FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation

# Quantitative evaluation of network reliability

Dipl.-Math. oec. Univ. Minh Lê

FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Rechnertechnik und Rechnerorganisation

# Quantitative evaluation of network reliability

Dipl.-Math. oec. Univ. Minh Lê

Vollständiger Abdruck der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigten Dissertation.

| | |
|---|---|
| Vorsitzender: | Univ.-Prof. Dr. Martin Bichler |
| Prüfer der Dissertation: | |
| | 1. Univ.-Prof. Dr. Arndt Bode |
| | 2. Univ.-Prof. Dr.-Ing. Markus Siegle, Universität der Bundeswehr München |

Die Dissertation wurde am 10.12.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.03.2014 angenommen.

Ich versichere, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 21. Mai 2014 Dipl.-Math. oec. Univ. Minh Lê

# Acknowledgments

The process of conceiving and rounding out this work would not have been possible without the support of those people to whom I would like to dedicate my gratefulness: First and foremost, I would like to thank my supervisor Prof. Dr. Arndt Bode from the Technische Universität München for his invaluable support and for providing the funding for this work. I am also highly indebted to my supervisor Prof. Dr. Markus Siegle from the Universität der Bundeswehr München for his enthusiastic help and our many fruitful discussions. I am deeply grateful to my advisor Dr. Max Walter from Siemens AG Nürnberg for his detailed, constructive comments and his important guidance throughout this work. I would like to express my gratitude and appreciation to Dr. Josef Weidendorfer who was abundantly helpful and offered invaluable support regarding implementational issues. Furthermore I would also like to thank Dr. Olaf Malassé and Grégory Buchheit from École Nationale Supérieure d'Arts et Métiers Metz for their close cooperation and support. In no particular order, my special thanks go to Alexander Gouberman, Martin Riedl and Dr. Johann Schuster from the Institute of Informatics at the Universität der Bundeswehr München for their excellent cooperation in the framework of the LARES project and, beyond this, for the many valuable discussions and feedback. During this period I have collaborated with many colleagues for whom I have great regard, and I wish to express my warmest thanks to all those who have helped me with my work in the Department of Informatics 10 of the Technische Universität München. I finally would like to express my gratitude towards my parents for their constant support and encouragement which has helped me in completion of this project.

# Abstract

The determination of the reliability value for technical systems whose components are subjected to random failure possesses a wide range of applicability, *e.g.* in data communication networks, computer architectures, electrical power networks or fault-tolerant systems in general. The reliability of the respective system is computed by means of a stochastic network which models the system's inherent redundancy structure. This task is known to be NP-hard even if independent component failures are assumed. Hence, efforts to conceive efficient solutions on restricted classes of networks have been pursued since the 1960s, leading to many different approaches and techniques. In this thesis, the state of the art in the field of exact reliability computation is reviewed and improved.

Substantial extensions or modifications are given to significantly improve the current fastest approaches regarding runtime and the current minimal requirements for memory consumption.

Due to the exponential nature of the problem, one must account for a prohibitive amount of computation time for large-scale networks. For this purpose, an efficient exact algorithm which gives fast converging bounds is investigated and extended for coping with networks of larger size (consisting of at least 100 nodes). In case the computation cannot be completed, one at least obtains lower bounds for the reliability.

With regard to modern applications it is not sufficient to assume independent component failures because in fault tolerant systems components may fail due to common cause failures or fault propagation. In order to account for dependent component failures, the combinatorial model is extended for the dependent case and the treated efficient combinatorial methods are given suitable modifications.

# Contents

**Acknowledgements** **vii**

**Abstract** **ix**

**I. Introduction** **1**

**1. Introduction** **3**
- 1.1. Motivation . . . 3
- 1.2. Contribution of this work . . . 4
- 1.3. Structure of the thesis . . . 5

**2. Theoretical Background** **9**
- 2.1. Boolean Expressions . . . 9
- 2.2. The System Model . . . 10
  - 2.2.1. Monotonous system . . . 10
  - 2.2.2. Probabilistic Graph Model . . . 11
  - 2.2.3. k-terminal reliability, node failures and directed edges . . . 12
  - 2.2.4. k-out-of-n, series and parallel systems . . . 13
- 2.3. Dependability . . . 13
  - 2.3.1. Reliability . . . 14
  - 2.3.2. Availability . . . 14
- 2.4. The Complexity of Network Reliability Computation . . . 15
  - 2.4.1. Complexity classes P, NP and #P . . . 15
  - 2.4.2. Reliability Polynomial . . . 16
- 2.5. Binary Decision Diagrams . . . 18
  - 2.5.1. Operations . . . 19

**II. State of the Art** **23**

**3. Techniques for exact k-terminal reliability computation** **25**
- 3.1. Path and Cut Enumeration . . . 25
  - 3.1.1. Enumerating minimal paths . . . 25
  - 3.1.2. Enumerating minimal cuts . . . 26
  - 3.1.3. Inclusion-exclusion method . . . 26
  - 3.1.4. Sum of disjoint products . . . 26
  - 3.1.5. Summary of this section . . . 27
- 3.2. Reduction techniques . . . 27

3.2.1. Series-parallel and degree-2 reductions . . . . . 28
3.2.2. Polygon-to-chain reductions . . . . . 28
3.2.3. Removal of irrelevant components . . . . . 28
3.3. Factoring with reductions . . . . . 29
3.3.1. Choice of the keystone edge . . . . . 29
3.3.2. Incorporating triconnected decomposition in a factoring with reductions algorithm . . . . . 30
3.3.3. Considering node failures . . . . . 31
3.3.4. Bottom line . . . . . 31
3.4. Kuo-Lu-Yeh approach . . . . . 31
3.4.1. The approach . . . . . 32
3.4.2. Case study . . . . . 33
3.4.3. Complexity discussion . . . . . 35
3.4.4. Extension for k-terminal case . . . . . 35
3.4.5. Accounting for node failures . . . . . 36
3.5. Decomposition method . . . . . 36
3.5.1. Boundary set and partitions . . . . . 37
3.5.2. Number of partitions . . . . . 37
3.5.3. Application example . . . . . 38
3.5.4. Details of the decomposition approach . . . . . 40
3.5.5. Runtime complexity . . . . . 40
3.5.6. Sorting of blocks . . . . . 42
3.5.7. Considering node failures . . . . . 42

**4. Bounds on Network Reliability** **43**
4.1. Bounds for networks with equal edge failure probabilities . . . . . 43
4.1.1. Simple bounds . . . . . 44
4.1.2. Ball-Provan bounds . . . . . 44
4.1.3. Lomonosov-Polesskii bounds . . . . . 45
4.2. Bounds for networks with arbitrary edge failure probabilities . . . . . 45
4.2.1. Edge-packing . . . . . 45
4.3. The Dotson-Gobien algorithm . . . . . 46
4.3.1. Derivation . . . . . 47
4.3.2. The approach . . . . . 48
4.3.3. Example . . . . . 49
4.3.4. Required data structures . . . . . 49
4.3.5. Considering node failures . . . . . 50

**III. Improving Bounding, KLY and Decomposition Algorithm** **51**

**5. Improving the Kuo-Lu-Yeh Approach** **53**
5.1. Recognition and deletion of redundant biconnected components . . . . . 54
5.1.1. Transformation to the biconnected component tree . . . . . 54
5.1.2. A case study . . . . . 55

**6. Improving the Decomposition Approach** **59**
6.1. Derivation of new partitions . . . 60
6.1.1. False-leaf when contracting . . . 61
6.2. A new heuristic for finding a good variable order . . . 62
6.2.1. Complexity discussion . . . 63
6.2.2. Application on Example Graphs . . . 63

**7. Improving the Dotson-Gobien Algorithm** **67**
7.1. Obtaining better bounds with the $\Delta$tree . . . 67
7.1.1. Memory-efficient implementation . . . 68

**IV. Measuring and Comparing** **71**

**8. Evaluation on different network structures** **73**
8.1. Results after applying modifications . . . 75
8.1.1. Results for the KLY approach . . . 75
8.1.2. Results for the decomposition approach . . . 79
8.1.3. Results for the modified Dotson-Gobien approach . . . 82
8.2. Comparing State-Of-The-Art . . . 84
8.2.1. Application of the new heuristic to KLY . . . 86

**V. Dependent Component Failures** **87**

**9. Considering dependent failures** **89**
9.1. Preliminaries . . . 89
9.1.1. The Gaussian Copula . . . 90
9.2. Extension for the DG with series-parallel reductions approach . . . 91
9.2.1. Case study . . . 92
9.2.2. Complexity considerations . . . 93
9.3. Extension for BDD-based approaches . . . 93
9.3.1. Case study . . . 95
9.3.2. Complexity discussion . . . 95

**VI. Conclusion** **99**

**10. Summary** **101**
10.1. Extensions and modifications . . . 102
10.2. Choosing the right algorithm . . . 104
10.3. Future research . . . 104

**Bibliography** **107**

# List of Figures

1.1. Gas supply network exposed to an earthquake. . . . . . . . . . . . . . . . . . . . 5

2.1. Common redundancy structures. . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.2. Reducedness conditions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.3. Example of *Apply*, variable ordering: $x_1 < x_2 < x_3 < x_4$ . . . . . . . . . . . . 21

3.1. Factoring with series-parallel reductions. . . . . . . . . . . . . . . . . . . . . . . 30
3.2. Triconnected decomposition and chain replacements. . . . . . . . . . . . . . . 30
3.3. Left: Edge Expansion Diagram, Right: Resulting ROBDD. . . . . . . . . . . . 35
3.4. Left: Articulation vertex, Right: Frontier set $F_k$. . . . . . . . . . . . . . . . . . . 37
3.5. OBDD for G, variable order: $e0 < e3 < e4 < e1 < e5 < e6 < e2$ . . . . . . . . 39

5.1. Deletion of redundant biconnected components. . . . . . . . . . . . . . . . . . . 55
5.2. Edge Expansion Diagram for Nw.1. . . . . . . . . . . . . . . . . . . . . . . . . . 57

6.1. Example for false-leaf when contracting edge. . . . . . . . . . . . . . . . . . . . 62
6.2. Fan network . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.3. N-node complete graph (two-level bfs recursion) . . . . . . . . . . . . . . . . . . 64
6.4. Frontier size in each BDD level for the complete 10-node network . . . . . . 65
6.5. Frontier size in each BDD level for the ARPANET 1979 . . . . . . . . . . . . . 66

8.1. Benchmark Networks 1-27. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
8.2. Samples for the randomized networks 28-31 . . . . . . . . . . . . . . . . . . . . 75
8.3. Hit ratio and ratio of hashmap sizes . . . . . . . . . . . . . . . . . . . . . . . . . 76

9.1. Initial graph. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
9.2. Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
9.3. Recursion Tree. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
9.4. Referring to the example from Figure 9.1, ROBDD (Left), ROMDD (Right). . 96

10.1. Graph model for the gas supply network under seismic hazard. . . . . . . . 101

# List of Tables

2.1. Truth table for $\neg x_1$ . . . . . 9
2.2. Truth table for $x_1 \wedge x_2$, $x_1 \vee x_2$ . . . . . 10
2.3. Time complexities for BDD manipulation operations . . . . . 20

3.1. # Partitions according to $|F_k|$ and # blocks . . . . . 38
3.2. Total # Partitions for two- and all-terminal case . . . . . 38

6.1. Case-by-case analysis for deduction of new partitions . . . . . 61
6.2. Contract I. . . . . . 61
6.3. Contract II. . . . . . 61
6.4. Delete I. . . . . . 61
6.5. Delete II. . . . . . 61

8.1. Reliability values and #BDD-nodes . . . . . 76
8.2. Comparison of results (original KLY vs. improved KLY) . . . . . 77
8.3. Comparison of results (decomposition method using bfs vs. new heuristic) 80
8.4. Memory consumption (KLY and decomposition approach) . . . . . 81
8.5. Bounds (relative accuracy of $0.1$) . . . . . 84
8.6. Exact results (DG method) . . . . . 85
8.7. Perfomance ranking ($KLY_{1/2}$ vs. $D_{1/2}$ vs. modified DG) . . . . . 86

10.1. KLY, decomposition and DG method compatibility towards modelling aspects for the independent case. . . . . . 102
10.2. KLY, decomposition and DG method compatibility towards modelling aspects for the dependent case . . . . . 104

# List of Notations

| | | |
|---|---|---|
| BDD | Binary Decision Diagram | 12 |
| bfs | breadth-first-search | 31 |
| CNF | Conjunctive Normal Form | 6 |
| dfs | depth-first-search | 31 |
| DG | Dotson-Gobien | 41 |
| DNF | Disjunctive Normal Form | 5 |
| EED | Edge Expansion Diagram | 26 |
| KLY | Kuo-Lu-Yeh | 25 |
| MTTF | mean time to failure | 9 |
| Nw. | Network | 51 |
| OBDD | Ordered Binary Decision Diagram | 13 |
| ROBDD | Reduced Ordered Binary Decision Diagram | 13 |
| ROMDD | Reduced Ordered Multiple Decision Diagram | 88 |
| SDP | sum of disjoint products | 20 |
| SIC | set of interdependent components | 83 |

# Part I.

# Introduction

# 1. Introduction

Network reliability analysis is generally understood to be the determination of the probability that a system, which consists of error-prone components, can perform its intended function at a fixed instant in time. There are many real-world settings where network reliability analysis finds its application: one of the earliest usages can be traced back to the reliability assessment of relay circuits with unreliable contacts [52]. In this case, both contacts and relays can be regarded as components which may fail. In order to ensure current flow, certain contacts must be closed so that the circuit (network) is closed. Reliability concerns are also directed towards communication networks. One is interested in the probability that a sent message arrives at one designated, several intended or all network participants under the assumption of communication link failure.
Colbourn *et al.* employed network reliability modelling techniques to analyse building house damages resulting from fire spread [16]: under the assumption that a building comprises a particular number of critical components and that the fire is considered to spread instantaneously, the probability that all critical components are rendered inoperative is sought.
Network reliability analysis models also offer a remedy for assessing highly reliable *fault-tolerant systems*[1], since their testing is technically not feasible due to very rare occurences of components failures. Further on, those reliability models can be used to conduct risk analysis on lifeline networks [68]: those types of networks are transmission systems for utility services such as water, sewage, gas or power networks. Obviously, one is concerned with the probability of the guaranteed supply of goods from distributor to recipients.
From the perspective of a system architect, network reliability analysis serves as a tool to figure out weak spots in reliable systems, to answer the question of whether a system meets reliability acceptance criteria or just to see the effect of a change in probability of failure of components. Based on these answers, the system architect may replace certain parts of the system with more reliable parts in order to increase the overall system reliability. However, we note that network reliability analysis does not directly provide the answer to the modeler or system architect as to how to change system design for increasing the system's overall reliability but rather a quantitative value from which a statement can be made about the reliability of the designed system.

