n-ary operators to be consistent with the mathematical concept. This leads to the second aim of this thesis. We will develop a heuristic to transform n-ary operators into their corresponding binary versions. The transformed expressions should lead to lower computation times when applying the minimization algorithm.

The thesis is divided into six chapters. The chapters are organized as follows:

- Chapter 2 provides the mathematical basics that are needed throughout the thesis. We define acyclic phase-type distributions and the operators that are used to connect them. We will also present the Cox & Convenience Calculus.

- In chapter 3, the minimization algorithm and the implementation of the *APHMIN* tool is presented.

- Chapter 4 is about the implementation of the actual application. We discuss the basic design ideas for the app and the communication between the device and the server.

- In chapter 5, the heuristic for transforming n-ary operators into their binary versions is developed. We also take a closer look at the experimental results about different criteria that lead to these results.

- In chapter 6, we summarize the main parts of the thesis and evaluate the developed heuristic on different expressions.

2 Basic Knowledge

2.1 Mathematical Basics

First we take a closer look at some basic mathematical definitions. This section reviews phase-type distributions, continuous-time Markov chains and two canonical forms to represent them [3][5].

2.1.1 Continuous-Time Markov Chains

Continuous-time Markov chains (*CTMC*) can be used to approximate any continuous probability distribution. A *CTMC* is defined as a triple $M = (S, Q, \pi)$ with the following components:

- $S = \{s_1, s_2, \ldots, s_n, s_{n+1}\}$ is a finite set of states.
- $Q : (S \times S) \to \mathbb{R}$ is an infinitesimal generator matrix which consists of the transition rates between the states. For any two distinct states $s, s' \in S$, $Q(s, s')$ describes the transition rate between s and s'.
- $\pi : S \to [0, 1]$ describes the initial distribution.

The probability, that a state change from s to s' occurs within time t is *exponentially distributed* over t with rate $Q(s, s')$, so the probability can be computed by $1 - exp(-Q(s, s')t)$. The following two properties hold by definition:

- $Q(s, s') \geq 0$ for all $s \neq s'$
- $Q(s, s) = -\sum_{s \neq s'} Q(s, s')$

The *exit rate* $E(s) = -Q(s, s)$ is the rate for the exponential distribution that describes the probability of leaving state s.

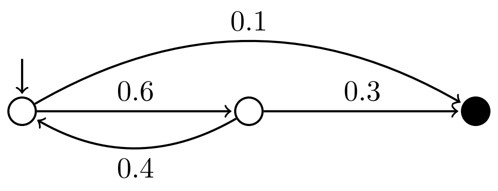


Figure 2.1: Cyclic *PH*-distribution

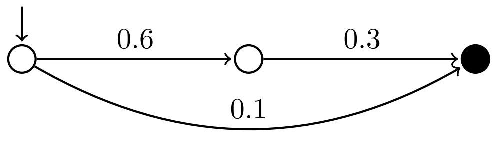


Figure 2.2: Acyclic *PH*-distribution

2.1.2 Phase-Type Distributions

If $E(s_i) = 0$ for a state s_i, this state is called an *absorbing state*. A state s_i is called *transient* if the probability that a state is not visited again once it is left is larger then 0. If there is an absorbing state s_{n+1} in a *CTMC M* and all other states s_i are transient, the distribution of the time to absorption is called a *phase-type* (*PH*) distribution. The generator matrix can be written as follows:

$$Q = \begin{bmatrix} A & \vec{A} \\ \vec{0} & 0 \end{bmatrix}$$

The matrix A is called a *PH-generator*. The vector $\vec{A}$ consists of the transition rates of every state s_i to the absorbing state s_{n+1} for every $1 \leq i \leq n$. The initial distribution $\pi = [\vec{\alpha}, \alpha_{n+1}]$ consists of the vector $\vec{\alpha}$, representing the probabilities of starting in a non-absorbing state, and the probability α_{n+1} of starting in the absorbing state. In the following, we only consider initial distributions with $\alpha_{n+1} = 0$.
Because $\sum_{s \in S} Q(s, s') = 0$ for every $s' \in S$, a *CTMC M* with $n + 1$ states (including the absorbing state) can be fully represented by the $n \times n$ PH-generator A and the n-dimensional row vector $\vec{\alpha}$. The notation $PH(\vec{\alpha}, \mathrm{A})$ is used do describe the *PH*-distribution.

During this thesis, we will normally deal with *acyclic PH*-distributions (*APH*). This subset of *PH*-distributions doesn't contain any cycles in their graph-representations. The graph in Figure 2.1 shows a cyclic *PH*-distribution while Figure 2.2 is an acyclic one.

A *minimal representation* of an *APH*-distribution is the representation of the same distribution with the fewest possible number of states. A minimal representation of an *APH*-distribution isn't necessarily acyclic. We call an *APH*-distribution *acyclic-ideal*, if there is a minimal representation of this distribution that is also acyclic.

2.1.3 Canonical Representations

There are two *canonical representations* for an *APH*-distribution. These two representations are unique according to isomorphism of their graph representations. The canonical representations are used later for the minimization algorithm.

Ordered Bidiagonal Representation

A *PH*-generator is called a *bidiagonal generator* if it has the following form:

$$\begin{bmatrix} -\lambda_1 & \lambda_1 & 0 & \dots & 0 \\ 0 & -\lambda_2 & \lambda_2 & \dots & 0 \\ 0 & 0 & -\lambda_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & -\lambda_n \end{bmatrix}$$

The generator can be denoted by $Bi(\lambda_1, \lambda_2, \dots, \lambda_n)$. In this representation, the exit rates of the generator (i.e. the negative diagonal values) are in ascending order. A representation of a *PH*-distribution using a bidiagonal generator as its *PH*-generator is called an *ordered bidiagonal representation.* Every *APH*-distribution can be transformed into an ordered bidiagonal representation according to the following theorem:

Theorem 1. *Let $(\vec{\alpha}, A)$ be an acyclic phase-type representation, $\lambda_1, \lambda_2, \dots, \lambda_n$ be the eigenvalues of $-A$, and, without loss of generality, assume that $\lambda_n \geq \lambda_{n-1} \geq \dots \geq \lambda_1$. Then there exists a unique bidiagonal representation, the ordered bidiagonal representation, $(\vec{\beta}, Bi(\lambda_1, \lambda_2, \dots, \lambda_n)$ such that:*

$$PH(\vec{\beta}, Bi(\lambda_1, \lambda_2, \dots, \lambda_n)) = PH(\vec{\alpha}, A)$$

To transform an *APH*-distribution into its ordered bidiagonal representation, the *spectral polynomial algorithm* (*SPA*)[2] is used. The algorithm's complexity is $O(n^3)$ where n is the number of states.

Cox Representation

The *Cox representation* is another canonical representation for *PH*-distributions. A *Cox generator* is a *PH*-generator that is used to build the Cox representation of a *PH*-distribution. Let $0 \leq p_i < 1$, for $1 \leq i \leq n-1$. A Cox generator has the following form:

$$\begin{bmatrix} -\lambda_1 & p_1\lambda_1 & 0 & \dots & 0 \\ 0 & -\lambda_2 & p_2\lambda_2 & \dots & 0 \\ 0 & 0 & -\lambda_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & -\lambda_n \end{bmatrix}$$

In this case, the exit rates are in *descending order.* The notation for the Cox generator is $Cx([\lambda_1, p_1], [\lambda_2, p_2], \dots, [\lambda_{n-1}, p_{n-1}], \lambda_n)$. The advantage of this representation over the bidiagonal representation is, that the initial distribution of the Cox representation is a *Dirac distribution* to the highest exit rate state. We define another theorem now:

Theorem 2. *Let $(\vec{\beta}, Bi(\lambda_1, \lambda_2, \dots, \lambda_n))$ be an ordered bidiagonal representation, and let vector $\vec{\delta} = [1, 0, \dots, 0]$. Then there exists a unique Cox representation*

$$(\vec{\delta}, Cx([\lambda_n, x_n], [\lambda_{n-1}, x_{n-1}], \dots, \lambda_1))$$

such that:

$$PH(\vec{\delta}, Cx([\lambda_n, x_n], [\lambda_{n-1}, x_{n-1}], \dots, \lambda_1))) = PH(\vec{\beta}, Bi(\lambda_1, \lambda_2, \dots, \lambda_n))$$

The vector $\vec{x}$ determining the Cox representation is derived from vector $\vec{\beta}$ by:

$$x_i = 1 - \beta_i \prod_{j=i+1}^{n} \tfrac{1}{x_j}, \text{ for } 2 \leq i \leq n$$

Considering these two theorems, we know that every *APH*-distribution can be transformed into an equivalent distribution in both ordered bidiagonal and Cox representation.