## 1.1. Motivation

At this point we want to further illustrate the subject and point out different requirements for network reliability modelling by means of an example. Figure 1.1 can be regarded as a map which shows a gas supply network under earthquake hazard. The network par-

[1]A fault-tolerant system is a system that continues operating properly in the event of the failure of some components. Hence, a fault-tolerant system tolerates a certain combination of component failures.

ticipants are one gas supplier S and four gas distributors D1-D4, each of them located at different geographic points. The participants are physically connected by gas pipelines of different length and diameter. To guarantee a sufficient gas supply between S and D4, at least two out of three gas pipelines P1, P2 and P3 must be intact (2 out of 3 are good). The reason therefore is that the pipelines P1, P2 and P3 have a smaller diameter compared to the others. The gas network is further exposed to an earthquake whose seismic intensity is strong enough to severely impair pipelines P5, P6, P7. Under these circumstances, a distributor would be concerned about an assured supply of gas. Otherwise, a supplier would like to know the chance that the gas arrives at all predetermined distributors. This example contains several important aspects that should be considered in a reliability evaluation model. In general, a good model should take into account all relevant aspects to allow an accurate assessment and at the same time be kept as simple as possible. Without the occurrence of the earthquake, the current state-of-the-art Boolean models for network reliability consider the pipelines as system components that fail independently with a certain probability. In addition, the participants are assumed to be perfectly reliable. The gas flow direction can also be taken into account: for this example, we assume that the pipelines allow a bidirectional flow. Following this, the concern of a distributor corresponds to the well-known *terminal-pair reliability* problem and the described interest of the supplier to the *k-terminal reliability* problem. Despite the assumption of independent component failures and perfectly reliable interconnection points (participants), we will see that these problems are already NP-hard. Hence, the amount of computation time increases exponentially with the size of the input. This had led to research attempts at finding polynomial time algorithms for restricted classes of networks: the k-terminal problem for series-parallel [64] and bounded treewidth networks [5] can be solved in time linear to the size of the network. When all system component fail with the same probability, the *all-terminal problem* on complete networks is solvable in polynomial time [30].

Even though we cannot expect to find polynomial time algorithms for arbitrary networks, it is worth spending the effort in improving the current most efficient exponential time algorithms. The hope is that the runtime of the improved algorithm, which has still exponential complexity, does not grow as fast as the original algorithm with the size of the input: for instance, if we could find a way to reduce a complexity of $2^{2n}$ to $2^n$, $n$ equals the size of an input, we would be able to compute an input of double size in the same amount of time.

The example in Figure 1.1 further shows that in the event of the illustrated earthquake, the failures of pipelines P5, P6 and P7 are spatially correlated. Assuming independent failures are therefore not realistic and may lead to overoptimistic results. This prompts us to extend the current evaluation techniques for taking into account dependent component failure events. The main contributions of this thesis are summarized in the following.

## 1.2. Contribution of this work

In this thesis, we first give a chronological overview of the most important exact approaches developed for solving the terminal reliability problems. The two state-of-the-art methods, namely the Ku-Luo-Yeh method and the decomposition method, which proved to be the currently most efficient methods for general networks, are significantly improved

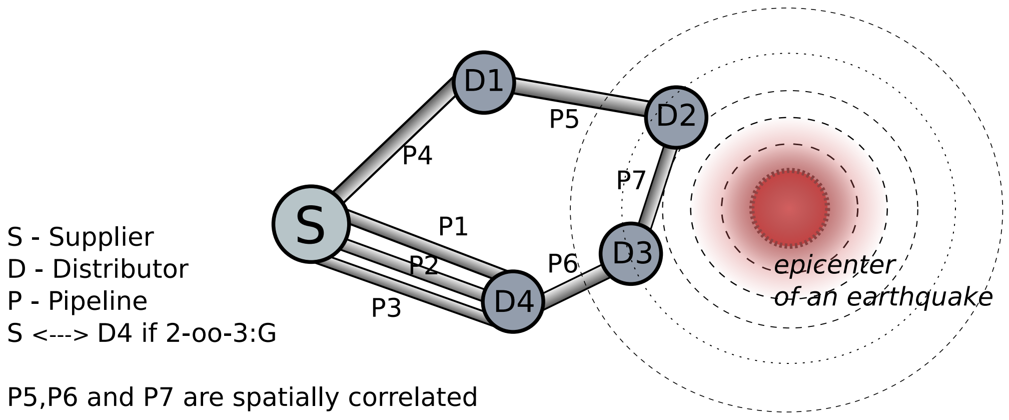


Figure 1.1.: Gas supply network exposed to an earthquake.

with regard to runtime and the current memory requirements. Since the mentioned problem is of exponential nature, bounding approaches are indispensable alternatives towards exact approaches. Therefore, we extend an efficient exact approach (Dotson-Gobien approach) which gives fast converging bounds, for coping with network sizes of higher scale. To demonstrate the achieved improvements and identify the most suitable approach, the implementations of all three approaches are compared to their state-of-the-art and against each other in a large-scale measurement series comprising regular networks and a set of randomized networks of different sizes. Finally, all three approaches are given an appropriate extension for considering dependent failure events.
Before presenting the structure of this thesis, we refer to Chapter 10 where the respective abstraction (graph model) for the gas supply network is established.

## 1.3. Structure of the thesis

This thesis is divided into six parts with the following contents:

### Part I: Introduction

#### CHAPTER 1: MOTIVATION

The first chapter lists several real-world applications and different types of networks where fault-tolerance mechanisms may apply. It points to the necessity of dependent failure consideration by means of an example: a gas supply network exposed to the risk of an earthquake.

#### CHAPTER 2: FOUNDATION

The second chapter presents the use of the probabilistic graph model for describing the redundancy structure and independent component failures of any fault-tolerant system. After introducing the relevant measures of network reliability (2-terminal, k-terminal and all-terminal reliability), the complexity of their computation is discussed. Thereafter, an important data structure which has left a large impact in the field of Network Reliability Calculus is presented: the so-called BDD (Binary Decision Diagram).

## Part II: State of the Art

Chapter 3: Techniques for exact two-terminal reliability computation

This chapter presents the reliability-preserving graph transformations and reductions. Depending on the available graph structure, these operations may lead to a drastic simplification of the network model and should therefore be incorporated into exact approaches whenever possible. The advantages and disadvantages of different classes of efficient exact approaches are discussed subsequently.

Chapter 4: Bounds on Network Reliability

Since determining network reliability is an NP-complete task, we discuss numerous efforts to find approximate solutions. They are briefly described for models with equal and distinguished edge failure probability.

## Part III: Improving KLY, Decomposition and Bounding Algorithm

Chapter 6: Improving the Kuo-Lu-Yeh Approach

Implementational details are given for the representation of the combinatorial network graph. The unambiguity of reliability isomorphic graphs is then inferred from the representation. A contribution is made to speed up the Kuo-Lu-Yeh algorithm: by recognizing redundant biconnected components, the subgraph sizes can be drastically reduced leading to lower memory consumption and saving of redundant computations.

Chapter 6: Improving the Decomposition Approach

Another state-of-the art algorithm which has a completely different ansatz, but which also applies BDD, is presented. For certain network structures this algorithm has polynomial runtime restricted to the network's treewidth. We describe a new heuristic for finding a good variable ordering and empirically show that this ordering is to be preferred to the current breadth-first-search variable ordering. As a consequence the algorithm is significantly faster and consumes at the same time less memory for many sample networks. Furthermore, implementational details for efficiently representing upcoming partitions are worked out.

Chapter 7: Improving the Dotson-Gobien Algorithm

An efficient approach by Dotson and Gobien is improved by introducing an own data structure called delta-tree. Additionally, the memory is migrated to low-bandwidth high-capacity storage.

## Part IV: Measuring and Comparing

Chapter 8: Regular and randomized network structures

Our theoretical modifications are put into practice for a wide variety of network structures frequently used as benchmarks in the literature. To cover up the structure diversity we test the improved algorithms on randomized networks. A pre-post comparison shows the effects of the modifications.

## Part V: Dependent Component Failures

CHAPTER 9: CONSIDERING DEPENDENT FAILURES

In this chapter we show how to appropriately extend the three improved approaches to the case of dependent component failures. As a result, the combinatorial model acts jointly with a stochastic model in a hybrid context.

## Part VI: Conclusion

CHAPTER 10: SUMMARY AND OUTLOOK

Finally, we point out the impacts of the contributions we have made for the field of reliability evaluation. For the current state-of-the-art, we conclude the choice of appropriate approaches for certain input networks.
An outlook is given to open and substantial issues which can be further explored for improving and extending the state-of-the-art .

# 2. Theoretical Background

This chapter provides the essential mathematical concepts for the assessment of network reliability. The underlying system model that is assumed throughout this work is specified after the introduction of Boolean expressions. This particular model represents a common abstraction that is widely used in the literature. We further discuss some variations of the stipulated model which allow to consider important issues in network reliability such as failure of routers and directed communication link failures. Subsequently, we give an overview over the different reliability measures along with the complexity of the underlying combinatorial problems. Finally, we explain the meaningful BDD (Binary Decision Diagram) data structure which has become an integral part of modern reliability algorithms.

## 2.1. Boolean Expressions

Boolean expressions are composed of Boolean variables $x_1, x_2, \dots, x_n$ $(n \in \mathbb{N})$, constants *true* $\equiv 1$ or $false \equiv 0$ which again can be combined by binary operators such as *conjunction* $\wedge$, *disjunction* $\vee$ or the unary operator *negation* $\neg$. The priorities are, with the highest first: $\neg$, $\wedge$, $\vee$. The Boolean variable can be assigned to either $0$ or $1$. Two Boolean expressions $b$ and $b'$ are equal if they yield the same constant value ($0$ or $1$) for each assignment of the variables. If the constant values are regarded as integers, the following calculation rules hold:

$$\begin{aligned} x_1 \wedge x_2 &= x_1 x_2 = \min(x_1, x_2), \\ x_1 \vee x_2 &= x_1 + x_2 - x_1 x_2 = \max(x_1, x_2), \\ \neg x_1 &= 1 - x_1. \end{aligned}$$

Table 2.2 and 2.1 shows the respective outcomes for each assignment of $x_1$ and $x_2$ with respect to the basic operators.

Table 2.1.: Truth table for $\neg x_1$

| $x_1$ | $\neg x_1$ |
|---|---|
| 1 | 0 |
| 0 | 1 |

For Boolean variables $a, b, c$, we list some basic calculation rules:

- identity: $a \vee 0 = a, \quad a \wedge 1 = a,$
- distributivity: $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c), \quad a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c),$
- complement: $a \vee \neg a = 1, \quad a \wedge \neg a = 0,$

Table 2.2.: Truth table for $x_1 \wedge x_2$, $x_1 \vee x_2$

| $x_1$ | $x_2$ | $x_1 \wedge x_2$ | $x_1 \vee x_2$ |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

- idempotence: $a \wedge a = a, \quad a \vee a = a$,
- De Morgan's law: $\neg(a \wedge b) = \neg a \vee \neg b, \quad \neg(a \vee b) = \neg a \wedge \neg b$.

A Boolean expression is in DNF (Disjunctive Normal Form) if it consists of a disjunction of conjunctions of variables or negations of variables, *i.e.*, if it is of the form

$$\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_j} x_i^j = \max_{i=1,\dots,m} \left( \min_{j=1,\dots,n_j} \left( x_i^j \right) \right). \tag{2.1}$$

Similarly, a CNF (Conjunctive Normal Form) is an expression that is represented as

$$\bigwedge_{i=1}^{m} \bigvee_{j=1}^{n_j} x_i^j = \min_{i=1,\dots,m} \left( \max_{j=1,\dots,n_j} \left( x_i^j \right) \right). \tag{2.2}$$

Each $x_i^j$ is either a variable or a negated variable. It follows that any Boolean expression is equal to an expression in CNF and an expression in DNF. According to [3], the conversion between CNF and DNF has exponential complexity.

## 2.2. The System Model

The system to be evaluated is a time-independent (static), binary-state system which consists of binary-state components. The components are either failed or working. Once they are failed, they remain in failed state since we assume that there is no repair. Depending on the particular states of the components, the system itself is either up or down. Our goal is to determine the overall probability of the successful operation of a system. Therefore, we first state the abstraction for the system at hand. Afterwards the particular system model is specified.

### 2.2.1. Monotonous system

After Kohlas [35] the structure of a binary-state system can be defined as follows:

**Definition 2.1.** *Let $B$ be a finite set and $S$ a collection (or family) of subsets of $B$ with the following properties:*

1. *$B \in S$ and $\emptyset \notin S$.*
2. *If $A \in S$ and $A \subset A'$, then also $A' \in S$,*

*then $(B, S)$ is called a monotonous system.*

For this general case, the subsets in $S$ are regarded as *connections*. Assuming that the elements of $B$ may fail, we are particularly interested in those cases where the functioning elements still form a connected structure. The monotonicity property (2) then assures that a functioning system cannot turn into a failed state due to a repair of one of its elements. Oppositely, a failed system will not turn into a functioning state if another component fails.

### 2.2.2. Probabilistic Graph Model

At this point the abstract model is specified with regard to the terminal-pair problem which is used throughout this work. It is called the probabilistic graph model. We refer to [62] for similar representations, such as RBD (reliability block diagrams) and fault-trees, that are frequently used in the field of reliability.
The redundancy structure of a system to be evaluated is modelled by an undirected connected graph[1] $G := (V, E)$ with no loops (a node is connected with itself by an edge), where $V$ stands for a set of vertices or nodes and $E \subset V \times V$ a set of unordered pairs of vertices, called edges with $|V| = n$, $|E| = m$. We define two not necessarily injective maps: $f : E \to C$ assigns the edges to the set of system components $C$, with $|C| = q$, and $g : E \to V \times V$ assigns each edge to a pair of nodes. This definition allows the multiple occurrence of a system component. So there may be several edges that are mapped to the same component. Defining

$$\mathcal{E}_i := \{e \in E : f(e) = e_i\}, \;\; 1 \leq i \leq q$$

to be the set of edges that are mapped to the same component $e_i \in C$. Then all edges in $\mathcal{E}_i$ have each the same edge label $e_i$. If $f$ was injective, each edge $e \in E$ would stand for a distinct system component. Thus, $|\mathcal{E}_i| = 1, \forall i$ and $q = m$. For instance, $f$ is injective for the series or parallel structure and surjective for the 2-out-of-3 structure in Figure 2.1. Since each component is subject to failure, each edge $e$ can be in two states: either failed or working. The probability of failure $q_e := 1 - p_e$ is given for each $e \in E$, where $p_e$ is the probability for edge $e$ working. We assume that the components fail independently. Furthermore, all nodes are considered to be perfect. In $G$ we specify two nodes $s$ and $t$ that are called *terminal nodes*. The system's terminal-pair reliability, $R_{s,t}(G)$, is the probability that the two specified terminal nodes are connected by at least one path[2] consisting of only edges associated with working components. Such a path $P$ describes one possible component setting which leads to system operation. $P$ is a set of working components which forms a subgraph $G' = (V, E')$ of $G$, where $E' := \{e \in E : f(e) \in P\}$.
The system state is therefore determined by the particular states of the components and is represented by a *binary-state vector* $\vec{x} \in \{0, 1\}^q$. The i-th content $x_i$ of $\vec{x}$ is a Boolean variable which keeps track of the state of component $e_i$, whereas $x_i = 1/0$ for $e_i$ working/failed. The state vector $\vec{x}$ is also called *elementary event* of the binary system. The system's *structure*

[1]A graph is undirected if its edges have no orientation. The edge $(u, v)$ is identical to the edge $(v, u)$, *i.e.*, they are not ordered pairs. An undirected graph is said to be connected if every pair of vertices in the graph is connected by at least one path. Note that the monotonous property implies a connected graph as valid input.

[2]An $s, t$-path in $G$ is any sequence $s = v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k = t$ of nodes $v_i \in V$ and edges $e_j = (v_{j-1}, v_j) \in E$, $j = 1, \ldots, k$.

*function* with respect to $G$:

$$X_G : \vec{x} \in \{0,1\}^q \to \{0,1\} \tag{2.3}$$

describes a binary function mapping a binary-state vector $\vec{x}$ to value $0/1$ when the system is in failed/working state. $X_G$ also stands for the set of all possible elementary events with cardinality $2^q$. Hence, $X_G$ represents a Boolean function with the following properties:

- $X_G$ is monotonous, since $X_G(a_1, a_2, \dots, a_q) \leq X_G(b_1, b_2, \dots, b_q)$ if $a_i \leq b_i, \forall i = 1, 2, \dots, q$ with $a_i, b_i \in \{0,1\}$.
- $X_G(1, 1, \dots, 1) = 1$ and $X_G(0, 0, \dots, 0) = 0$.

To relate the Boolean function $X_G$ to the definition of a monotonous system, we set $S$ to be the set of all subgraphs of $G$ where $s$ and $t$ are connected by working edges, meaning that the system is working. Note that the subgraph of $G$ only consists of working edges (edges that are mapped to working components). Denote $B$ to be the finite set where all components $c \in C$ are working. Let $A \subset B$ be a set of working components which corresponds to a subgraph of $G$ where s and t are connected. $A$ also stands for an $s$-$t$-path. Since property (2) of Definition 2.1 is fulfilled, $(C, S)$ is a monotonous system. Furthermore, it holds that $X_G(\vec{x}^A) = 1$, where $x_i^A = 1$ if $e_i \in A$ and $x_i^A = 0$ for $e_i \notin A$. Because of the monotonous property of the Boolean function $X_G$, we can follow that also $(X_G, S)$ is a monotonous system.
Since the failures are independent, the probability for an elementary event $\vec{x}$ is defined as

$$\mathbb{P}(\vec{x}) = \prod_{i=1}^{q} ((1 - x_i) \cdot q_i + x_i \cdot p_i). \tag{2.4}$$

Ultimately, the terminal-pair reliability, $R_{s,t}$, is expressed in terms of the sum of probabilities of elementary events implying system operation:

$$R_{s,t}(G) := \sum_{X_G(\vec{x})=1} \mathbb{P}(\vec{x}). \tag{2.5}$$

A naive approach to compute the reliability polynomial would be to enumerate all possible subgraphs and sum up the probabilities of those where the terminal nodes are connected. Unfortunately, this method is of exponential complexity, since there are $2^q$ possibilities. Therefore, more efficient techniques are presented in Chapter 3.

### 2.2.3. k-terminal reliability, node failures and directed edges

By restricting the model to perfect nodes, undirected edges and two terminals, we do not want to withhold important reliability aspects such as failures of routers, directed communication link failures or communication of several stations: these aspects can be extended in a simple way for some of the approaches presented in the following chapters. The introduced model forms the intersection of the models postulated in many effective terminal-pair approaches. Thus, it offers primarily a good basis for comparison.
In general, the cardinality of the set of terminal nodes $K$ can range between $2 \leq |K| \leq |V|$. As we will see in some of the exact approaches treated in Chapter 3, the solution for the

all-terminal ($|K| = |V|$) and the k-terminal ($2 \leq |K| \leq |V|$) reliability can either be directly inferred from the solution of the two terminal problem ($|K| = 2$) or their determination only require some slight modification of the terminal-pair approach.
The undirected graph model is merely a special case of the more general directed graph model, since each undirected edge can be replaced by two oppositely directing edges. To consider node failures, Ball [7] suggested to transform the network with unreliable nodes to a network with perfectly reliable nodes and perform the computation on the transformed network.

We do not follow this idea, since this transformation unfortunately highly increases the number of unreliable edges which in turn increases the magnitude of the problem. Instead, node failures can be considered with just a linear overhead after having solved the model with perfect nodes [36, 73, 74, 26].

### 2.2.4. k-out-of-n, series and parallel systems

In practice, *k-out-of-n* redundancy structures are very common. For a system consisting of $n$ components, at least $k$ components must be working for a successful system operation. By allowing multiple edges assigned to the same component, the redundancy structure of such a system can be considered by our model. On the left of Figure 2.1, the appropriate graph is illustrated for a 2-out-of-3 system. The terminal nodes $s$ and $t$ are marked in black. In general, we have $\binom{n}{k}$ minimal $s$-$t$-connections (or paths) of length $k$. They are minimal, since $s$ and $t$ are disconnected in any of their subsets.
For a *series* system (in the centre), there is exactly one minimal $s$-$t$-connection consisting of all edges. And for a *parallel* system (on the right), each edge is itself a minimal $s$-$t$-connection. Unlike the *k-out-of-n* systems, each of the edges in a series or parallel system represents a different component.

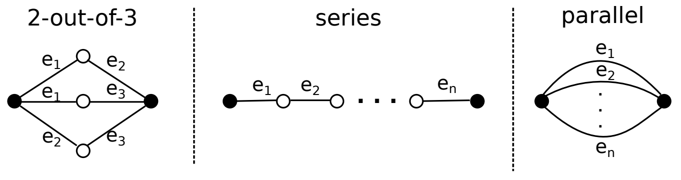


Figure 2.1.: Common redundancy structures.

## 2.3. Dependability

In our model we assess the dependability of non-repairable systems for a fixed point of time $t$. The dependability of a system is a probability measure which is understood as both system reliability and system availability. Only $t$ needs to be fixed in the probability distribution function of these two probability measures in order to obtain the probability of failure for each system component. These two frequently used probability measures are defined in the following subsections.

### 2.3.1. Reliability

Consider a set of several homogenous components which are put into service for a long time period. By observing their failure frequency over time, we would be able to statistically infer the component's lifetime that is defined as the time period $[0, T]$. Starting with time zero when the component is put into service, under the assumption that it correctly functions. The component fails at time $T$, whereas $T$ is a random variable with distribution function $F(t) = \mathbb{P}(T \leq t)$. Let $X_t,\ t \geq 0$ be a function that records the component's state at each point of time. $X_t$ is a stochastic process with $X_t = 1$ if the component is working and $X_t = 0$ if the component is failed at time $t$. By definition $X_t = 1,\ 0 \leq t < T$.
The reliability $R(t)$ of a component equals its probability of surviving which is at the same time the complementary probability of $F(t)$:

$$R(t) = 1 - \mathbb{P}(T \leq t) = \mathbb{P}(T > t) = \mathbb{P}(X_t = 1). \tag{2.6}$$

Assume that $f(t)$ is the density function of lifetime $T$ with $f(t) = \frac{dF(t)}{dt}$, then the MTTF (mean time to failure) is defined as the expected value of lifetime $T$:

$$MTTF := E(T) = \int_0^\infty x f(x)dx = \int_0^\infty R(x)dx. \tag{2.7}$$

The second equality of Equation 2.7 follows from partial integration. Another characteristic measure used to describe the components' vulnerability to failure after a time $t$, is the failure rate $r(t)$:

$$r(t) = \lim_{h \to +0} \frac{F(t+h) - F(t)}{h \cdot R(t)} = -\frac{\frac{d}{dt}R(t)}{R(t)} = -\frac{d}{dt}\left(\ln R(t)\right). \tag{2.8}$$

Denote $\mathbb{P}(T > t + h | T > t) = \frac{\mathbb{P}(T > t+h)}{\mathbb{P}(T > t)} = \frac{R(t+h)}{R(t)}$ to be the component's conditional surviving probability. The first term of the Equation 2.8 is then derived from the complementary probability $\mathbb{P}(T \leq t + h | T > t) = \frac{\mathbb{P}(t < T \leq t+h)}{\mathbb{P}(T > t)} = \frac{F(t+h) - F(t)}{R(t)}$ - meaning that a component fails within time $h$ after having survived $t$ amount of time. By integrating Equation 2.8 and considering the boundary condition $\ln R(0) = \ln 1 = 0$, the reliability $R(t)$ can also be expressed as:

$$R(t) = \exp\left(-\int_0^t r(x)dx\right). \tag{2.9}$$

For reliability modelling, it is very common to assume the negative exponential distribution function $F(t) = 1 - \exp(-\lambda t)$ with failure rate $\lambda > 0$. In doing so, $R(t) = \exp(-\lambda t)$ and $MTTF = \frac{1}{\lambda}$.

### 2.3.2. Availability

While the reliability, $R(t)$, is a measure for an interval of time, the availability, $A(t)$, is a measure for a point of time: $A(t)$ is the probability that a component works correctly at time $t$ and by definition, $R(0) = A(0) = 1$. The reliability and availability of components are equal for all $t \geq 0$ if no component repair is allowed. However, we omit further details for the availability involving component repair, since the above definitions suffice our purpose of determining the dependability of a non-repairable system: We only need to fix a point of time $t$ and set probabilities $p_i = R_i(t)$ or $p_i = A_i(t)$ for each component $i$ of the system. Nevertheless, we remark that for the case where components can be repaired, the component states alternate between the failed and available state while operating. In contrast to $R(t)$, $A(t)$ does not tend to zero for $t \to \infty$, but to a positive value less than or equal to one (see [69]) which is the component's steady state availability.

## 2.4. The Complexity of Network Reliability Computation

For the purpose of a better understanding, we first present some fundamental definitions from complexity theory[3] before discussing the complexity of network reliability measures.

### 2.4.1. Complexity classes P, NP and #P

In general, we are faced with the task of computing a function whose inputs and outputs are typically restricted to finite *strings of bits*. For any integer $n \in \mathbb{N}$, the set of length-$n$ *strings of bits* is denoted as $\{0,1\}^n$. We then define $\{0,1\}^* := \bigcup_{k\geq 0} \{0,1\}^k$ to be the set of all finite *strings of bits*. For a function $f : \{0,1\}^* \to \{0,1\}$ and $L_f := \{x : f(x) = 1\}$, the computational problem of deciding whether $x \in L_f$ is called *decision problem*. $L_f$ is also called *language*.

**Definition 2.2.** *A language $L \subset \{0,1\}^*$ is in complexity class P if and only if there exists a deterministic Turing machine DTM, such that DTM runs for polynomial time on all inputs $x$ and decides on $L_f$:*

$$\forall x \in L_f \text{ DTM outputs 1}, \forall x \notin L_f \text{ DTM outputs 0.}$$

In principle, it can be said that P corresponds to a class of problems that can be efficiently solved on a computer (polynomially decidable).

A language $L \subseteq \{0,1\}^*$ can also be defined by a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, such that $L_R = \{x : \exists y,\ (x,y) \in R\}$. For an instance $x$, we search for some $y$ such that the relation $R$ is fulfilled, meaning that $(x,y) \in R$. We call this *search problem*. For the following class of problems, a provided solution $y$ to a search problem can be efficiently verified (polynomially verifiable), but it may take exponential time to compute a solution.

**Definition 2.3.** *$L_R$ is in complexity class NP (which is equivalent to that $R$ is an NP-relation), if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time deterministic Turing machine $DTM$ (called the verifier for $L_R$) such that $\forall x \in \{0,1\}^*$,*

$$x \in L_R \Leftrightarrow \exists \left(y \in \{0,1\}^{p(|x|)} \text{ so that } DTM(x,y) = 1.\right)$$

In this case, $y$ is called *certificate* for $x$ with respect to the language $L_R$. Furthermore, $|x|$ denotes the length of an instance $x$ and the length of $y$ is polynomially bounded by the length of $x$. It holds that P $\subseteq$ NP since the polynomial $p(|x|)$ can be zero, meaning that $y$ is an empty string.

[3]The respective definitions are in compliance with [6].

There are well-known and widely studied combinatorial problems which are categorized into certain complexity classes such as P, NP or NP-Complete. These classes contain *decision problems* (*e.g.* Boolean satisfiability problem - deciding whether there exists an assigment for the arguments so that a given Boolean function evaluates to true) or *optimization problems* (*e.g.* finding a minimum cost traveling salesman tour). For *counting problems*, #P and #P-Complete are the correspondent classes to NP and NP-Complete. The complexity class #P is defined as follows:

**Definition 2.4.** *A function $f : \{0,1\}^* \to \mathbb{N}$ is in #P if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time deterministic Turing machine DTM such that for every $x \in \{0,1\}^*$:*

$$f(x) = \left| \left\{ y \in \{0,1\}^{p(|x|)} : DTM(x,y) = 1 \right\} \right| .$$

It follows that every *counting problem* is at least as hard as its *search problem*.
In general, we are interested in showing that a language $L'$ is at least as hard as some other language $L$. According to [6], the following definition of a *reduction* serves this purpose.