2.1.4 Stochastic Operators

When working with *APH*-distributions, three stochastic operators are commonly used. Let X_1 and X_2 be two independent random variables. Then

- $X_{max} = max\{X_1, X_2\}$ is the *maximum,*

- $X_{min} = min\{X_1, X_2\}$ the *minimum* and
- $X_{con} = con\{X_1, X_2\}$ the *convolution*

of the two random variables. These operators can be applied to PH-distributions. Let $F(t)$ and $G(t)$ be two distribution functions. The maximum, minimum and convolution of $F(t)$ and $G(t)$ are computed by

- $max(F(t), G(t)) = F(t)G(t)$
- $min(F(t), G(t)) = 1 - (1 - F(t))(1 - G(t))$
- $con(F(t), G(t)) = \int_0^t F(t - x)G(x)dx$

When applying these operators to APH-distributions, the resulting distributions are also APH-distributions.

Theorem 3. *Let $PH(\vec{\alpha}, A)$ and $PH(\vec{\beta}, B)$ be two* PH-*distributions of size n and m. Then:*

1. *their convolution is a* PH-*distribution $PH(\vec{\delta}, D)$ of size $m + n$, where:*

$$\vec{\delta} = [\vec{\alpha}, \alpha_{m+1}\vec{\beta}] \text{ and } D = \begin{bmatrix} A & A\vec{\beta} \\ 0 & B \end{bmatrix}$$

2. *their minimum is a* PH-*distribution $PH(\vec{\delta}, D)$ of size mn, where*[1]*:*

$$\vec{\delta} = \vec{\alpha} \otimes \vec{\beta} \text{ and } D = A \oplus B$$

3. *their maximum is a* PH-*distribution $PH(\vec{\delta}, D)$ of size $mn + m + n$, where:*

$$\delta = [\vec{\alpha} \otimes \vec{\beta},\ \beta_{n+1}\vec{\alpha},\ \alpha_{n+1}\vec{\beta}] \text{ and } D = \begin{bmatrix} A \oplus B & I_A \otimes B & A \otimes I_B \\ 0 & A & 0 \\ 0 & 0 & B \end{bmatrix}$$

All three operators are *associative* and *commutative.* Although the PH-generator and the initial distribution may vary when applying associativity and commutativity, the corresponding distribution functions stay the same.

[1] $\oplus$ and $\otimes$ denote the Kronecker sum and product operators, respectively.

2.2 Cox & Convenience Calculus

The Cox representation is one way to represent APH-distributions with simple graph representations. It is possible to depict every APH-distribution in Cox representation by a calculus. This section introduces a calculus developed by Reza Pulungan and Holger Hermanns [5]. The *Cox & Convenience Calculus* (CCC) is able to represent every APH-distribution in its Cox representation and connect APH-distributions with the stochastic operators introduced in the last section.

Each term built by the following grammar is a CCC delay:

$$P :== \quad \lambda \quad | \quad \mu \lhd \lambda.P \quad | \quad P \, ; \, P \quad | \quad P \oplus P \quad | \quad P \, || \, P$$

where $\lambda \in \mathbb{R}_+$ and $\mu \in \mathbb{R}_{\geq 0}$ are rates. The calculus consists of two parts, the *Cox* and the *Convenience Part.* The *Cox* part is used to build simple distributions in Cox representation. The *Convenience* part provides the possibility to manipulate the distributions using three operators.

2.2.1 Basic Delay

The basic block for every CCC term is a simple *exponentially distributed delay*, expressed by $\lambda \in \mathbb{R}_+$. The semantic behind this is an activity that induces a delay distributed according to an exponential distribution with rate λ and then terminates. The rule of inference for this basic case is defined as follows:

$$\frac{}{\lambda \xrightarrow{\lambda} \mathit{stop}}$$

In CCC, *stop* is the terminal symbol describing that the absorbing state is reached.

2.2.2 Cox Operator

The *Cox operator* ($\lhd$) is the only operator that is needed to build expressions representing every APH-distribution in its Cox representation. The operator takes the form $\mu \lhd \lambda.P$, where μ presents the preemptive termination delay unless delay λ occurs first. Intuitively, there is a transition μ leading immediately to the absorbing state and another one λ going out to the graph built by P. The rules of inference for the Cox operator are defined as follows:

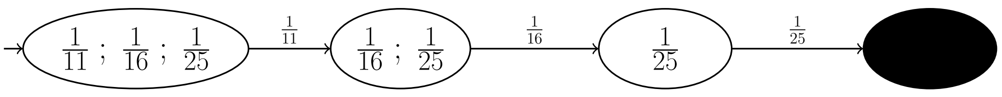
$$\frac{}{\mu \lhd \lambda.P \xrightarrow{\lambda} \mathit{stop}} \qquad \frac{}{\mu \lhd \lambda.P \xrightarrow{\lambda} P}$$

2.2.3 Sequential Composition

Although basic delays and the Cox operator suffice to build every *APH*-distribution, the Convenience part of *CCC* provides further operators to connect and manipulate different delays. The *sequential composition operator* (;) provides the opportunity to string together several distributions. Intuitively, this means that if the first operand reaches its absorbing state, the second operand starts and has to be completed. This leads to the following rules of inference:

$$\frac{P \xrightarrow{\lambda} P' \qquad P' \neq \mathit{stop}}{P \;;\; Q \xrightarrow{\lambda} P' \;;\; Q} \qquad \frac{P \xrightarrow{\lambda} \mathit{stop}}{P \;;\; Q \xrightarrow{\lambda} Q}$$

Example. The *CCC* expression $\frac{1}{11} \;;\; \frac{1}{16} \;;\; \frac{1}{25}$ evaluates to the following *CTMC*:


$\frac{1}{11} \;;\; \frac{1}{16} \;;\; \frac{1}{25}$
$\frac{1}{11}$
$\frac{1}{16} \;;\; \frac{1}{25}$
$\frac{1}{16}$
$\frac{1}{25}$
$\frac{1}{25}$

2.2.4 Parallel Composition

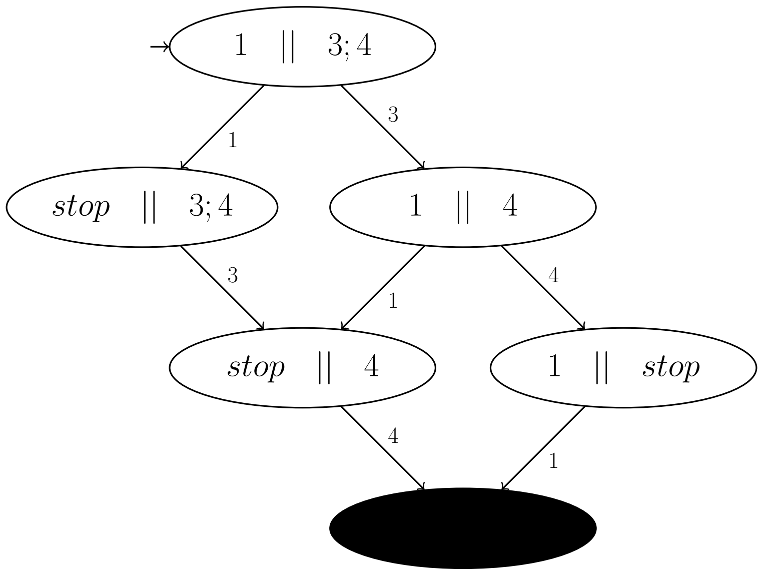
The *parallel composition operator* (||) provides another opportunity to join two *CCC* expressions. Using this operator, *CCC* is able to connect two distributions and let them proceed concurrently. It acts similar to the parallel operator in *CCS* developed by Robin Millner. Considering this behaviour, *CCC* provides the following rules of inference:

$$\frac{P \xrightarrow{\lambda} P'}{P \,||\, Q \xrightarrow{\lambda} P' \,||\, Q} \qquad \frac{Q \xrightarrow{\lambda} Q'}{P \,||\, Q \xrightarrow{\lambda} P \,||\, Q'}$$

The parallel composition operator is a static operator, which would lead to several absorbing states. Because we defined *APH*-distributions with exactly one absorbing state, we need the following assumption:

$$\mathit{stop} \,||\, \mathit{stop} \;\equiv\; \mathit{stop}$$

Example. The *CCC* expression 1 || 3 ; 4 evaluates to the following *CTMC*:



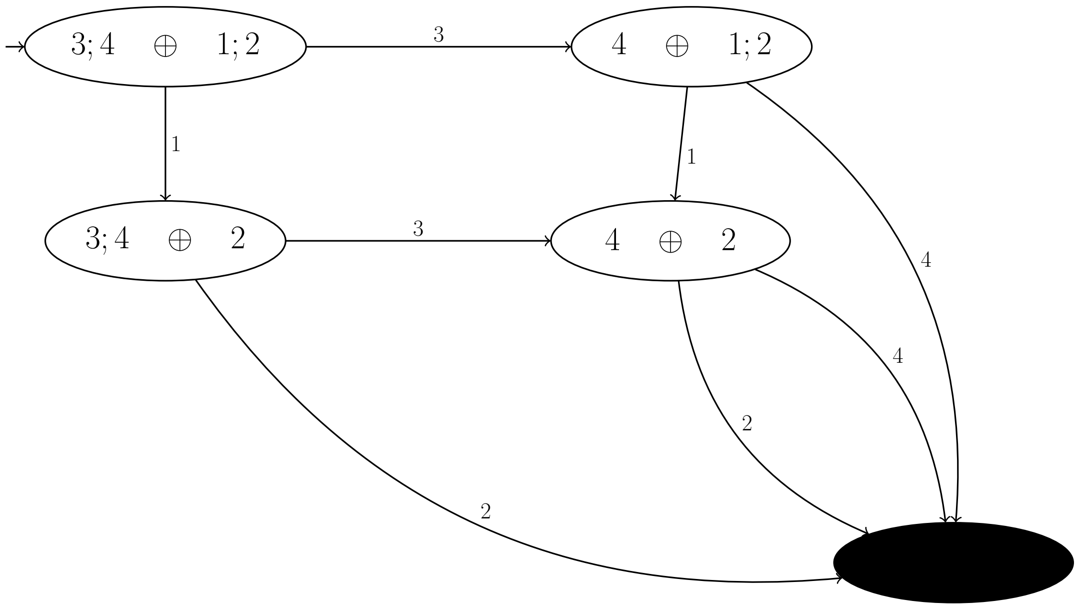
2.2.5 Choice Operator

The *choice operator* ($\oplus$) is a little different to the one in *CCS*. The choice which component to use doesn't need to be made at the beginning. Every component can take its own transitions until one of them terminates. This leads to the following rules of inference:

$$\frac{P \xrightarrow{\lambda} P' \qquad P' \neq stop}{P \ \oplus \ Q \xrightarrow{\lambda} P' \ \oplus \ Q} \qquad \frac{Q \xrightarrow{\lambda} Q' \qquad Q' \neq stop}{P \ \oplus \ Q \xrightarrow{\lambda} P \ \oplus \ Q'}$$

$$\frac{P \xrightarrow{\lambda} stop}{P \ \oplus \ Q \xrightarrow{\lambda} stop} \qquad \frac{Q \xrightarrow{\lambda} stop}{P \ \oplus \ Q \xrightarrow{\lambda} stop}$$

Example. The *CCC* expression $3 \ ; \ 4 \oplus 1 \ ; \ 2$ evaluates to the following *CTMC*:



The operators of the Convenience part represent the stochastic operators we defined in the previous section. The following theorem holds:

Theorem 4. *For all delays $P, Q \in CCC$:*

1. $con(PH(P), PH(Q)) = PH(P; Q)$,
2. $min(PH(P), PH(Q)) = PH(P \oplus Q)$ *and*
3. $max(PH(P), PH(Q)) = PH(P||Q)$.

We can conclude that for every $P \in CCC$, $M_P = (S, Q, \pi)$ is a $CTMC$ underlying an acyclic phase-type representation with Dirac initial distribution.

3 Minimization of PH-Distributions

With the stochastic operators presented in the previous chapter, we are able to build large *APH*-distributions. The size of these *APH*-distributions have a large influence on the computation time when analyzing them and working with them. This chapter presents a minimization algorithm developed by Reza Pulungan and Holger Hermanns [4].

3.1 Minimization Algorithm

The representation of a *PH*-distribution that is mainly used by the minimization algorithm is the *Laplace-Stieltjes transform* (LST).

Definition 1 (Laplace-Stieltjes transform)**.** *A* PH-*distribution can be characterized by its Laplace-Stieltjes transform:*

$$\tilde{f}(s) = \vec{\alpha}(sI - A)^{-1}\vec{A} + \alpha_{n+1},\ s \in \mathbb{R}_0^+$$

where I is the n-dimensional identity matrix.

The LST of an exponential distribution with rate λ is given by $\tilde{f} = \frac{\lambda}{s+\lambda}$. Let $L(\lambda) = \frac{s+\lambda}{\lambda}$. We call $L(\lambda)$ the *L-term* of λ. The LST of a PH distribution $Bi(\vec{\beta}, (\lambda_1, \lambda_2, \ldots, \lambda_n))$ is given by:

$$\tilde{f}(s) = \frac{\beta_1}{L(\lambda_1)\ldots L(\lambda_n)} + \frac{\beta_2}{L(\lambda_2)\ldots L(\lambda_n)} + \cdots + \frac{\beta_n}{L(\lambda_n)} = \frac{\beta_1 + \beta_2 L(\lambda_1) + \cdots + \beta_n L(\lambda_1)\ldots L(\lambda_n)}{L(\lambda_1)L(\lambda_2)\ldots L(\lambda_n)}$$

We already know, that every *APH* distribution can be transformed into its bidiagonal representation using *SPA* in $O(n^3)$. So we can bring every *APH* distribution into this LST form. The aim of the algorithm is to remove states of the distribution, i.e. to find common L-terms in both the numerator and determinator polynomials.

Theorem 5. *If for some $1 \leq i \leq n$, $\beta_1 + \beta_2 L(\lambda_1) + \cdots + \beta_i L(\lambda_{i-1})$ is divisible by $L(\lambda_i)$ and the resulting vector $\vec{\delta}$ is substochastic (i.e. $\delta_i \geq 0$ for all $1 \leq i \leq n$ and $\vec{\delta}\vec{1} \leq 1$), then the following equation holds:*

$$PH(\vec{\beta}, Bi(\lambda_1, \ldots, \lambda_n)) = PH(\vec{\delta}, Bi(\lambda_1, \ldots, \lambda_{i-1}, \lambda_{i+1}, \ldots, \lambda_n))$$

The complete minimization algorithm for a given *APH* distribution $(\vec{\alpha}, A)$ works as follows:

1. Use *SPA* to turn $(\vec{\alpha}, A)$ into its bidiagonal representation $(\vec{\beta}, Bi(\lambda_1, \lambda_2, \ldots, \lambda_n))$ which takes $O(n^3)$ time.
2. Set i to 2.
3. While $i \leq n$
   a) Check divisibility w.r.t. λ_i which takes $O(n)$.
   b) • If not divisible: set i to $i + 1$ and continue while-loop.
      • If divisible: If possible, eliminate λ_i as described in [4] and decrease n by 1.
        If elimination isn't possible, don't decrease n and continue with $(\vec{\beta}, Bi(\lambda_1, \lambda_2, \ldots, \lambda_n))$ and with i increased by 1.
4. Return $(\vec{\beta}, Bi(\lambda_1, \ldots, \lambda_n))$

This algorithm ensures minimality only for *acyclic-ideal* distributions. The Cox part of *CCC* ensures that only acyclic-ideal distributions can be built. The other operators don't ensure that always. However, the following theorem holds for the stochastic operators:

Theorem 6. *[4][Lemma14] Convolution, maximum and minimum operations are acyclic-ideal preserving almost every time.*

This theorem ensures, that the algorithm works for distributions built with *CCC* almost every time.

3.2 APHMIN Tool

APHMIN is a collection of tools developed by Reza Pulungan [5]. It implements the minimization algorithm presented in the previous section. The webinterface for using the tool suite can be found at `http://depend.cs.uni-sb.de/tools/aphmin/`.

3.2.1 Input

As input, *APHMIN* accepts every expression built with a special grammar. The grammar is quite similar to *CCC*, with small differences:

$$P ::= \quad exp(\lambda) \mid erl(n, \lambda) \mid cox(\mu, \lambda, P) \mid con(P, P) \mid min(P, P) \mid max(P, P)$$

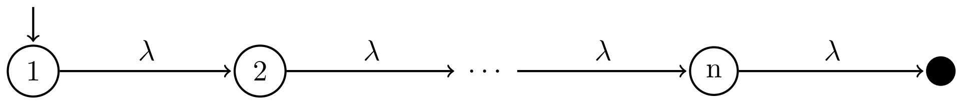


Figure 3.1: Graph of the Erlang distribution $Erl(n, \lambda)$

The grammar is more or less a prefix-notation of *CCC*. The *max*, *min*, *con* and *Cox* operator correspond to the *CCC* operators ($||$), ($\oplus$), ($;$) and ($\triangleleft$). To build large graphs more user friendly, the grammar offers the possibility to use *Erlang distributions*. Erlang distributions represent exponential distributions with the same rate, sequentially stringed together.

Definition 2 (Erlang distributions)**.** *An Erlang Distribution $Erl(n, \lambda)$ with* rate λ *and shape n can be represented by an acyclic* PH-*distribution $Bi(\underbrace{\lambda, \lambda, \ldots, \lambda}_{n})$. The graph of the Erlang distribution $Erl(n, \lambda)$ is depicted in Figure 3.1.*

An important fact is that the operators are defined as binary operators, similar to the mathematical definition in the previous chapter. This works well for the webinterface, but leads to an uncomfortable work flow using the app later.