**Definition 2.5.** *A language $L \subseteq \{0,1\}^*$ is polynomial-time reducible to a language $L' \subseteq \{0,1\}^*$, denoted by $L \leq_p L'$, if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in L$ if and only if $f(x) \in L'$.*
*We say that $L'$ is NP-hard if $L \leq_p L'$ for every $L \in NP$. We say that $L'$ is NP-Complete if $L'$ is NP-hard and $L' \in NP$.*

For two given algorithms $L, L'$ with $L \leq_p L'$, we could compute the output $L(x)$, where $x$ is any valid input for $L$, by using algorithm $L'$ and a polynomial-time input-transforming function $f$ such that $L(x) = L'(f(x))$. It holds that if $L' \in P$ then $L \in P$ and $L'$ is at least as hard as $L$.

### 2.4.2. Reliability Polynomial

To relate the k-terminal reliability problem to known combinatorial problems which have already been classified into the mentioned complexity classes, we can write $R_K(G)$ ($K$ denoting the set of terminal nodes) in terms of a polynomial in $p$ (the *reliability polynomial* [7]), where we set $p_i = p,$ for all components $i \in \{1, \dots, q\}$:

$$R_K^p(G) = \sum_{i=1}^{q} N_i p^i (1-p)^{q-i}, \tag{2.10}$$

where $N_i := |\{M : |M| = i, \;\; M \in S\}|$ denotes the cardinality of the set of all subsets with $i$ working components connecting all nodes in $K$. For the two-terminal problem $K = \{s,t\}$, the set of subgraphs corresponds to the set of paths connecting the terminal nodes.
In this formulation we have $i$ working components and $q - i$ failed components. For instance, the two-terminal reliability in a *k-out-of-n* system, where all components operate with the same probability $p$, equals $\sum_{i=k}^{q} \binom{q}{i} p^i (1-p)^{q-i}$.
Another commonly used formulation for the *reliability polynomial* is with regard to the complements of pathsets:

$$R_K^p(G) = \sum_{i=0}^{q} F_i p^{q-i} (1-p)^i, \tag{2.11}$$

where $F_i := N_{q-i}$ and hence $F_i := |\{M : |M| = i,\ \ E \setminus M \in S\}|$.
This means that the set of working edges $E \setminus M$ of cardinality $q - i$ form a subgraph connecting the terminal nodes $K$.
Define a *minimal cardinality cut* to be a minimal set of components whose failure implies the system failure, more specifically, the deletion of the correspondent minimal set of edges disconnects any terminal node in $K$. Following this, a *minimal cardinality pathset* is a minimal set of working components which connects the terminal nodes and thus implying system operation. The following relations hold for $F_i$ and $N_i$:

$$\begin{array}{rcl} 0 \le F_i \le \binom{q}{i} & , & 0 \le N_i \le \binom{q}{i}, \text{ for } i = 0, 1, \dots, q, \\ F_i = \binom{q}{i}, \text{ for } i < c & , & N_i = \binom{q}{i}, \text{ for } i > q - c, \\ F_i = 0, \text{ for } i > q - l & , & N_i = 0, \text{ for } i < l, \end{array}$$

$$\begin{aligned} N_{q-c} &= F_c = \binom{q}{c} - n_c, \\ N_l &= F_{q-l} = n_l. \end{aligned}$$

Here $c$ = cardinality of a minimal cardinality cutset, $n_c$ = number of minimal cardinality cutsets, $l$ = cardinality of a minimal cardinality pathset and $n_l$ = number of minimal cardinality pathsets.
Determining each of the $F_i$ corresponds to a *counting problem*. If all coefficients $F_i$ can be computed, the exact reliability will be obtained.
According to Ball [7], this problem is called the *functional reliability analysis problem f-Rel* and the corresponding original problem, where each component has its own probability of failure $1 - p_i$, is called the *rational reliability analysis problem r-Rel*.
*f-Rel* takes a probabilistic graph and a value $p$ as inputs and returns all coefficients $F_i$. Instead *r-Rel* takes takes a probability vector $\vec{p}$ as second argument. Each entry $p_i$ of $\vec{p}$ corresponds to the reliability of component $i$. The output of *r-Rel* is a pair of integers $a, b$ with $\frac{a}{b}$ corresponding to the system reliability.

**Two- and all-terminal complexities** The minimum cardinality pathset and cutset search problems with respect to the two-terminal measure correspond to the shortest path and the minimal cut problems respectively. Although efficient algorithms exist for this [38, 51], the minimal cardinality cutset counting problem is however #P hard [56] implying the #P hardness of the two-terminal problem. For the all-terminal reliability problem, the corresponding minimal cardinality pathset and cutset problems are the minimal cardinality spanning tree[4] and minimal cardinality network cut problems[5]. Provan and Ball [56] showed that both two- and all-terminal reliability problems belong to the class of #P-Complete problems. They achieved this result by establishing a sequence of polynomial

[4]A tree is a connected undirected graph with no cycles. It is a spanning tree of a graph $G$ if it spans $G$ (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G). A spanning tree of a connected graph G can also be defined as a minimal set of edges connecting all vertices.
[5]A network cut in $G$ is any set of edges that disconnects a node $s \in V$ from any node in $V \setminus \{s\}$.

time reductions starting with a #P complete counting problem and leading further to the problem of counting minimal cardinality *s*-*t*-cuts[6] and network cuts. By a similar reduction technique, *f-Rel* can be reduced in polynomial time to the corresponding *r-Rel* [7]. In other words, if a particular functional reliability analysis problem is NP-hard then the corresponding rational reliability analysis problem is NP-hard.
Furthermore, Provan and Ball [56] showed that both undirected and directed reliability analysis problems are NP-hard. The undirected terminal-pair reliability problem even remains NP-hard for planar networks with node degrees bounded by three (see Provan [57]).

**k-terminal complexity** As one would expect, the k-terminal reliability problem which consists of computing all subgraphs connecting every pair of vertices of $K$, was proven by Valiant [76] to be NP-hard. The search for a minimal cardinality pathset with respect to $K$ corresponds to the problem of computing a minimal cardinality Steiner tree[7]. Since the corresponding search problem is already NP-hard for both directed and undirected networks [33], the associated *f-Rel* and *r-Rel* are NP-hard. Apart from series-parallel graphs[8], where k-terminal reliability can be determined in linear time [64], the unpromising results leave us with the conclusion that even polynomial time algorithms for solving reliability problems are very unlikely to exist for other restricted classes of networks. However, this should not be a reason for us to cease in our efforts in finding further improvements for the state-of-the-art exact solution techniques (see Chapter 5, 6 and 7). Although it is to be expected that even the most efficient algorithms have exponential complexity, it is however worth to find ways for reducing complexity. Even if the reduction is only of small magnitude, it may yet bring along significant improvements in runtime and memory consumption. Consequently, a higher number of input arguments could be processed.

## 2.5. Binary Decision Diagrams

A BDD (Binary Decision Diagram) is a data structure which corresponds to an $n$-dimensional Boolean function $f : \{0,1\}^n \to \{0,1\}$. This data structure stores all Boolean variable assignments along with their respective constant outcomes in a compressed form. Any Boolean function can be represented by a BDD which is a directed acyclic rooted graph having a set of decision nodes $N$ and two leafs labeled with $0$ (false) and $1$ (true).
There is one internal vertex, called the *root*, which has no incoming edge. Each decision node $v \in N$ is labeled by a Boolean variable $b := var(v)$ and has two outgoing edges connected to two child nodes, called low and high child. Edges pointing to a low/high child are represented as dashed/solid lines. The low/high child is reached when $b$ is assigned to $0/1$.
We denote $f_{b_i}$ to be the *restriction* of $f$ when some argument $b_i,\ 1 \leq i \leq n$ is replaced by a

[6]An $s,t$-cut in $G$ is any minimal set of edges that intersects every $s,t$-path.
[7]Given an undirected connected graph $G = (V, E)$ and a set of terminals $K \subseteq V$, a Steiner tree $T_G$ is a minimal cardinality set of edges connecting all nodes in $K$, whereas nodes from $V \setminus K$ may belong to $T_G$. The nodes in $V \setminus K$ are called "Steiner nodes" and furthermore the tree $T_G$ is an acyclic, connected subgraph of $G$.
[8]A series-parallel graph can be reduced by standard series, parallel and degree-2 reduction (an extension of the series reduction for problems with $|K| > 2$) to a single edge (see Section 3.2).

constant $c \in \{0,1\}$:

$$f_{b_i=c}(b_1,\ldots,b_n) = f(b_1,\ldots,b_{i-1},c,b_{i+1},\ldots,b_n).$$

Each BDD node, $v$, is itself a Boolean function $f$ which can be decomposed in terms of the associated Boolean variable $b$ (Shannon decomposition of $f$ with respect to $b$):

$$f_b = b \cdot f_{b=1} + \overline{b} \cdot f_{b=0}, \tag{2.12}$$

where $high(v) := f_{b=1}/low(v) := f_{b=0}$ is the high/low child of $v$.
The size of a BDD representing a Boolean function equals the number of decision nodes and highly depends on the variable ordering. A BDD with respect to a certain fixed variable order is called OBDD (Ordered Binary Decision Diagram) and it is called ROBDD (Reduced Ordered Binary Decision Diagram) if it is *reduced* implying the two following properties:

- **uniqueness**: no two distinct nodes $u$ and $v$ have the same label, low and high child, *i.e.*, $var(u) = var(v), low(u) = low(v), high(u) = high(v)$ implies $u = v$ (left side of Figure 2.2), and
- **no redundant nodes**: no internal node $u$ has identical low- und high child, *i.e.* $low(u) \neq high(u)$ (right side of Figure 2.2).

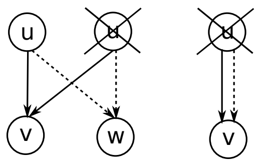


Figure 2.2.: Reducedness conditions.

The representation of a Boolean function in terms of a ROBDD is canonical [11]:

**Lemma 2.6.** *(Canonicity Lemma)*
*For any Boolean function $f : \{0,1\}^n \to \{0,1\}$ and a given variable ordering there is exactly one ROBDD denoting $f$, any other OBDD denoting $f$ contains more vertices.*

It follows that two Boolean functions are equivalent if their ROBDDs are isomorphic.

### 2.5.1. Operations

Without any decompression, operations or manipulations can directly be performed on a BDD graph. Many useful elementary operations (such as AND, OR, negation, testing for satisfiability, testing for equivalence of two functions) can be efficiently performed on this representation. Therefore Bryant [11] suggested efficient algorithms which have time complexities proportional to the size of the BDD graphs being manipulated. With $G_f$, denoting the BDD graph for a Boolean function $f$, the time complexities are listed in Table 2.3 for two BDD operations that are later used in this thesis, namely *Reduce* and *Apply* (there are more fundamental BDD operations which can be found in [11]). The *Reduce* algorithm transforms an OBDD graph into a ROBDD graph denoting the same Boolean function. It is based on the idea of the isomorphy test for trees (see Aho *et al.* [2]). We note, that the reduction complexity of $\mathcal{O}(|G_f| \cdot log|G_f|)$ established by Bryant, was brought down to linear time by Wegener [65].

Table 2.3.: Time complexities for BDD manipulation operations

| Operation | Input | Output | Time complexity |
|---|---|---|---|
| Reduce | OBDD $f$ | ROBDD $f$ | $\mathcal{O}(\|G_f\|)$ |
| Apply | ROBDD $f, g$ | ROBDD $f \circ g$ | $\mathcal{O}(\|G_f\| \cdot \|G_g\|)$ |

**Apply** The *Apply* operation takes two ROBDDs representing functions $f, g$ and produces the ROBDD for $f \circ g$, whereas $\circ$ can be any binary Boolean operation, such as AND or OR operation. This operation can also be used to test for implication (compare $\overline{f_1} \vee f_2$ to 1) or to complement a function. The *Apply* algorithm is derived from the Shannon expansion with regard to the input functions. They can be recursively composed in terms of a Boolean variable $b$:

$$f \circ g = b \cdot (f_{b=1} \circ g_{b=1}) + \overline{b} \cdot (f_{b=0} \circ g_{b=0}).$$

The algorithm takes two ROBDD root nodes $r_f, r_g$ of the argument graphs $G_f, G_g$. Proceeding downwards the argument graphs, vertices are created for the result graph $G_{f \circ g}$ at the branching points of the two argument graphs. For this there are several cases to distinguish. If $r_f, r_g$ are leafs, the result graph consist of a leaf with value $value(r_f) \circ value(r_g)$. Without loss of generality we impose the variable order $x_1 < x_2 < \ldots < x_n$ for the result BDD. Suppose at least one of the two root nodes is not a leaf. If $var(r_f) = var(r_g) = x_i$, we create a vertex $u$ with $x_i = var(u)$ and apply the algorithm recursively on $low(r_f)$ and $low(r_g)$ to obtain the subgraph whose root becomes $low(u)$, and on $high(r_f)$ and $high(r_g)$ to create the subgraph with root $high(u)$. Otherwise, suppose that $var(r_f) = x_i$ and either $r_g$ is a leaf or $var(r_g) > x_i$, which means that the function represented by the graph with root $r_g$ is independent of $x_i$. Then a node $u$ with $var(u) = x_i$ is created and the algorithm is recursively applied to $low(r_f)$ and $r_g$ to generate the subgraph with root node $low(u)$, and on $high(r_f)$ and $r_g$ to generate the subgraph with root node $high(u)$. To cover all cases, exchange the roles of the two argument root nodes.
The complexity of this operation can possibly be decreased by maintaining a table which stores the result node $u$ for each application of the algorithm on root nodes $r_f, r_g$. Each time the algorithm is applied, we first have to check for the existence of the pair of root nodes inside the table. If these exist, we directly return the result node $u$. Otherwise, we proceed as described before and add a new entry to the table. For further implementational details we refer to [3]. The application of the *Apply* method on two small ROBDDs is exemplified in Figure 2.3.

**Complexity of finding an optimal variable ordering** The complexity of BDD manipulation algorithms heavily depends on the size of the BDD input graphs. Thus, the BDD size should preferably be minimal under the optimal variable ordering. Unfortunately, the problem of finding the optimal variable ordering is NP-complete [21]. There are $n!$

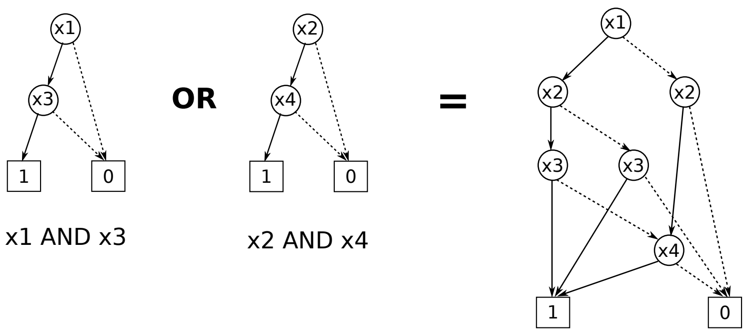


Figure 2.3.: Example of *Apply*, variable ordering: $x_1 < x_2 < x_3 < x_4$

possible variable orderings for a function of $n$ variables. The best known algorithm for determining the optimal variable ordering has complexity $\mathcal{O}(n^2 3^n)$ [21].

# Part II.

# State of the Art

# 3. Techniques for exact k-terminal reliability computation

There exists an extensive literature treating the exact computation of the two-terminal, k-terminal and all-terminal reliability. In this chapter, we give an overview of the most important techniques that have evolved over time, listed in accordance with their efficiency - starting with the least efficient path and cut enumeration technique. For some techniques, we first explain the two-terminal case since the k-terminal case works in a similar way or can be easily deduced from the two-terminal version.

## 3.1. Path and Cut Enumeration

The k-terminal reliability can be computed by first enumerating minimal paths (minpath) or minimal cuts (mincuts)[1] which gives the system's structure function in terms of a Boolean expression. In the second step the probability of the obtained Boolean expression is determined by either applying the inclusion-exclusion method or the sum of disjoint products technique. The second step is needed, since minimal paths or cuts are not necessarily mutually disjoint.

### 3.1.1. Enumerating minimal paths

Denote $\mathcal{P}$ to be the set of all minimal paths of a graph $G$ and let $|\mathcal{P}| = h$. A minimal path $P_i$, $1 \leq i \leq h$ corresponds to a conjunction term $T_{P_i}$ that consists of Boolean variables representing working edges in $P_i$. For a minimal path comprising $N$ different components, the respective conjunction term $T_{P_i}$ is as follows:

$$(x_1 \wedge x_2 \wedge \ldots \wedge x_N).$$

The structure function $X$ is then expressed as:

$$X_G = \bigvee_{i=1}^{h} T_{P_i}.$$

For obtaining the k-terminal reliability, we need to compute:

$$R_K(G) := \mathbb{P}(X_G = 1) = \mathbb{P}\left(\bigvee_{i=1}^{h} T_{P_i}\right).$$

[1] The subtle difference with regard to the definitions given in Section 2.4.2 is the unfixed cardinality of the respective set of edges causing system operation/failure. The paths/cuts are called minimal, since none of their subsets is also a path/cut.

We refer to [58] for the computation of all minpaths with regard to the two-terminal reliability[2]. For the k-terminal reliability, Steiner trees are computed by using the method in [50].

### 3.1.2. Enumerating minimal cuts

Analogously, denote $\mathcal{C}$ to be the set of all minimal cuts and let $|\mathcal{C}| = g$. Each minimal cut $C_i$, $1 \leq i \leq g$ corresponds to a conjunction term $T_{C_i}$ which is composed of Boolean variables representing failed edges in $C_i$. We say that a minimal cut $C_i$ fails if all components in $C_i$ fail, implying system failure. The conjunction term $T_{C_i}$ of a minimal cut with $N$ different components is then

$$(\overline{x}_1 \wedge \overline{x}_2 \wedge \ldots \wedge \overline{x}_N).$$

In order to achieve system operation, no mincuts are allowed to fail. Thus, the k-terminal reliability is the complementary probability that at least one minimal cut fails:

$$R_K(G) := 1 - \mathbb{P}(\overline{X_G} = 1) = 1 - \mathbb{P}\left(\bigvee_{i=1}^{g} T_{C_i}\right),$$

here $\overline{X_G}$ is the negated structure function of the system (as explained in [45]). In terms of the two-terminal reliability, one possible method for computing all mincuts can be found in [32].

### 3.1.3. Inclusion-exclusion method

Poincaré's theorem can be used to compute the probability of the Boolean expression $X_G$ (analogously for $\overline{X_G}$):

$$\mathbb{P}\left(\bigvee_{i=1}^{h} T_{P_i}\right) = \sum_{i=1}^{h} \mathbb{P}(T_{P_i}) - \sum_{1 \leq i_1 < i_2 \leq h} \mathbb{P}(T_{P_{i_1}} \cdot T_{P_{i_2}}) + \ldots + (-1)^{h-1} \cdot \mathbb{P}\left(\prod_{i=1}^{h} T_{P_i}\right). \quad (3.1)$$

The inefficiency of this method is rooted in the doubly-exponential complexity: the number of minpaths, $h$, might be exponential in $|C|$ and so $2^h - 1$ terms are generated in Equation 3.1. Thus, in practice one reverts to other, more efficient, methods for the exact computation. However, the inclusion-exclusion method can be used for obtaining lower and upper bounds of the reliability by only accounting the first terms of the sum. A lower/upper bound is obtained by interrupting the computation after a negative/positive term.

### 3.1.4. Sum of disjoint products

A more efficient way is to first disjoint the conjunction terms, $T_{P_i}$, by using the SDP (sum of disjoint products) and sum up the probabilities of the disjoint terms afterwards. The

[2]Here the minpaths correspond to *s*-*t*-paths.

SDP is based on the Shannon decomposition and has the following representation when applied to the DNF of $X_G$:

$$\bigvee_{i=1}^{h} T_{P_i} = T_{P_1} + \sum_{i=2}^{h} T_{P_i} \cdot \prod_{j=1}^{i-1} \overline{T}_{P_j}. \tag{3.2}$$

Hereafter, the probability for each term can be easily derived. Finally, they are summed up to obtain the reliability value:

$$\mathbb{P}\left(\bigvee_{i=1}^{h} T_{P_i}\right) = \mathbb{P}(T_{P_1}) + \sum_{i=2}^{h} \mathbb{P}\left(T_{P_i} \cdot \prod_{j=1}^{i-1} \overline{T}_{P_j}\right). \tag{3.3}$$

We note that Equation 3.2 is again applied to $\overline{T}_{P_j} = \left(\bigvee_{k=1}^{N} \overline{x}_k^j\right)$ to obtain the following simplified expression:

$$\bigvee_{i=1}^{N} \overline{x}_k^j = \overline{x}_1^j + \sum_{i=2}^{N} \overline{x}_i^j \cdot \prod_{k=1}^{i-1} x_k^j.$$

There are several different strategies for computing the SDP. For instance one can use the Abraham method [1]. Though this method exhibits better performance than the inclusion-exclusion method for many practical examples, its worst case complexity is exponential, since the number of conjunction terms (minimal cuts or paths) may exponentially grow with the number of system components. In addition, there is no known polynomial bound for the length of the simplified Boolean expression (left side of Equation 3.3) in terms of the number of minpaths or mincuts [7].

### 3.1.5. Summary of this section

Instead of enumerating all $2^{|C|}$ states, minimal cutsets and pathset are enumerated. One hopes to explore a smaller number of system states by enumerating either pathsets or cutsets. As mentioned in Section 2.4.2, the enumeration of all minimal cutsets and pathsets is #P hard [76, 57] . Hence, in the worst case this technique requires exponential time. Later in Section 3.3, we will outline an approach which explores only the relevant states and is hence more efficient than the path and cut enumeration method.

## 3.2. Reduction techniques

The exponential complexity of the reliability problem (see Section 2.4) gave rise to the focus of research on developing reduction techniques for certain graph structures which aim to significantly reduce the graph size or its structural complexity. It is hoped that reduction will take at most polynomial time to obtain an exponential improvement. In fact, in many of the cases only local knowledge (*e.g.* node degrees) is needed in order to detect reducible structures. A reduction is defined with respect to an input graph $G$, its terminal nodes $K$ and the given edge failure probabilities. Let $G$ be a graph with at least two terminal nodes. Reducing $G$ involves replacing or deleting edges and nodes to form a new graph $G'$ which has adapted edge probabilities and a new terminal set $K'$ such that

$R_K(G) = \Omega R_{K'}(G')$, whereas $\Omega$ is a constant factor. Basically, a reduction changes the graph structure while preserving reliability. In the following, we highlight three major types of reductions: series-parallel reductions, polygon-to-chain reductions and deletion of irrelevant components (see Wood [37, 82]).

### 3.2.1. Series-parallel and degree-2 reductions

According to Wood [37], a *series reduction* replaces two edges $e_a = (u, v)$ and $e_b = (v, w)$, such that $u \neq w$, $deg(v) = 2$, $v \notin K$, with a single edge $e_c = (u, w)$ and it follows that $p_c = p_a p_b$, $K' = K$, $\Omega = 1$. A *parallel reduction* replaces a pair of edges $e_a = (u, v)$ and $e_b = (u, v)$ with a single edge $e_c = (u, v)$ and $p_c = p_a + p_b - p_a p_b$, $K' = K$, $\Omega = 1$. Now suppose that $e_a = (u, v)$, $e_b = (v, w)$ and $u \neq w$, $deg(v) = 2$, $\{u, v, w\} \subseteq K$, then a *degree-2* reduction replaces $e_a$ and $e_b$ with a single edge $e_c = (u, w)$ and $\Omega = p_a + p_b - p_a p_b$, $p_c = \frac{p_a p_b}{\Omega}$, $K' = K \setminus \{v\}$.
Without any special distinction of terminal and non-terminal nodes, a *series-parallel graph* is a graph that can be reduced to a tree by successive series and parallel replacements. If a series-parallel graph is biconnected[3], then the tree consists of only one edge. Wood [37] further defines a graph $G$ to be irreducible series-parallel if $G$ cannot be completely reduced to a single edge using simple reductions. In combination with the following important type of reductions (polygon-to-chain reductions), the k-terminal reliability of a series-parallel graph can be computed in linear time [64].

### 3.2.2. Polygon-to-chain reductions

As the name states, polygon structures consisting of at most six edges are reduced to chains of at most length three. In fact, those polygon structures can be found in linear time by triconnected decomposition [27]. In general, polygon-to-chain reductions always reduce $|V| + |E|$ by at least one. A *chain* $\chi$ is defined as an alternating sequence of distinct nodes and edges, $v_1,\ e_1,\ v_2,\ e_2,\ \dots,\ v_{k-1}, e_{k-1},\ v_k,\ e_k,\ v_{k+1}$, where $e_i = (v_i, v_{i+1})$, internal nodes $v_i, i = 2, \dots, k$ are of degree two and border nodes $v_1, v_{k+1}$ can have degree more than two. For two chains $\chi_1, \chi_2$ with length $l_1, l_2$ and common border nodes $v_1, v_{k+1}$, the resulting *polygon* $\Delta = \chi_1 \cup \chi_2$ is of length $l_1 + l_2$. Since (i) every degree-2 node of $G$ is a terminal node, (ii) a chain cannot have more than two terminal nodes, and (iii) the length of a chain is at most three, the polygons in $G$ - if they exist - can only be one of the seven types shown by Satyanarayana and Wood [64] (see also Lucet and Manouvrier [49]). For the two-terminal case, there are only five types and no such reduction is possible for the all-terminal case. Wood used conditional probability and distinguished between possible configurations of subgraph $(G \setminus \Delta) \cup (\{v_1\} \cup \{v_{k+1}\})$ to conclude $K'$, $\Omega$ and the new edge probabilities for the resulting chains. For a detailed derivation, we refer to Satyanarayana and Wood [64].

### 3.2.3. Removal of irrelevant components

According to Wood [82], any connected component $G^0 = (V^0, E^0)$ of $G$ which is connected to $(G \setminus V^0) \cup \{v^c\}$ by an articulation vertex $v^c$ is irrelevant and hence can be removed, only if $K \cap (V^0 \setminus \{v^c\}) = \emptyset$. Consequently, $G' = (G \setminus V^0) \cup \{v^c\}$, $K' = K$ and $\Omega = 1$. Furthermore,

[3] A connected graph $G$ is biconnected if at least two nodes must be deleted to disconnect $G$.

any self-loop is irrelevant.
The recognition of those irrelevant components can be efficiently done in linear time (see Hopcroft and Tarjan [28]). We will later make use of this removal for significantly improving a state-of-the-art exact approach (see Chapter 5).

## 3.3. Factoring with reductions

When an undirected graph $G$ does not further allow any reductions, the reliability of $G$ can be decomposed by applying conditional probability with respect to an edge $e$ that is known as the *keystone edge*. By *factoring* on $e$, we obtain for the reliability of $G$ (*Factoring theorem* [52]):

$$R_K(G) = p_e \cdot R_{K'}(G * e) + (1 - p_e) \cdot R_K(G - e). \tag{3.4}$$

where $G * e$ is $G$ with edge $e = (u, v)$ contracted and $G - e$ is $G$ with $e$ deleted. $K' = K$ if $u, v \notin K$ or $K' = (K \setminus \{u, v\}) \cup (\{u\} \cup \{v\})$ if $u \in K$ or $v \in K$. So $G * e$ represents the subgraph with $e$ working and $G - e$ the subraph with $e$ failed. The reductions can then again be applied to either of the subgraphs. The factoring recursion ends when the terminals are either disconnected or merged into one single node. In general, the recursion depth is limited by the number of components.