3.2.2 Output

The *APHMIN* tool suite is wrapped into a simple webservice. Minimization requests can be sent using the SOAP 1.1 protocol via HTTP. It offers a single operation `aphmin`, which receives an input expression and delivers the minimized representation. The tool either responds immediately or can send the minimized result via email. The tool also offers the user information about the computation time, the number of states in the original graph and also the number of states in the minimal representation. When receiving a response via email, *APHMIN* also adds a ".tra" (a simple list of all transitions) or ".cox" (the expression in cox representation) file of the minimized representation. The WSDL-file describing the web-service can be found at: `http://ada.cs.uni-saarland.de/aphminserver.php?wsdl`.

4 The Android App

Having set the basics, we now take a closer look at the actual aim of the thesis. Inspired by Drag & Drop applications on mobile devices, we developed an app for Android OS to use the *APHMIN* tool from every Android device. The design is based on the folder system design of the Android OS using a simple Drag & Drop mechanism. The app uses nested boxes to create and display *CCC* expressions.

4.1 Basic Design Ideas

The Android OS provides the opportunity to organize its apps in folders. The design idea for this feature is quite simple. A simple click on the folder symbol opens a small frame which displays the apps contained by the folder. This design structure needs to be modified for our purpose. The app should display every delay and operator by a different box. Because *CCC* is able to build expressions of any depth, the app also needs to build nested boxes in contrast to the folders in Android, which can only contain apps, not any other folders. A toolbar containing the different operators and basic delays is the basis of the Drag & Drop mechanism. The expressions should be built by simply dragging an operator or delay and dropping it into an already existing box representing another operator. The actual minimization algorithm is executed by *APHMIN* on the server, because it may need more resources than a mobile device could deliver. So the app needs to send the expression to the tool via Internet connection. To avoid errors, the app should also ensure that only valid expressions should be sent to *APHMIN*. The graphical design of the app is shown in Figure 4.1.

We have already seen, that *CCC* defines sequential composition, parallel composition and the choice operator as *binary operators* because these operators correspond to the stochastic operators *min*, *max* and *con*. We have also seen that the grammar *APHMIN* uses is defined in the same way. Because the operators are associative and commutative, they are commonly used as n-ary operators in mathematics. Because of that, the app should also be able to build expressions with n-ary operators.

We now give a short overview over the implementation of the app. We describe the data structure, the graphical design and the underlying classes with their functions. We

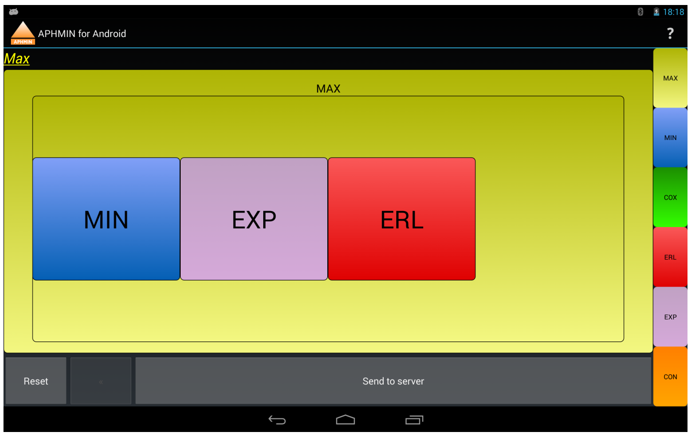


Figure 4.1: *APHMIN* App

will only describe the most important ones.

4.2 User Interface and Semantics

The user interface is implemented by several classes that already contain the underlying semantics. This works because the tree structure that Android uses to build its user interfaces can be easily used for this purpose.

4.2.1 Operator and subclasses

The boxes that represent the different operators are objects of the class *Operator*, a subclass of the Android API class *LinearLayout*. We have to take a closer look at every possible operator and basic delay to understand the class structure.

- ***max*, *min* and *con* operator**: These operators should be designed as n-ary operators. So the design needs an area where the user can drop basic delays or other operators. These subexpressions need to be saved in a *List* field because the number of children isn't predefined.
- ***Cox* operator**: The Cox operator also needs an area to drop a subexpression. But it can only contain one subexpression, so it simply needs an *Operator* field to save this single subexpression.
- ***Exponential* and *Erlang* distributions** These basic blocks don't need an area to drop delays or operators. They can't contain any subexpressions and simply

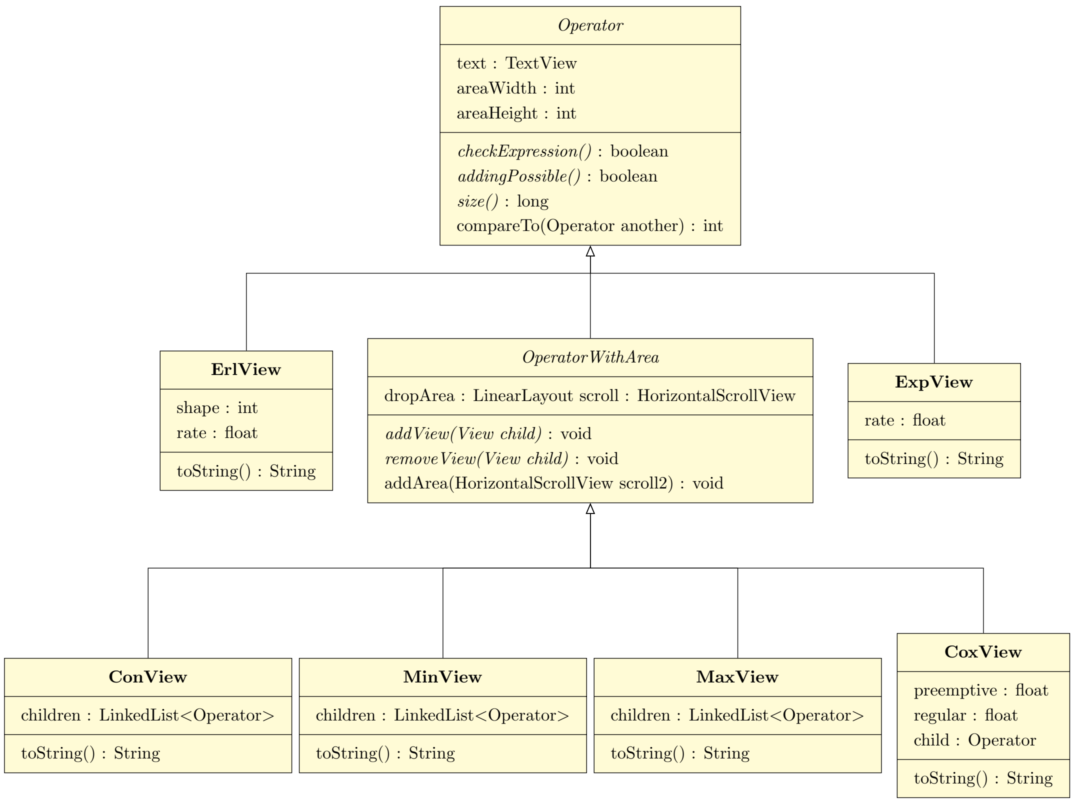


Figure 4.2: UML diagram of the operator types

need one or rather two fields that contain the transition rates.

These operators are implemented as shown in Figure 4.2. At first, we take a closer look at a part of the abstract class *Operator*.

```
public abstract class Operator extends LinearLayout implements
Comparable<Operator> {

        public TextView text;
        public Boxes parent;
        protected int areaWidth;
        protected int areaHeight;

        public abstract boolean checkExpression();
        public abstract boolean addingPossible();
        public abstract long size();

        public int compareTo(Operator another) {
                return (int) (another.size() - size());
        }

        //...

}
```

In this abstract class, all fields that are needed by every kind of operator are implemented. The *TextView text* will simply indicate in the user interface, what operation or delay is represented. It is not necessary to implement a field *child*, because the basic and Erlang delays don't have any children and form the leaves of the tree.

Operator also implies two abstract methods. The method *checkExpression()* checks, if a box already builds a correct and complete *CCC* expression. This method is used to visualize if we can already send the expression to the server. The method *addingPossible()* checks, if there still can be added any operators or delays to the referred box.

The abstract class *OperatorWithArea* extends the class *Operator*. Additional fields and methods are added in this subclass.

```
public abstract class OperatorWithArea extends Operator {

        public LinearLayout dropArea;
        public HorizontalScrollView scroll;

        @Override
        public abstract void addView(View child);

        //...

}
```

The objects *dropArea* and *scroll* represent the area, where the user can drop new operators or delays. We need both of them because Android requires to put the scroll view into a *LinearLayout*. The method *addView()* is used to add new operators to the area. It is abstract because it is implemented a little different for each operator. The Cox operator displays its child in the center, because no other children can be added.

The different explicit classes implement the abstract methods and contain additional fields that are required for the different kind of delays and operators. The classes for the graphical interface also imply the underlying semantics for the expressions. Because the structure of *CCC* is not that complex, this doesn't lead to any problems. The *toString()* of the subclasses of *Operator* are implemented recursively and return a *String* representation of the expression according to the grammar used by *APHMIN*.

4.2.2 Icon

In the next step, we focus on the navigation through the expression tree. Because of that, a class *Icon* is implemented which is a subclass of *LinearLayout*. These graphical objects are used inside the regular *Operator* objects to lead to the corresponding subexpression and to display them. There aren't any subclasses of *Icon* because we simply change the background and the text for the different kinds of operators and delays. The class simply consists of two fields, referring to the corresponding *Operator* object and the *TextView* displaying the type of the operator.

4.2.3 Aphmin

The class *Aphmin* implements the main *Activity* of the app. The interface is designed as shown in Figure 4.1. The class provides several methods to interact with the interface. The most important ones are:

- ***refreshBreadCrumbs()*:** This method refreshes the breadcrumbs that are used to navigate through the expression tree.
- ***reset()*, *oneLevelUp()* and *calculateResult()*:** The methods that are used by the buttons at the bottom of the *Activity*.
- ***editExpression()* and *removeExpression()*:** The methods that are used by some PopUp menus which allow the user to edit and remove subexpressions.

4.3 Connection to Webservice

The *APHMIN* tool provides a webservice to make it usable for other tools. The data exchange between the server and the app uses the SOAP protocol. The Android SDK doesn't offer a possibility to use SOAP connections. So we need to use a third party library called *ksoap2-android* [1]. The connection to the webservice is implemented in the class *AphminRequestTask*. The class extends the class *AsyncTask*, so the request is sent

[1]The library is available under `https://code.google.com/p/ksoap2-android/`

in the background of the application. The body of the SOAP object built by the app looks as follows:

```
<v:Body>
<n0:aphmin id="o0" c:root="1" xmlns:n0="urn:APHMIN">
        <expression i:type="d:string">exp(5.0)</expression>
        <expression i:type="d:string">aphmin@aphmin.net</
            expression>
</n0:aphmin>
</v:Body>
```

The webservice only offers the operation *aphmin*. In the example shown above, an email address was added to the request. It is also possible to send a request without an email address. In this case, the response of the webservice looks like this:

```
<SOAP-ENV:Body>
<ns1:aphminResponse>
<return xsi:type="ns2:Map">
<item>
        <key xsi:type="xsd:string">computationtime</key>
        <value xsi:type="xsd:float">0.010884046554565</value>
</item>
<item>
        <key xsi:type="xsd:string">originalsize</key>
        <value xsi:type="xsd:int">2</value>
</item>
<item>
        <key xsi:type="xsd:string">minimizedsize</key>
        <value xsi:type="xsd:int">2</value>
</item>
<item>
        <key xsi:type="xsd:string">model</key>
        <value xsi:type="xsd:string">exp(5)</value>
        </item>
</return>
</ns1:aphminResponse>
</SOAP-ENV:Body>
```

The response message returned by the webservice contains the same pieces of information that are displayed by the webinterface. Whenever the request contains an email address, the values are simply set to zero or contain no value. *AphminRequestTask* also implements the method *writeResultFile*. For every request, the app also creates a new file on the device which contains the original and the minimized expression.

5 Dealing with n-ary Operators

The *max*, *min* and *con* operators are displayed in the app as n-ary operators. In the already existing *APHMIN* tool, these operators are implemented as binary operators, so the tool cannot handle expressions built with n-ary operators. The aim of this chapter is to find a method to break expressions with n-ary operators down to expressions consisting only of binary operators.

5.1 Intuitional Approach

When we recap the definitions of the *max*, *min* and *con* operator, these operators are *associative* and *commutative.* In the following, we will use $S \in \{max, min, con\}$ to represent this operators. We also use S_b as the binary version of an operator S, so $S_b \in \{max_b, min_b, con_b\}$. The associativity and commutativity of the operators lead to a first intuitional approach.

Approach 1. In a *CCC* expression M, every n-ary operator S which is associative and commutative is replaced in the following way:

$$S(x_1, x_2, x_3, \ldots, x_n) = (\ldots(x_1\ S_b\ x_2)\ S_b\ x_3) \ldots\ S_b\ x_n$$

with $x_i \in CCC$ with $1 \leq i \leq n$.

This approach intuitively means that we interpret S as a *left-associative* version of $n-1$ binary operators S_b. Because S is associative, this doesn't lead to any problems. Another possibility is to interpret S as a *right-associative* version of $n-1$ binary operators S_b.

Approach 2. In a *CCC* expression M, every n-ary operator S which is associative and commutative is replaced in the following way:

$$S(x_1, x_2, x_3 \ldots x_n) = x_1\ S_b\ (x_2\ S_b\ (x_3\ S_b \ldots (x_{n-1}\ S_b\ x_n))) \ldots)$$

with $x_i \in CCC$ with $1 \leq i \leq n$.

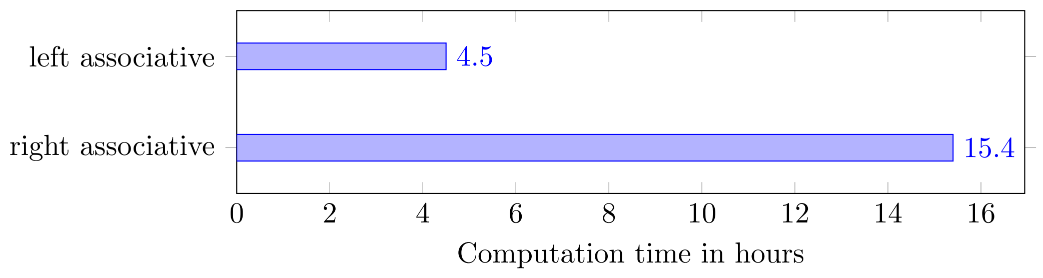
At first sight, these two approaches look quite similar. We compare the two approaches on an example.

Example. We analyze the expression $max(erl(20,4), erl(20,5), erl(10,3))$. The graph represented by the expression consists of 4851 states. The two approaches replace the *max*-operator by its binary version. They lead to the following two expressions:

1. $\boxed{\text{max(} \boxed{\text{max(erl(20,4), erl(20,5))}} \text{ , erl(10,3))}}$

2. $\boxed{\text{max(erl(20,4), } \boxed{\text{max (erl(20,5), erl(10,3))}} \text{)}}$

These two expressions represent the same distribution and can be minimized by the *APHMIN* tool. We compare the computation time that *APHMIN* needs to minimize the two expressions:



The result *APHMIN* delivers for both expressions are equal. The minimal representation of the *CTMC* consists of 196 states. The example also shows that the computation times differ a lot. In this particular example, the second approach needs more then 3 times as much computation time as the first approach. We see that the way the parenthesis are set in the subexpressions has a strong influence on the runtime. The next step is to find a good heuristic to transform n-ary into binary operators to reduce the computation time of the minimization algorithm.

5.2 Basic Ideas

To get a basic idea for a heuristic, it is important to understand how *APHMIN* exactly works. The parser needs to convert the expressions built by the user into ".tra" files which describe the graphs that represent the expressions. The parser works recursively, so that the subexpressions are converted step by step. For every operator, *APHMIN* doesn't only convert the expression to the ".tra" format, it immediately minimizes the graph. This paradigm is called *divide and conquer*. A quite similar idea is used in [1]. *APHMIN* is able to reduce the computation time a lot using this paradigm. In the following, we use $Red(\cdot)$ to name the minimization algorithm of *APHMIN*.

Let $S_b \in \{max_b, min_b, con_b\}$. We assume that we want to compute

$$Red(S_b(x_1, S_b(x_2, S_b(\dots S_b(x_{n-1}, x_n)\dots)$$

Whenever *APHMIN* parses a subexpression into a ".tra" file, it immediately minimizes it. *APHMIN* computes the minimization of the above expression as follows:

$$Red(S_b(Red(x_1), Red(S_b(\dots Red(S_b(Red(x_{n-1}), Red(x_n)))\dots)$$

This paradigm leads to the differences in the computation time of the first two approaches. Because the runtime of the algorithm depends on the number of states, the heuristic should reduce the state size as early as possible. The two most important criteria to improve the runtime seem to be the *size of the subexpressions* and the *density of certain transitions.*

The heuristic should break down an n-ary operator S to $n-1$ pairs according to the following procedure:

1. The first pair consists of two operands
2. The i-th pair (with $1 < i \leq n$) either consists of a previously built group and one operand not already assigned to another group, two previously built groups or two operands not already assigned to another group.

We now consider different aspects and criteria for choosing the order of the operands when building the pairs.

5.3 Size of Operands

An important criterion for the runtime seems to be the size of the operands i.e. the number of states. We're going to analyze differences in computation time for every n-ary operator provided by *CCC*. In the experiments that follow, we apply the minimization algorithm multiple times and display the average computation time. The section only shows a small extract of the most significant results. Expressions with different sizes and structures were analyzed and all lead to similar results.