### 3.3.1. Choice of the keystone edge

In Figure 3.1, we have applied the factoring with series-parallel reductions on a small size grid network. After two series reductions, we choose edge $e3$ for factoring. This results in two subgraphs that can be reduced to a single edge by alternating series and parallel reductions. The respective recursion graph consisting of three nodes is depicted on the right upper corner of Figure 3.1. $L_{s,t}(G)$ is the number of leafs - which is two - of the recursion graph. The number of leafs in the factoring recursion tree is used as a measure of the complexity of a factoring algorithm and defined as $L_{s,t}(G) = \frac{N_{s,t}(G)+1}{2}$ [63], where $N_{s,t}(G)$ is the total number of nodes in the recursion tree. Apparently, the target is to obtain a minimal number of leafs. The minimal number of leafs can be attained by an optimal edge-selection strategy for factoring with respect to the used reductions. Therefore, Satyanarayana and Chang [63] gave an optimal strategy for the choice of the keystone edge with respect to series-parallel reductions, such that $G * e$ and $G - e$ remain *coherent*[4] after factoring. In general, if we arbitrarily choose an edge for factoring, irrelevant components may evolve in the course of a factoring procedure. Hence, redundant computation will be carried out if one does not delete irrelevant components. In our example, the choice of edge $e3$ complies with the optimal strategy, since the resulting subgraphs are biconnected and contain no irrelevant components.

[4]A graph $G$ is *coherent* if the cardinality of the terminal set is at least two and G contains no irrelevant edges or vertices [82].

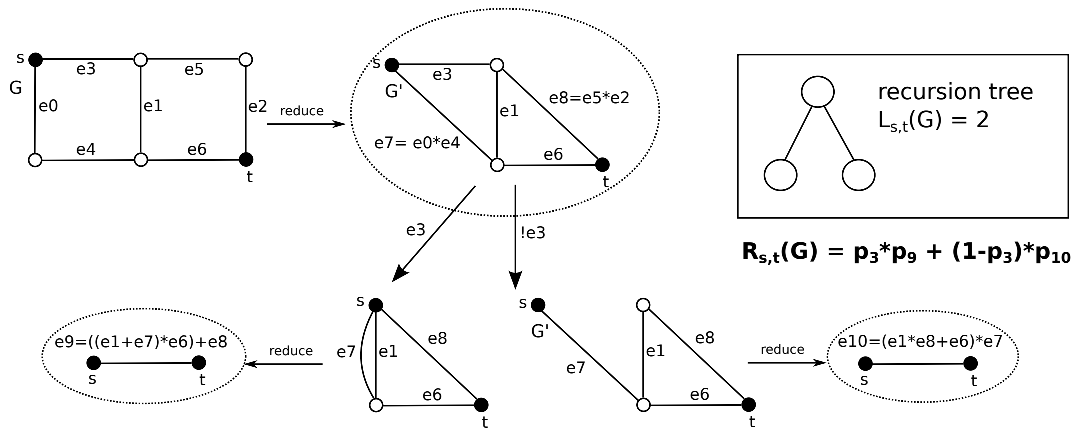


Figure 3.1.: Factoring with series-parallel reductions.

### 3.3.2. Incorporating triconnected decomposition in a factoring with reductions algorithm

Wood suggested to add triconnected decomposition to the factoring with reductions algorithm and worked out conditions for the two- and all-terminal case which ensures that factoring with triconnected decomposition and reductions is at least as good as a pure factoring algorithm [83]. To be precise, the generated number of recursion-tree leafs for factoring with triconnected decomposition and reductions is never more than the number of leafs for a pure factoring algorithm. Unfortunately, the conditions are rather restrictive for the k-terminal case. By triconnected decomposition, a graph $G$ can be decomposed along its separating node-pair $\{u, v\}$ such that $G = \tilde{G} \cup G^+$, $\tilde{V} \cap V^+ = \{u, v\}$, $\tilde{E} \cap E^+ = \emptyset$, $|\tilde{E}| \geq 2$ and $|E^+| \geq 2$. Using the same argumentation for deriving polygon-to-chain reductions, Wood shows that $G^+$ may be replaced by a chain $\chi$ of at most three edges between $u$ and $v$ such that $R_K(G) = \Omega R_{\tilde{K}'}(\tilde{G} \cup \chi)$. Therefore one has to only solve one, two, three, or four k-terminal reliability problems defined on $G^+$. The discussed technique is briefly illustrated in Figure 3.2. Note that we have not distinguished between non-terminal and terminal nodes. For a detailed insight, we refer to [83]. Pertaining the polygon-to-chain reductions, $G^+$ corresponds to certain polygon structures derived in [64].

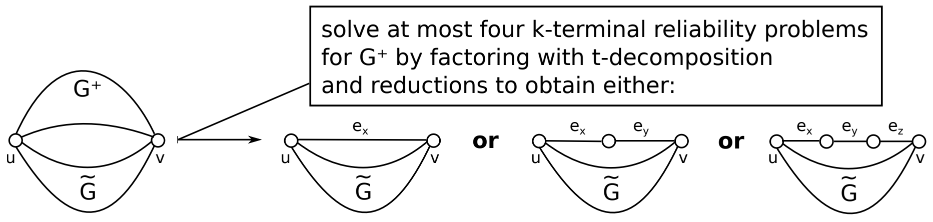

solve at most four k-terminal reliability problems for G⁺ by factoring with t-decomposition and reductions to obtain either:
G⁺
u
v
G̃
$e_x$
or
$e_x$
$e_y$
or
$e_x$
$e_y$
$e_z$

Figure 3.2.: Triconnected decomposition and chain replacements.

### 3.3.3. Considering node failures

The failure of nodes can be taken into account by factoring on nodes like on edges. Unfortunately, this exponentially increases the complexity with the number of nodes. To avoid this, Carlier and Theologou modified the factoring with reductions algorithm in a way that imperfect nodes can be considered with only small additional cost: factoring is still conducted on edges and after each factoring step the node failure probabilities are adapted for the induced graphs. The factoring with reductions algorithm is practically left unchanged except that polygon-to-chain reductions can no longer be used. Though this restriction can significantly increase the complexity, it is still more efficient than factoring on nodes. For details and respective performance measurements on sample networks, we refer to [73].

### 3.3.4. Bottom line

The efforts to find clever ways to reduce the combinatorial complexity of reliability computation resulted in several efficient reduction techniques. The best results that could be achieved by appropriately applying the discussed reductions in conjunction with the factoring technique are:

1. Determination of the k-terminal reliability for series-parallel graphs in linear time (Satyanarayana and Wood [64]).
2. Significantly reducing the complexity of a pure factoring algorithm in terms of two- and all-terminal reliability for general graphs (Wood [83]).

Additionally, Chang and Satyanarayana showed that factoring with reductions [63] is more efficient than cut or path enumeration. Nevertheless, the best factoring with reductions algorithm has exponential complexity and care should be taken to the exhaustive memory needed for storing subgraphs evolving in the course of factoring. Hence, not only the time factor poses a limitation to the reliability computation but also the high memory requirements. Apart from this, the major disadvantage of applying reductions is that numerical adaptations to edge reliability prevent the creation of the more desired symbolic representation of the combinatorial graph structure: in this case the reliability computation does not need to be conducted another time if certain edge probabilities have changed. Facing this direction and turning away from the reductions, we will present in the next two sections the two state-of-the art approaches which efficiently extract and symbolically store the combinatorial structure of the reliability graph. In addition, redundant computations are avoided by *dynamic programming*[5] and elegant representations for subgraphs or subproblems are defined to keep the memory consumption moderate.

## 3.4. Kuo-Lu-Yeh approach

Among all efficient algorithms solving the terminal-pair problem, the KLY approach is currently the most efficient approach besides the decomposition approach (see Section 3.5). No one, however, has opposed the performance of these two approaches in a fair

[5]Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. When two subproblems overlap, the solution of the already solved subproblem is reused.

manner. We refer to Section 8.2 for an extensive evaluation and comparison of these two approaches.
The central ideas making the KLY approach so efficient, are the clever use of OBDDs and the recognition of reliability isomorphic subgraphs. The results from KLY [85] reveal a vast improvement in computation time when compared with efficient approaches which do not make use of OBDDs. Since the efficiency of the KLY approach highly depends on the size of OBDDs that represent Boolean functions coming up in the course of the algorithm, the goal must be to keep their size as small as possible to allow efficient OBDD manipulations. As described in Section 2.5, two Boolean functions $f$ and $g$ can be recursively composed by basic Boolean operations such as AND, OR $=: \circ$ in terms of a Boolean variable $b$:

$$f \circ g = b \cdot (f_{b=1} \circ g_{b=1}) + \bar{b} \cdot (f_{b=0} \circ g_{b=0}).$$

These compositions are applied iteratively to construct the OBDD path function during the KLY algorithm. However, the size of an OBDD notably depends on the variable ordering. As we already stated in Section 2.5, the problem of finding an optimal variable ordering is NP-complete [21]. Hence, Kuo, Lu and Yeh propose the bfs (breadth-first-search) variable order to be a good alternative for the terminal-pair reliability problem. They empirically obtained the best results with bfs.
Since the essential success of the KLY approach lies on the recognition of reliability isomorphic subgraphs, we therefore give a definition at this point.

**Definition 3.1.** *Two coherent terminal-pair networks are reliability isomorphic if they coincide topologically and each of their edges and nodes represent the same component.*

By recognizing reliability isomorphic subgraphs, the reliability computation of several reliability isomorphic subgraphs can hence be reduced to just one of these subgraphs. It is therefore a matter of concern to compactly store and rapidly retrieve subgraphs in order to keep the overhead as low as possible. In Chapter 5, we explain our way of implementing this.

### 3.4.1. The approach

We now recapitulate the original approach [85] (see Algorithm 2). Instead of explicitly enumerating all $s$-$t$-paths of a given network, they are efficiently stored in an OBDD by conducting the following recursion: starting with $s$, all its adjacent edges in the initial graph $G$ are visited (in dfs order). This is done by deleting the node where $s$ is originally located along with all its adjacent edges. Thereafter, $s$ is relocated to one of the adjacent nodes which was chosen to be visited over edge $e$. This operation is called contraction of $G$ with $e$: $G * e$ (see line 10 of Algorithm 2). A new subgraph, $sub_G$, emerges from this graph manipulation step. Redundant nodes which are not part of any $s$-$t$-path, are removed from $sub_G$ (line 11). Those are nodes unequal $s$ or $t$ with degree one. Then $sub_G$ is again processed as described. According to [85], each recursive call stands for a $G_{node}$. Each $G_{node}$ has outgoing edges to its child $G_{node}$ whereas each edge is labeled with the Boolean variable of the visited edge. Such a recursion graph is called an Edge Expansion Diagram (EED) which is a directed acyclic graph. The above recursive step is repeated until $s$ has reached $t$. The first time we return from the recursion, the OBDD is created

for a Boolean variable $B$ labeling the last visited edge $e$ between $s$ and $t$. Then the just-constructed OBDD is put into a global hashmap with the subgraph consisting of one edge $e$ as the hashmap key. The central idea is to avoid redundant computations by detecting reliability isomorphic subgraphs (see below). This is done by incrementally building a hashmap from the computed subgraphs with their corresponding OBDD value. At each recursion, the subgraph to be computed is looked up in the global hashmap and if there is a match (or hit), the appropriate OBDD will be returned (see line 5-6).
The returned OBDD is composed with other returned OBDDs coming from the adjacent branches - if they exist - of the current $G_{node}$ by the OR operation to a new OBDD which is returned to the parent $G_{node}$ (see line 14). The final OBDD, representing the structure function $X_G$ with respect to graph $G$, is obtained as soon as all recursions return to the root of the EED. According to [85], the structure function is expressed as follows:

$$X_G = \sum_{i=1}^{k} B_i \cdot X_{G*e_i},$$

where $k$ stands for the number of $s$-adjacent edges und $B_i$ is the Boolean variable for edge $e_i$. The number of $G_{node}$ in the EED equals the number of hashmap entries, where each hashmap entry value is a reference to the BDD root node of a Boolean function. Since all paths from the root to the terminals of the OBDD are mutually disjoint, the probability of the implied Boolean function, equal to the reliability value, can be easily evaluated by Algorithm 3 (suggested by KLY [85]). The whole algorithm is subsumed in the Main Algorithm 1.

Algorithm 1: KLYMain

```
Require: Graph $G$, source $s$, sink $t$
 1: find a variable ordering for $G$ according to bfs
 2: $bddresult$ =PathConstruct($G, s$)
 3: Reliability = Prob($bddresult$)
```

### 3.4.2. Case study

By means of a 2x3 ladder network $G$, we briefly demonstrate the workings of the KLY approach. The left side of Figure 3.3 shows the respective EED. It consists of eight $G_{node}$ where the $G_{node}$ are numbered according to a possible processing order. They are shown together with their respective Boolean expressions. In the course of the KLY method, BDD-apply operations are conducted many times. The first time the BDD-apply operation is carried out in this example is at $G_{node}$ G5: $e_2$ AND $true$. The resulting OBDD is then hashed together with the underlying subgraph, serving as its key. In $G_{node}$ G6 and G7, the redundant edges $e_4, e_2$, highlighted with dotted lines, are recognized and removed. Only one hit (one isomorphic graph) occurs in this example: at $G_{node}$ G5, since more than one EED-edge points to it. Finally, the resulting ROBDD (depicted on the right side of Figure 3.3) is returned at $G_{node}$ G0. Algorithm 3 traverses the ROBDD in dfs order. Each ROBDD node is labeled with the respective variable and in addition the computed probabilities are

Algorithm 2: PathConstruct

```
Require: Graph $G$, currentsource $s$
 1: BDD $bddnode, bddresult$
 2: if $currentsource == t$ then
 3:    return $bddtrue$
 4: end if
 5: if $(bddresult = hashmap.find(G, s))$ is a hit then
 6:    return $bddresult$
 7: end if
 8: $bddresult = bddfalse$
 9: for all $s$-adjacent-edges $e$ do
10:    $sub_G = G * e$
11:    $removeRedundantNodes(sub_G)$
12:    $bddnode$ =PathConstruct$(sub_G, source_{sub_G})$
13:    $bddnode = BDD.apply(bbd(e), bddnode,$ AND$)$
14:    $bddresult = BDD.apply(bddresult, bddnode,$ OR$)$
15: end for
16: $hashmap.put(G, s, bddresult)$
17: return $bddresult$
```

Algorithm 3: Prob

```
Require: BDD $bddnode$
 1: $//p$ is the reliability of the component represented by the $bddnode$ variable
 2: if $bddnodeisbddtrue$ then
 3:    return $1$
 4: end if
 5: if $bddnodeisbddfalse$ then
 6:    return $0$
 7: end if
 8: if $(result = hashmap.find(bddnode)) \neq NULL$ then
 9:    return $result$
10: else
11:    $result = p\cdot$ Prob$(bddnode.high) + (1-p)\cdot$ Prob$(bddnode.low)$
12:    $hashmap.put(bddnode, result)$
13: end if
14: return $result$
```

attached. Here we assume that all components fail with the same probability of 0.1. In the end the seeked reliability $R$ value is returned at the root node $e_0$.