5.3.1 Maximum Operator

At first, we take a look at the *maximum operator*. To simplify the experiments, we use expressions of the form $max(erl(k_1, \lambda_1), erl(k_2, \lambda_2), \ldots, erl(k_n, \lambda_n))$. At first, we don't change the order of the operands and use only Erlang distributions with the same transition rates. This avoids that different transition rates affect the computation times. Running the *APHMIN* tool with different expressions and groupings lead to the following results:

Expression 1

The first expression is $max(erl(5, 1), erl(20, 1), erl(50, 1))$. In this case, we got three operands. This leads to the following two ways to parenthesise the sub expressions:

- $\boxed{\text{max(}\boxed{\text{max(erl(5,1),erl(20,1))}}\text{, erl(50,1))}}$ displayed by 🟥
- $\boxed{\text{max(erl(5,1),}\boxed{\text{max(erl(20,1),erl(50,1))}}\text{)}}$ displayed by 🟦

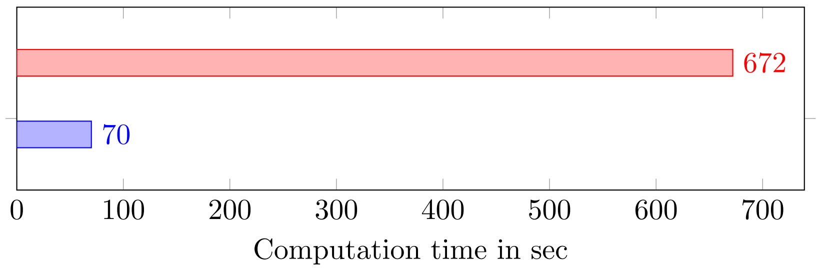


Figure 5.1: Expression 1

We can see in Figure 5.1 that the computation time diversifies a lot. Parenthesising the large distributions first leads to a much faster computation time. In this expression, the Erlang distributions are in descending order regarding to their size.

Expression 2

The next expression consists of four operands in ascending order. The expression $max(erl(2, 0.2), erl(7, 0.2), erl(10, 0.2), erl(65, 0.2))$ offers more possibilities to build groups. We compare the computation times for the following expressions:

- $\boxed{\text{max(}\boxed{\text{max(}\boxed{\text{max(erl(2,0.2),erl(7,0.2))}}\text{,erl(10,0.2))}}\text{,erl(65,0.2))}}$ displayed by 🟪

- $\boxed{\mathrm{max}(\boxed{\mathrm{max}(\mathrm{erl}(2,0.2),\boxed{\mathrm{max}(\mathrm{erl}(7,0.2),\mathrm{erl}(10,0.2))})},\mathrm{erl}(65,0.2))}$ displayed by ⬛
- $\boxed{\mathrm{max}(\mathrm{erl}(2,0.2),\boxed{\mathrm{max}(\boxed{\mathrm{max}(\mathrm{erl}(7,0.2),\mathrm{erl}(10,0.2))},\mathrm{erl}(65,0.2))})}$ displayed by 🟫
- $\boxed{\mathrm{max}(\mathrm{erl}(2,0.2),\boxed{\mathrm{max}(\mathrm{erl}(7,0.2),\boxed{\mathrm{max}(\mathrm{erl}(10,0.2),\mathrm{erl}(65,0.2))})})}$ displayed by 🟥
- $\boxed{\mathrm{max}(\boxed{\mathrm{max}(\mathrm{erl}(2,0.2),\mathrm{erl}(7,0.2))},\boxed{\mathrm{max}(\mathrm{erl}(10,0.2),\mathrm{erl}(65,0.2))})}$ displayed by 🟦

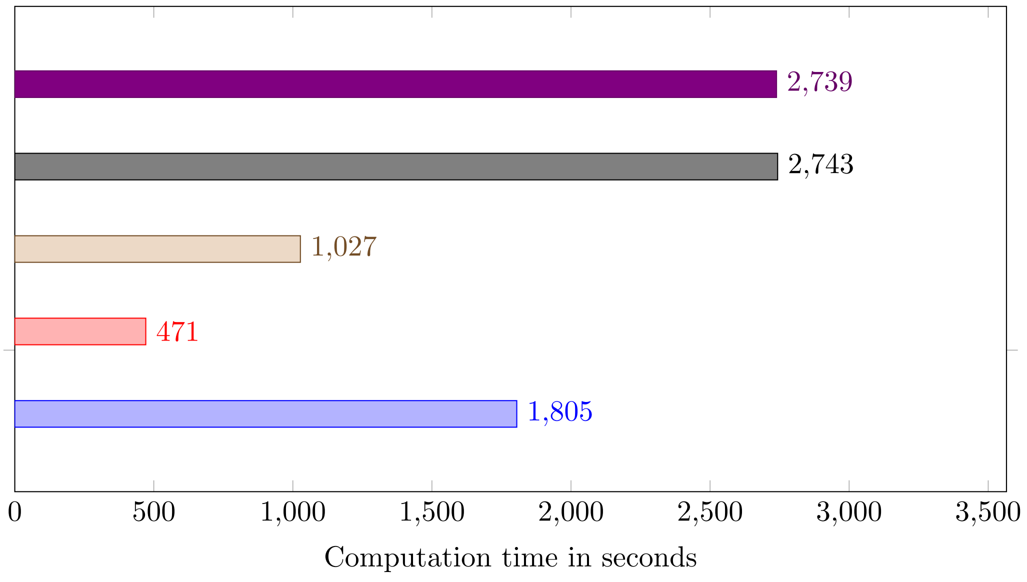


Figure 5.2: Expression 2

The results in Figure 5.2 underline the results of the previous example. In these expressions, the operands are ordered according to their size. We can see that the computation times vary a lot. We get the lowest computation times whenever we put the largest operands together first.

The experiments were repeated with different rate and shape parameters. We always achieve the best computation times when the largest operands are paired first. The differences in computation time grow together with the gaps of size between the different operands. The results didn't change for other distributions than Erlang distributions.

Another important aspect to look at is the order of the operands. We analyzed different orders of the subgroups to determine if they have an influence on the computation time.

The results showed that the order doesn't influence the computation time.

5.3.2 Minimum Operator

The next operator to analyze is the *minimal operator*. We analyze this operator with the same procedure as the *max* operator. Again, we use expressions of the form $min(erl(k_1, \lambda_1), erl(k_2, \lambda_2), \ldots, erl(k_n, \lambda_n))$.

Expression 3

We analyze a *min* operator with 4 operands first. The expression

$$min(erl(10, 5), erl(20, 5), erl(50, 5), erl(200, 5))$$

can be parenthesised in 4 different ways.

- $\boxed{\text{min(}\boxed{\text{min(}\boxed{\text{min(erl(10,5),erl(20,5))}}\text{,erl(50,5))}}\text{,erl(200,5))}}$ displayed by 🟪
- $\boxed{\text{min(}\boxed{\text{min(erl(10,5),}\boxed{\text{min(erl(20,5),erl(50,5))}}\text{)}}\text{,erl(200,5))}}$ displayed by ⬛
- $\boxed{\text{min(erl(10,5),}\boxed{\text{min(}\boxed{\text{min(erl(20,5),erl(50,5))}}\text{,erl(200,5))}}\text{)}}$ displayed by 🟫
- $\boxed{\text{min(erl(10,5),}\boxed{\text{min(erl(20,5),}\boxed{\text{min(erl(50,5),erl(200,5))}}\text{)}}\text{)}}$ displayed by 🟥
- $\boxed{\text{min(}\boxed{\text{min(erl(10,5),erl(20,5))}}\text{,}\boxed{\text{min(erl(50,5),erl(200,5))}}\text{)}}$ displayed by 🟦

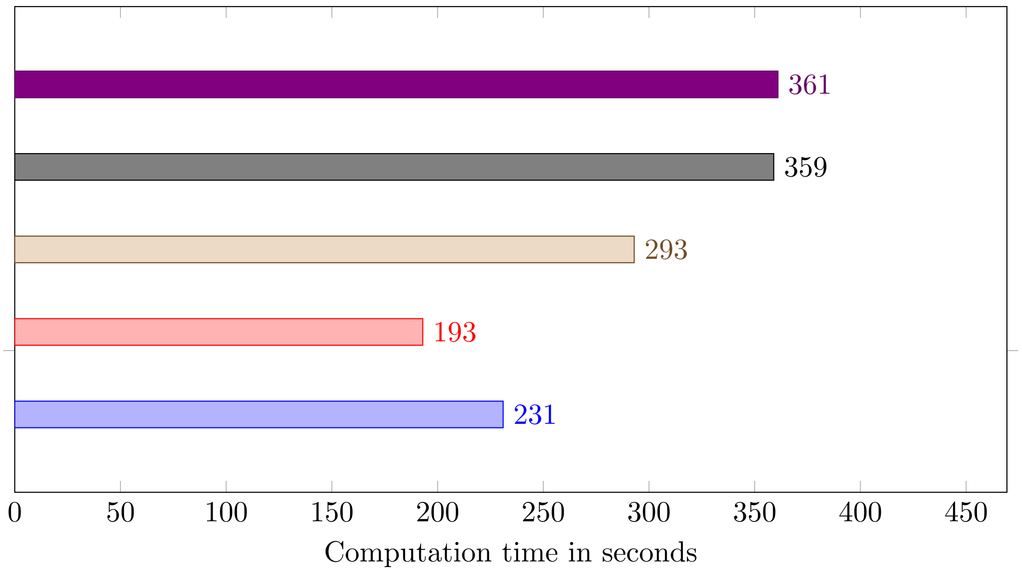


Figure 5.3: Expression 3

The differences in the computation time in Figure 5.3 aren't as large as the one of the *max* operator. But the best computation times are still received when the largest expressions are parenthesised first. We analyze again the order of the operands of the *min* operator.

Expression 4

We now analyze the same expression as before, but with inverse order of the operands.

- $\boxed{\text{min(}\boxed{\text{min(}\boxed{\text{min(erl(200,5),erl(50,5))}}\text{,erl(20,5))}}\text{,erl(10,5))}}$ displayed by 🟪
- $\boxed{\text{min(}\boxed{\text{min(erl(200,5),}\boxed{\text{min(erl(50,5),erl(20,5))}}\text{)}}\text{,erl(10,5))}}$ displayed by ⬛
- $\boxed{\text{min(erl(200,5),}\boxed{\text{min(}\boxed{\text{min(erl(50,5),erl(20,5))}}\text{,erl(10,5))}}\text{)}}$ displayed by 🟫
- $\boxed{\text{min(erl(200,5),}\boxed{\text{min(erl(50,5),}\boxed{\text{min(erl(20,5),erl(10,5))}}\text{)}}\text{)}}$ displayed by 🟥
- $\boxed{\text{min(}\boxed{\text{min(erl(200,5),erl(50,5))}}\text{,}\boxed{\text{min(erl(20,5),erl(10,5))}}\text{)}}$ displayed by 🟦

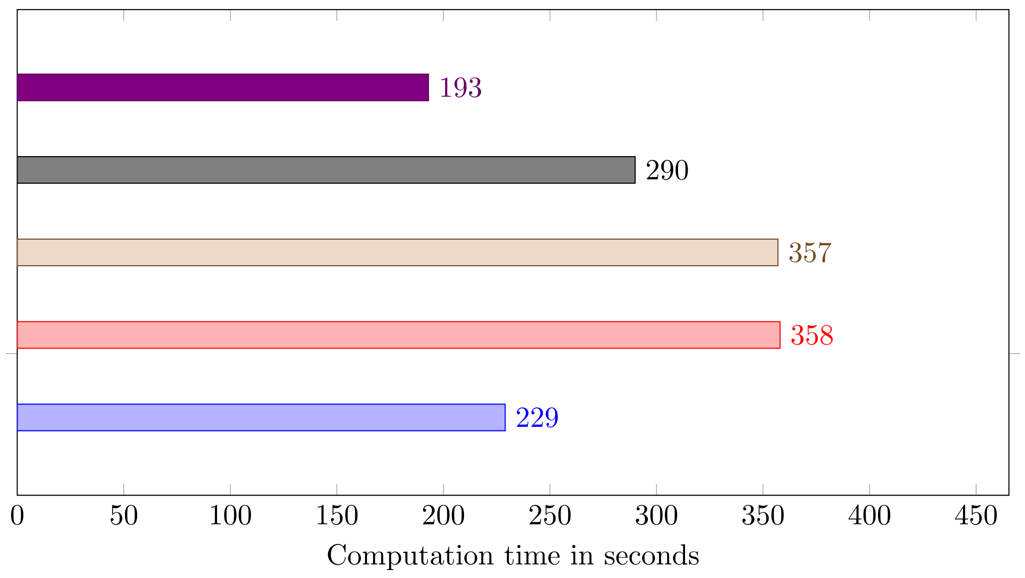


Figure 5.4: Expression 4

The expressions show again that also for the *min* operator, the influence of the order of the different groups is negligible. The best strategy for the *min* operator seems to be equal to the one for the *max* operator.

5.3.3 Convolution Operator

The operands for the *convolution operator* have to be larger to get significant results. If we used expressions of the form

$$Con(Erl(k_1, \lambda_1), Erl(k_2, \lambda_2), \ldots, Erl(k_n, \lambda_n)),$$

the input expression would already be in bidiagonal representation. So we use other large operands to get significant results.

Convolution Expression 5

We use the following expressions to analyze the computation time:

- **Op1:** $Max(erl(10, 5), Erl(5, 5))$
- **Op2:** $Min(Erl(10, 5), Erl(20, 5))$
- **Op3:** $Max(Erl(10, 5), Erl(30, 5))$

- **Op4:** $Max(Erl(30, 5), Erl(80, 5))$

This computation times for the expression $Con(Op1, Op2, Op3, Op4)$ look as follows:

- $\boxed{\text{con}(\boxed{\text{con}(\boxed{\text{con(Op1, Op2)}}, \text{Op3})}, \text{Op4})}$ displayed by 🟪
- $\boxed{\text{con}(\boxed{\text{con(Op1, }\boxed{\text{con(Op2, Op3)}})}, \text{Op4})}$ displayed by ⬛
- $\boxed{\text{con(Op1, }\boxed{\text{con}(\boxed{\text{con(Op2, Op3)}}, \text{Op4})})}$ displayed by 🟫
- $\boxed{\text{con(Op1, }\boxed{\text{con(Op2, }\boxed{\text{con(Op3, Op4)}})})}$ displayed by 🟥
- $\boxed{\text{con}(\boxed{\text{con(Op1, Op2)}}, \boxed{\text{con(Op3, Op4)}})}$ displayed by 🟦

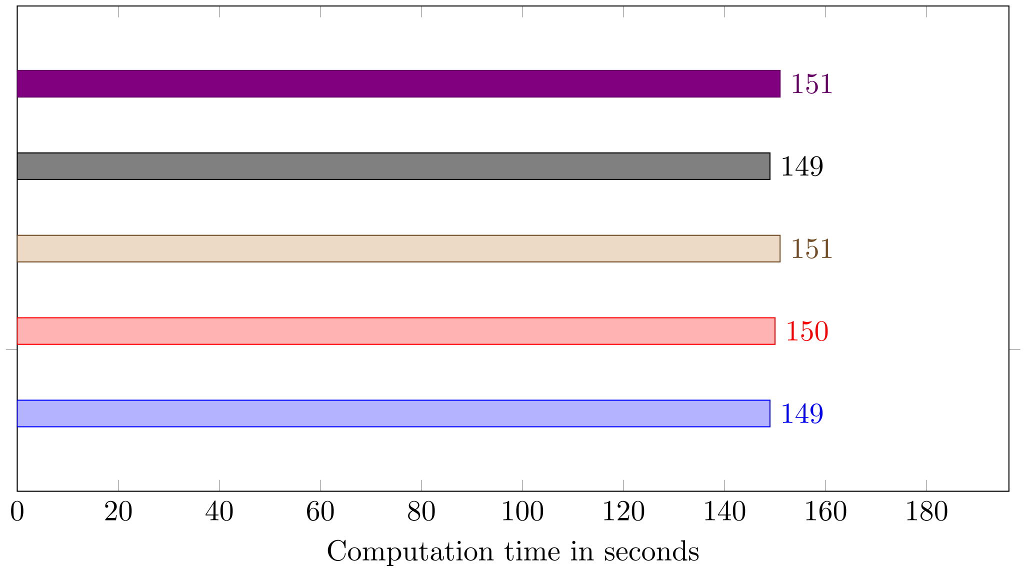


Figure 5.5: Expression 5

The results in Figure 5.5 clearly show that the way the different operands are put together don't influence the computation time. Other experiments with different operands underline this observation.

5.3.4 Results

The results show that for the *max* and *min* operator, the best computation time is achieved if the largest operands are put together first. For the *con* operator, the order of the operands doesn't influence the computation time. The reason for these differences is that applying the different operators lead to different sizes of the distributions. While the size of *max* and *min* is computed by multiplication ($mn+m+n$ and mn), the size of *con* is calculated simply by addition of the operand sizes ($m+n$). When minimizing the largest operands first, we keep the size small in early stages. This leads to good results in computation time.

5.4 Density of Transitions

Another interesting criterion to look at is the density of certain transitions inside the operators. We know that the size of the results the minimization algorithm delivers depends on the number of equal transitions the operands contain. We analyzed that parenthesizing groups with a lot of similar transitions lead to better results in computation time.

The results were similar to the ones considering the size of the operands. The best computation times for the *max* and *min* operators were achieved when parenthesizing groups with a large number of equal transitions first. For the *con* operator, there weren't any differences.

5.5 Transition Density vs. Size

We now have two criteria to parenthesize operands in n-ary operators. The next task is to prioritize these criteria.