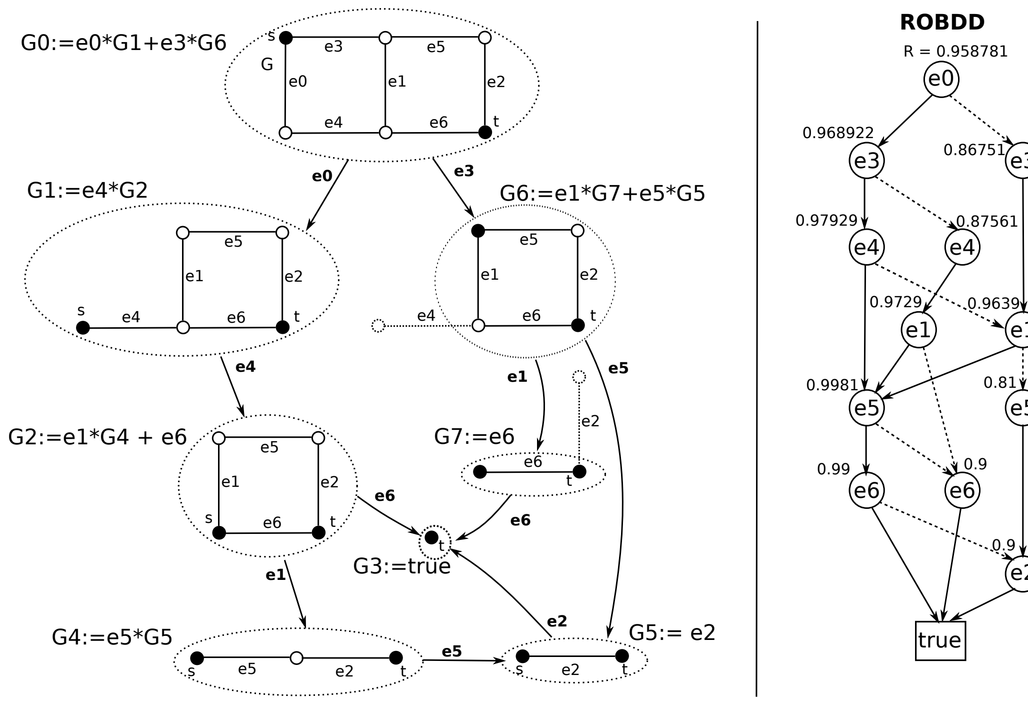


Figure 3.3.: Left: Edge Expansion Diagram, Right: Resulting ROBDD.

### 3.4.3. Complexity discussion

Though the KLY approach has exponential complexity with the number of components in the worst case, it turns out that for many different network structures a much better performance is attained, as empirically shown in Section 8.2.
According to [85], in the best case the expansion of the OBDD terminates in $h$ steps, where $h$ equals the length of the shortest path. This is the case when the topology of the input network results in identical isomorphic subgraphs during each expansion iteration.
In the worst case, a very large expansion is generated if there are no isomorphic subgraphs. The generated number of $G_{node}$ has exponential complexity, $\mathcal{O}((d_m)^k)$, where $d_m$ is the maximal out-degree among all nodes and k the maximal *s*-*t*-path length.

### 3.4.4. Extension for k-terminal case

For the undirected k-terminal case, Yeh et. al [84] defined a *feasible set* to be a node set where all nodes in $K$ are covered by $|K| - 1$ terminal-pairs. Then the k-terminal reliability can be obtained by applying the Boolean $AND$ operation to all the structure functions of the $k-1$ two-terminal networks. Thus, the resulting structure function is expressed as

$$X_G = \prod_{l=1}^{|K|-1} X_G(n_s^l, n_t^l), \tag{3.5}$$

where $X_G(n_s^l, n_t^l)$ is the structure function with respect to terminals $n_s^l, n_t^l$ and $n_s^l, n_t^l \in K, n_s^l \neq n_t^l$ for all $l$. Since the terminal set $K$ is covered by a feasible set, it holds that

$$\bigcup_{l=1}^{|K|-1} \left(n_s^l \cup n_t^l\right) = K. \tag{3.6}$$

To avoid redundant computation, the selected nodes $n_s^l, n_t^l$ must be marked and treated as terminal nodes during the path construction algorithm. Their path constructing process can be terminated earlier when multiple terminals exist.

### 3.4.5. Accounting for node failures

With little overhead, imperfect nodes can be considered by *incident edge substitution*: for the undirected case, the Boolean variable $B_i$ of an edge $e_i = (n_a^i, n_b^i)$, is simply replaced with $n_a^i \wedge B_i \wedge n_b^i$, where $n_a^i$ and $n_b^i$ represent the Boolean variables of the two respective end nodes of edge $e_i$. Following this, the structure function is represented as

$$X_G = \sum_{i=1}^{k} (s \wedge B_i \wedge n_b^i) \wedge X_{G*e_i}. \tag{3.7}$$

On a whole, the KLY approach is carried out for the original graph G as if all nodes are perfect. Subsequently, the composition operation is used to recursively incorporate the incident edge substitution on the resulting BDD, $BDD(G)$. With regard to a predetermined variable order $s < e_1 < n_b^1 < n_a^2 < e_2 < n_b^2 < \ldots < n_a^m < e_m < t$, the appropriate composition operation is expressed in terms of the following Boolean expression:

$$BDD(G) = \left((s \wedge B_1 \wedge n_b^1) \wedge BDD(G)_{|B_1=1}\right) \vee \left(\overline{(s \wedge B_1 \wedge n_b^1)} \wedge BDD(G)_{|B_1=0}\right). \tag{3.8}$$

Here, $BDD(G)_{|B_k=1}$ and $BDD(G)_{|B_k=0}$ are the high and low BDD child nodes of Boolean variable $B_k$ respectively. In addition, the Boolean variable for terminal $t$ must be returned, instead of *bddtrue*. Finally, the reliability value is computed by traversing the result BDD with Algorithm 3. Implementational details, extensions for the directed case and results for the additional overhead can be found in [36].

## 3.5. Decomposition method

As mentioned in Section 3.4, the decomposition approach is on the one hand along with the KLY approach the currently most efficient approach for exact terminal-pair computation and on the other hand competing with the KLY approach only with regard to undirected networks. Unlike the KLY approach, the decomposition approach does not record directed path information, so that its application is restricted to undirected networks. We remark that it has not yet been established to make the decomposition method applicable for directed networks. Its great strength however is the unsurpassed effective calculation of regular undirected network structures. It is based on factoring and consists of three parts: first, determine a suitable edge ordering. Then, all edges are consecutively factored according to the stipulated ordering. Concurrently, a result BDD is iteratively created until all edges have been factored. Finally, the created result BDD is traversed by Function 3 to obtain the reliability value. In what follows, we reveal the details of the decomposition approach. Its main idea can be attributed to Rosenthal [61] who suggested to classify the different network states by means of a set of frontier nodes (boundary set) $F_k$.

### 3.5.1. Boundary set and partitions

Suppose, that the initial ordering is $e_1 < e_2 < \ldots < e_k < \ldots < e_m$. At factoring-level $k$, we have $E_k := \{e_1, e_2, \ldots, e_k\}$ and $\overline{E_k} := \{e_{k+1}, \ldots, e_m\}$. The already factored edges are in the set $E_k$ and have a fixed state (either working or failed). The edges that still have to be factored are contained in $\overline{E_k}$. For a graph $G = (V, E)$, let $A = (V_k, E_k)$ and $B = (\overline{V_k}, \overline{E_k})$ be two subgraphs of $G$ with $E_k, \overline{E_k} \subseteq E$ and $E_k \cup \overline{E_k} = E, E_k \cap \overline{E_k} = \emptyset$, $V_k, \overline{V_k} \subseteq V$ and $V_k \cup \overline{V_k} = V$. Furthermore, $F_k := V_k \cap \overline{V_k}$ is called the k-th frontier set. The maximal size of the frontier sets is defined as $|F_{max}| := \max\limits_{i=1,2,\ldots,m-1} |F_i|$. Thus, we have $V_k := \bigcup_{i=1}^{k} F_i$ and $\overline{V_k} := (V \setminus V_k) \cup F_k$. Subgraph $A$ and its complement $B$ are depicted in Figure 3.4. On the right side of Figure 3.4 $|F_k| \geq 2$, whereas on the left side $F_k$ consist of only one vertex, called the articulation vertex. In this case, the reliability of $G$ can be

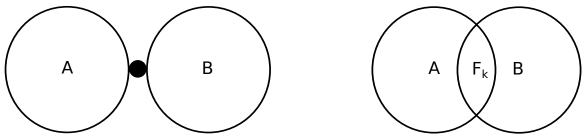


Figure 3.4.: Left: Articulation vertex, Right: Frontier set $F_k$.

computed as $R(G) = R(A) \cdot R(B)$. Rosenthal defined an equivalence relation for $F_k$ [61]: vertices from $F_k$ are arranged into one *block*, if there is a connecting path in $A$ connecting them. The equivalence classes are uniquely represented by *partitions* consisting of at least one block. All vertices in a block can be seen as being grouped into a single vertex. So $F_k$ are the vertices needed to encode the current network state in the k-th level. The partitions describe the connections between the frontier vertices and also record whether each vertex in $F_k$ is also connected to any of the terminals $K$.

### 3.5.2. Number of partitions

The order of the partitions stems from the *Stirling numbers of the second kind* and has the following recursive formula:

$$A_{i,j} = j \cdot A_{i-1,j} + A_{i-1,j-1}, \text{ if } 1 < j \leq i \text{ with } A_{i,1} = 1 \text{ and } A_{i,j} = 0 \text{ if } 0 < i < j$$

$A_{i,j}$ is the number of partitions of $j$ blocks made of $i$ nodes. In Table 3.1 we have explicitly computed for $2 \leq |F| \leq 13$ the number of partitions consisting of $i$ blocks, with $1 \leq i \leq |F|$. The total number of partitions $B_{|F|}$ is known as the *Bell number* and defined as (see [23]):

$$B_{|F|} = \sum_{j=1}^{|F|} A_{|F|,j}. \tag{3.9}$$

$B_{|F|}$ grows exponentially with the size of $F$. In the third column of Table 3.2, $B_{|F|}$ is explicitly determined for the all-terminal case. The maximal number of partitions for the two-terminal reliability case[6] is shown in the second column and after Hardy *et al.* [23], it can be deduced from

$$\sum_{j=1}^{|F|} \left( A_{|F|,j} \sum_{k=0}^{\min(2,j)} \binom{j}{k} \right). \tag{3.10}$$

Table 3.1.: # Partitions according to $|F_k|$ and # blocks

| $\vert F\vert$ | **# Partitions consisting of $\vert 1\vert 2\vert \ldots \vert\vert F\vert\vert$ blocks** |
|---|---|
| ***2*** | $\vert 1\vert 1\vert$ |
| ***3*** | $\vert 1\vert 3\vert 1\vert$ |
| ***4*** | $\vert 1\vert 7\vert 6\vert 1\vert$ |
| ***5*** | $\vert 1\vert 15\vert 25\vert 10\vert 1\vert$ |
| ***6*** | $\vert 1\vert 31\vert 90\vert 65\vert 15\vert 1\vert$ |
| ***7*** | $\vert 1\vert 63\vert 301\vert 350\vert 140\vert 21\vert 1\vert$ |
| ***8*** | $\vert 1\vert 127\vert 966\vert 1,701\vert 1,050\vert 266\vert 28\vert 1\vert$ |
| ***9*** | $\vert 1\vert 255\vert 3,025\vert 7,770\vert 6,951\vert 2,646\vert 462\vert 36\vert 1\vert$ |
| ***10*** | $\vert 1\vert 511\vert 9,330\vert 34,105\vert 42,525\vert 22,827\vert 5,880\vert 750\vert 45\vert 1\vert$ |
| ***11*** | $\vert 1\vert 1,023\vert 28,501\vert 145,750\vert 246,730\vert 179,487\vert 63,987\vert 11,880\vert 1,155\vert 55\vert 1\vert$ |
| ***12*** | $\vert 1\vert 2,047\vert 86,526\vert 611,501\vert 1,379,400\vert 1,323,652\vert 627,396\vert 159,027\vert 22,275\vert 1705\vert 66\vert 1\vert$ |
| ***13*** | $\vert 1\vert 4,095\vert 261,625\vert 2,532,530\vert 7,508,501\vert 9,321,312\vert 5,715,424\vert 1,899,612\vert 359,502\vert 39,325\vert 2,431\vert 78\vert 1\vert$ |

Table 3.2.: Total # Partitions for two- and all-terminal case

| $\vert F\vert$ | $\vert K\vert = 2$ | $K = V$ |
|---|---|---|
| ***2*** | 6 | 2 |
| ***3*** | 18 | 5 |
| ***4*** | 58 | 15 |
| ***5*** | 206 | 52 |
| ***6*** | 810 | 203 |
| ***7*** | 3,506 | 877 |
| ***8*** | 16,558 | 4,140 |
| ***9*** | 84,586 | 21,147 |
| ***10*** | 463,898 | 115,975 |
| ***11*** | 2,714,278 | 678,570 |
| ***12*** | 16,854,386 | 4,213,597 |
| ***13*** | 110,577,746 | 27,644,437 |

## 3.5.3. Application example

In the following, we want to illustrate the workings of this approach by means of a small example. The two-terminal reliability with respect to the terminals $s$ and $t$ is computed for the 2x3-ladder network depicted on the right of Figure 3.5. First we conduct a bfs-traversal from $s$ to obtain a possible variable ordering: $e0 < e3 < e4 < e1 < e5 < e6 < e2$. Then $e0$ is taken for factorization. The first frontier set $F_1$ consists of the end nodes of $e0$. Factoring on edge $e0$, we obtain partition $[0\ 1]^*$ and $[0]^*[1]$ for $e0$ working and failed respectively (see Figure 3.5).[7] The blocks that contain terminal nodes are marked by an asterisk. The first/second partition has one/two block(s). In fact, there can be six different partitions:

[6]This number is larger than in the all-terminal case since we additionally have to differentiate between unmarked and marked blocks (blocks that contain terminal nodes).