The expressions used in the experiments were expressions of the form

$$S(Erl(k_1, \lambda), Erl(k_2, \lambda), Erl(k_3, \lambda), \ldots, Erl(k_{n-1}, \lambda), Erl(n, \lambda'))$$

where $S \in \{max, min\}$ and $k_1 < k_2 < \cdots < k_{n-1} < k_n$. We can see that the largest Erlang distribution is the only one with a different transition rate. The results for this kind of expressions all looked quite similar. We take a look at a specific example:

$$max(erl(2,5), erl(5,5), erl(10,5), erl(12,5), erl(25,12))$$

We compared two cases of parenthesizing:

- Ordered by the size of the operands

- Ordered by its transition rates 🟦

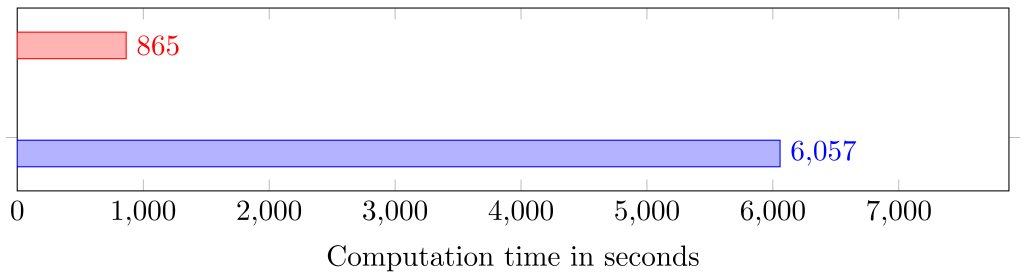


Figure 5.6: Comparison of size and density

We used several expressions to compare both criteria. Different transition rates were selected, and also distributed in several ways over the operands. The experiments showed that the size of the operands seems to be the better criterion. The influence of the size of the operands seems to be more important then the influence of the density of transitions. The app implements the two criteria, prioritized as described.

5.6 Implementation of the Heuristic

The results of the previous section lead to the following approach:

Approach 3. For every operator $S \in \{max, min\}$, the operands are ordered as follows:

1. The operands are ordered according to the density of the transitions
2. The number of states for every operand is computed recursively.
3. The operands are reordered in descending order according to their size. Operands of the same size stay in the same order as they were before.
4. The first group is built by the first two operands.
5. The i-th group is built by the $(i-1)$-th group and the i-th operand.

The operator *con* gets simply broken down to the binary operator con_b with *con* as a left-associative version of con_b.

This algorithm leads to good results in computation time. The algorithm is implemented in the *toString()* method of the different *Operator* implementations and in the class *Heuristic*. As an example, we take a look at the implementation of the *max* operator.

```
        public String toString() {
                Heuristic.sort(children);
                StringBuilder sb = new StringBuilder();
                for (int i = 1; i < children.size(); i++) {
                        sb.append("max(");
                }
                String prefix = "";
                String appendix = "";
                for (Box o : children) {
                        sb.append(prefix);
                        prefix = ",";
                        sb.append(o.toString());
                        sb.append(appendix);
                        appendix = ")";
                }
                return sb.toString();

        }

        public long size() {
                long s = 0;
                for (Box b : children) {
                        s = s * b.size() + s + b.size();
                }
                return s;
        }
```

The method *size()* computes the size of the subexpression recursively. Because *Operator* implements *Comparable*, the method *Collections.sort()* (called in the *sort()* method of *Heuristic*) sorts the operands according to their size.

The class *Heuristic* also implements the sorting of the operands according to their transition density. The implementation works as follows:

1. An $n \times n$ matrix M is created, where n is the number of operands. For two operands O and O', $M(O, O')$ describes the number of common transitions of O and O'.

2. The biggest entry of the matrix is searched. The corresponding operands are the

first ones in the order.

3. Line and column of the first operand are deleted from the matrix.
4. While the size of the matrix is bigger then 1
    a) Take the last added operand and search its matrix line for the biggest remaining entry.
    b) The line of the operand is deleted.
    c) The corresponding operand to the biggest entry is the next in the order.
5. Return the ordered list.

The complexity of the algorithm is $O(n^2)$ where n is the number of operands. We know that the complexity of the minimization algorithm is $O(n^3)$ where n is the number of states. For every operator, the number of states is at least as large as the number of operands, normally a lot larger. Therefore, the runtime of the heuristic is negligible compared to the runtime of the minimization algorithm, which depends on the number of states.

6 Evaluation & Conclusion

In this last chapter, we apply the developed heuristic to several expressions of different sizes and compare the computation times to the ones of the intuitional approach defined in Section 5.1. In the end, the chapter summarizes the observations made in this thesis and adds some remarks.

6.1 Evaluation

We want to compare now the computation times of the heuristic and the intuitional approach 1 from Section 5.1. To achieve large differences in computation time, we build the expressions in a way that the resulting transformations of the heuristic and the naive approach differ a lot. We first concentrate on the *max* operator to get the most significant results. Our expressions all have the following form $max(Op1, Op2, Op3, Op4)$ where only the different operands change. This leads to the results in Table 6.1.

Op1	**Op2**	**Op3**	**Op4**	**State Space**	**Computation time in sec**	
					Heuristic	Approach 1
$erl(1,1)$	$erl(2,2)$	$erl(2,3)$	$erl(25,4)$	468	6	33
$erl(1,1)$	$erl(2,2)$	$erl(5,3)$	$erl(50,4)$	1835	74	2985
$erl(1,1)$	$erl(3,2)$	$erl(5,3)$	$erl(50,4)$	2448	122	4010
$erl(1,1)$	$erl(3,2)$	$erl(5,3)$	$erl(60,4)$	2928	238	8443
$erl(1,1)$	$erl(3,2)$	$erl(5,3)$	$erl(70,4)$	3408	487	16487

Table 6.1: Different operations using the *max* operator

We can see that our heuristic gets much better results than the naive approach. For the large expressions, we reach computation times that are about 33-37 times faster than the naive approach.

We now also analyze expressions of the form $min(Op1, Op2, Op3, Op4)$. Table 6.2 shows the computation results.

Op1	**Op2**	**Op3**	**Op4**	**State Space**	**Computation time in sec**	
					Heuristic	Approach 1
$erl(4,1)$	$erl(6,2)$	$erl(20,3)$	$erl(200,4)$	96001	31	47
$erl(4,1)$	$erl(6,2)$	$erl(24,3)$	$erl(280,4)$	161281	84	120
$erl(4,1)$	$erl(6,2)$	$erl(24,3)$	$erl(400,4)$	230401	181	249
$erl(4,1)$	$erl(6,2)$	$erl(24,3)$	$erl(800,4)$	460801	882	1030
$erl(8,1)$	$erl(12,2)$	$erl(48,3)$	$erl(800,4)$	3686401	2883	4353

Table 6.2: Different operations using the *min* operator

The results underline the observations of the previous chapter. We still receive better computation times, but the differences aren't as large as the ones of the *max* operator.

6.2 Conclusion

Throughout the thesis, we developed an Android app that uses the *APHMIN* webservice and provides a user-friendly interface to build *CCC* expressions using Drag & Drop. The app uses n-ary versions of the *max*, *min* and *con* operator and transforms them into their binary versions, so that the webservice can handle them. A heuristic was developed for this transformation which leads to much better computation times than any intuitional approach. The idea behind the heuristic is based on the Divide & Conquer mechanism that is used by *APHMIN*. Because the tool applies the minimization algorithm to every operand of the operators, the heuristic tries to reduce the state space in early stages. The speed up that is reached by the heuristic depends on the actual expression that is transformed. The expressions used in the evaluation section were optimized to show large differences in computation. If the order of the operands had been inverted, the heuristic would have delivered the same transformations than the naive approach. This case is uninteresting, because we would achieve the exact same computation times.

A case where the heuristic may not compute the best transformation is whenever the sizes of the operands only differ a little bit. In this case, sorting the operands according to the density of the transitions that occur in the minimized representations might lead to slightly better computation times for other transformations. However, this is hard to predict because we don't have much information in advance about the minimization of the operands. So the heuristic leads to good results with efficient computation times.

Bibliography

[1] Pepijn Crouzen and Frédéric Lang. Smart reduction. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software Engineering*, volume 6603 of *Lecture Notes in Computer Science*, pages 111–126. Springer Berlin Heidelberg, 2011.

[2] Qi-Ming He and Hanqin Zhang. Spectral polynomial algorithms for computing bi-diagonal representations for phase type distributions and matrix-exponential distributions. *Stochastic Models*, 22(2):289–317, 2006.

[3] R. Pulungan. *Reduction of Acyclic Phase-Type Representations.* PhD thesis, Universität des Saarlandes, 2009.

[4] R. Pulungan and H. Hermanns. Acyclic minimality by construction—almost. In *Sixth International Conference on the Quantitative Evaluation of Systems, 2009. QEST '09.*, pages 63–72, 2009.

[5] Reza Pulungan and Holger Hermanns. A construction and minimization service for continuous probability distributions. *International Journal on Software Tools for Technology Transfer*, pages 1–14, 2013.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,________________________ ________________________________

(Datum/Date) (Unterschrift/Signature)

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement in Lieu of an Oath

I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken,………………………………..   ………………………………………….

(Datum / Date)   (Unterschrift / Signature)