[7]$[0\ 1]^*/[0]^*[1]$ is the high/low child of OBDD-root node $G$.

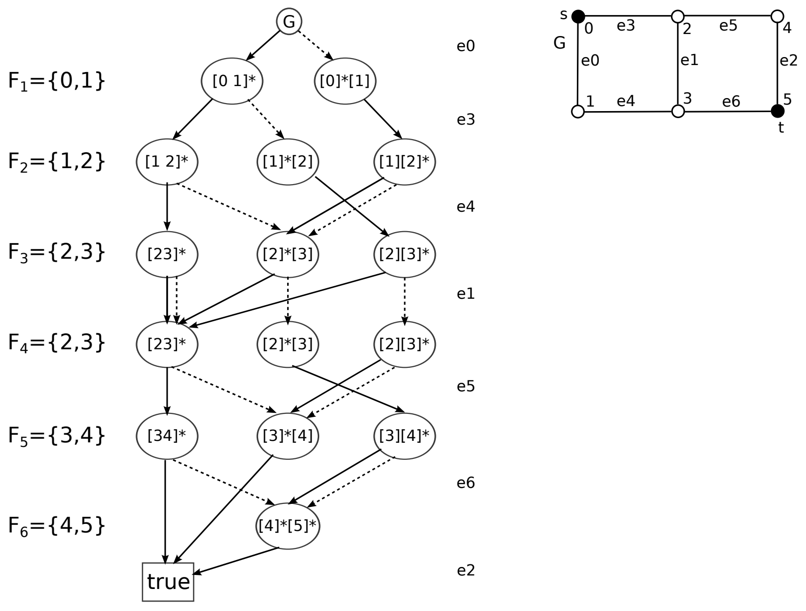


Figure 3.5.: OBDD for G, variable order: $e0 < e3 < e4 < e1 < e5 < e6 < e2$

$[0\ 1]$, $[0\ 1]^*$, $[0][1]$, $[0]^*[1]$, $[0][1]^*$, $[0]^*[1]^*$. In contrast to the factoring algorithm, the reliability subgraphs of $G$ are now represented by partitions. This comes along with great advantages with regard to space and time requirements. The total number of partitions in the k-th level depends on the cardinality of $F_k$. In particular, it grows exponentially with the size of $F_k$ (see [23]). Any partition that has no marked blocks is failed which means that at least one of the terminals is disconnected from the frontier set: in the example, the low child of $[0]^*[1]$ is $[1][2]$ and hence false.
Recognizing **reliability isomorphic subgraphs**[8] is further simplified by representing reliability subgraphs as partitions: two subgraphs $G_1$ and $G_2$ are reliability isomorphic if their associated partitions are identical with regard to the same edge ordering. In the fourth level of the depicted OBDD, partition $[2\ 3]^*$ is recognized three times. This recognition avoids redundant computations and contributes to a higher efficiency of the algorithm. If all terminals are connected in one block of a partition, a working configuration is found: the high child of $[4]^*[5]^*$ is $[4\ 5]^*$ and hence we have to link to the true-leaf. In total, the result OBDD consist of 16 nodes and only six partitions are held in memory at a time. By dropping the four redundant nodes which have the same low and high child, the size of this OBDD shrinks to 12. Since the representation of a Boolean function in terms of a ROBDD is canonical (see canonicity Lemma 2.6) with regard to a certain variable ordering, one can verify that we obtain the same ROBDD as on the right hand side of Figure 3.3 after applying OBDD reductions. The maximal width of the OBDD depends on $|F_{max}|$. Here, $|F_{max}| = 2$ and the maximal width equals three which also remains constant for larger sizes of this ladder type. So we can expect that the memory consumption is restricted to grow with adherence to the depth of the OBDD. Additionally, the runtime increases lin-

[8] In general, two subgraphs are reliability equivalent if they represent the same Boolean function w.r.t. a stipulated reliability measure. Thus, the reliability isomorphic property captures a subset of reliability equivalent graphs.

early with the size of the ladder length. The decomposition approach performs well for networks having regular structure. More precisely, its performance highly depends on the size of the maximal boundary set $F_{max}$ (see Section 3.5.5).

### 3.5.4. Details of the decomposition approach

The decomposition approach is given as pseudo code in Procedure 4. According to Hardy *et al.* [23], the partitions can be uniquely represented by a natural number. However, we have omitted the partition-number-transforming functions ($Part \rightarrow Number$ and $Number \rightarrow Part$ [23]) since the overhead for transforming partitions into numbers and numbers to partitions becomes a serious issue for large $|F_k|$ [9]. Instead, the partitions are directly hashed according to the conventions proposed by Hermann [25]: for example, partition $[1\ 2][3]^*[4]^*$ can be hashed by two vectors. The first vector $part = [1, 1, 2, 3]$ states that nodes $1, 2$ are in the first block, node $3$ in the second and node $4$ in the third block. The second vector consists of Boolean entries $b = [0, 1, 1]$ recording the block marking. Naturally, the size of $b$ equals the number of blocks.
So each partition node $p_k$ is attached with an integer vector and a Boolean vector. In addition to that, the appropriate BDD node which can be retrieved by the function $getbdd()$, is stored. After having determined an ordering by bfs, the initial step in Procedure 4 is to link the root node of the BDD to two new child nodes $p_0$ and $p_1$. They are put into the list $prevLevel$ in order to be further processed. The BDD emerges level by level whereas at each level $k$ boundary set $F_k$ has to be updated. The workloads or partition nodes are processed until $prevLevel$ is empty. The number of workloads held in $prevLevel$ corresponds to the width of level $k$ in the BDD. By processing the partition nodes from $prevLevel$, new partition nodes are created and added to the $nextLevel$. At most, there are two levels of partition nodes held in memory at once.
We have found that Hardy's approach lacks the possibility of creating a BDD false-leaf when merging an edge during factoring. This fact was also revealed by [25] and is considered in lines 12-14 of Procedure 4. In Section 6.1.1 we give an example, where this case might occur.
The derivation of new partitions $part1$ by merging and $part0$ by deleting the edge variable from $p_k$, is not an obvious task. Hence, we refer to Section 6.1 for details.

### 3.5.5. Runtime complexity

The complexity for the two-terminal reliability equals $\mathcal{O}\left(|E| \cdot |F_{max}|^3 \cdot B_{|F_{max}|}\right)$. Notably, the complexity is $\mathcal{O}\left(|E| \cdot |F_{max}| \cdot B_{|F_{max}|}\right)$ and $\mathcal{O}\left(|E| \cdot |F_{max}| \cdot B_{|F_{max}|} \cdot 2^{|F_{max}|}\right)$ for the all-terminal and k-terminal case respectively (see [23]). So this approach allows for computing the reliability of networks with bounded $|F_{max}|$ in time linear to the number of edges. The experimental results from the literature show that for regular network structures with small and bounded $F_{max}$, the decomposition

[9]According to Hermann [25], the disadvantage of using partition numbering is that such numbers quickly grow to be larger than can be stored in the standard data types of programming languages. This requires the use of adequate libraries for dealing with numbers of arbitrary size and greatly slows down the processing speed of the approach. In addition, he also claimed the unnecessary overhead for transforming the partitions to numbers and backwards.

Algorithm 4: $BDDPartition$

```
Require: Graph $G = (V, E)$, set of terminal nodes $K$
 1: Determine an edge ordering by bfs heuristic
 2: Initialize $F_1 = \{u, v\}$ with $e = (u, v)$ being the first edge
    in the variable order, initialize root node $BDD_{root}$
 3: $root$.high = $p_1.getbdd()$, $root$.low = $p_0.getbdd()$
 4: Add the first two partitions $p_0, p_1$ to $prevLevel$
 5: for $k = 2$ to $|E|$ do
 6:   compute $F_k$
 7:   while $prevLevel \neq \emptyset$ do
 8:     $p_k = prevLevel.pop()$
 9:     Derive $part1$ from $p_k$
10:     if all K-vertices are merged into the same block in $part1$ then
11:       $p_k.getbdd()$.high = true
12:     else if one K-vertex is disconnected in $part1$ then
13:       //This case is missing in Hardy's approach
14:       $p_k.getbdd()$.high = false
15:     else
16:       if $part1$ is not in hash table then
17:         $nextLevel.add(part1)$
18:         insert $part1$ in hash table
19:       end if
20:       $p_k.getbdd()$.high = $part1.getbdd()$
21:     end if
22:     Derive $part0$ from $p_k$
23:     if one K-vertex is disconnected in $part0$ then
24:       $p_k.getbdd()$.low = false
25:     else
26:       if $part0$ is not in hash table then
27:         $nextLevel.add(part0)$
28:         insert $part0$ in hash table
29:       end if
30:       $p_k.getbdd()$.low = $part0.getbdd()$
31:     end if
32:   end while
33:   $prevLevel = nextLevel$, $nextLevel.clear()$
34: end for
35: return $root$
```

approach has the edge over the KLY-approach (see [23, 25]). However, for networks with unregular structure such as randomized networks, a fair comparison was not provided. In Section 8.2 we will show that the decomposition method is not in general more efficient (with regard to runtime and memory consumption) than the KLY-approach, especially for unregular network structures.

### 3.5.6. Sorting of blocks

Hardy's approach lacks the description of distinctly arranging the blocks within the partitions. This is vital for correctly assigning and identifying the block numbers. Hence, we briefly propose a distinct arrangement scheme at this point: first, the blocks are sorted with respect to their cardinality. Subsequently, the blocks with the same cardinality are sorted according to the lowest vertex number. Since the vertex numbers are unique and appear only once in some block, we obtain an unambiguous block arrangement convention. Suppose, we have the following partition $p = [5][3\ 4][0\ 1\ 2][2\ 8][6]$, then after sorting we obtain $p = [5][6][2\ 8][3\ 4][0\ 1\ 2]$.

### 3.5.7. Considering node failures

According to Hermann [26], imperfect nodes can be considered by factoring on each node in turn, and immediately afterwards factoring on all edges adjacent to the just factored nodes. In this manner, the variable ordering for the example from Figure 3.5 would be $0 < e_0 < e_3 < 1 < e_4 < 2 < e_1 < e_5 < 3 < e_6 < 4 < e_2 < 5$. Then one constructs the result BDD according to this ordering. The time complexity for the two-terminal reliability problem then increases to $\mathcal{O}\left((|V| + |E|) \cdot |F_{max}|^3 \cdot B_{|F_{max}|}\right)$[10].

[10]See [26] for a more detailed explanation.

# 4. Bounds on Network Reliability

The computation of network reliability is in general a difficult task which is known to be #P-Complete. In order to be of practical use, many efforts have been devoted to devise algorithms which deliver accurate bounds in polynomial time. Two main reliability bounding directions can be found in the literature. The first one uses simulation which delivers an approximate value and a confidence interval [20]. However, the simulations cannot provide absolute certainty that the actual reliability value will fall into this interval. The second direction applies analytic methods to obtain lower and upper bounds. In this case it is guaranteed that the actual reliability value will fall between the bounds. In this chapter we outline some established methods for the second direction. We distinguish between models which allow arbitrary component failure probabilities and equal component failure probabilities. The only drawback of polynomial time bounding algorithms is their lack of accuracy: for certain networks, lower and upper bounds are so far apart that they turn out to be worthless. As an alternative, we illustrate a state-of-the-art exact terminal-pair approach (the Dotson-Gobien method [18]) which has exponential complexity but gives fast-converging bounds.

## 4.1. Bounds for networks with equal edge failure probabilities

The *reliability polynomial* $R_K^p$ (see Equation 2.10) forms the starting point for deducing bounds. With regard to equal edge operation probability, $p$, polynomial time bounding algorithms were mainly devoloped for the all-terminal reliability ($K = V$). Two effective strategies were considered for achieving good bounds. The first one uses enumeration techniques for counting operational subnetworks and the second one exploits graph-theoretical properties of the input network. Starting with the first one, we look at the reliability polynomial

$$R_V^p = \sum_{i=1}^{q} N_i p^i (1-p)^{q-i},$$

For the all-terminal reliability problem, spanning trees need to be counted. According to [24], at least $|V| - 1$ edges are needed to connect all nodes $V$. Hence for $i < |V| - 1$, $N_i = 0$ and $N_{|V|-1}$ is the number of spanning trees which can be efficiently computed [19]. Furthermore $N_i = \binom{q}{i}$, for $i > q - c$, and $N_{q-c} = \binom{q}{c} - n_c$ (see Section 2.4.2), where the cardinality of a minimal cardinality cutset $c$ and the number of minimal cardinality cutsets $n_c$ can be efficiently determined by [8]. Unfortunately, no efficient algorithms are known for computing $N_i$, $|V| \le i < q-c$. Thus, efforts were made to find lower and upper bounds for each of those $N_i$.

### 4.1.1. Simple bounds

Using the trivial bounds $0 \leq N_i \leq \binom{q}{i}$ leads to the first set of bounds, the *Jacobs bounds* [31] which were improved by Van Slyke and Frank [67], who expressed them in the following form:

$$\begin{aligned} R_V^p &\leq N_{|V|-1} p^{|V|-1}(1-p)^{q-|V|+1} + \sum_{i=|V|}^{q} \binom{q}{i} p^i (1-p)^{q-i} \text{ and} \\ R_V^p &\geq N_{|V|-1} p^{|V|-1}(1-p)^{q-|V|+1} + N_{q-c} p^{q-c}(1-p)^c + \sum_{i=q-c+1}^{q} \binom{q}{i} p^i (1-p)^{q-i} \end{aligned}$$

Each unknown $N_i$ is approximated by zero in the lower bound and by $\binom{q}{i}$ in the upper bound.
The parameter $p$ has a crucial effect on the reliability value: Kel'mans [34] found out that for two different networks $G_1$ and $G_2$, one can find certain values for $p$ where $R_V^p(G_1) < R_V^p(G_2)$, or where $R_V^p(G_1) > R_V^p(G_2)$. Thus, reliability depends on both the random failure probabilities and the inherent graph structures. Kel'mans further established the following bounds for all $p$:

$$N_{|V|-1} p^{|V|-1}(1-p)^{q-|V|+1} \leq R_V^p \leq 1 - n_c p^{q-c}(1-p)^c.$$

Furthermore he observed that $R_V^p$ tends more to the lower bound for very small $p$ and to the upper bound for very large $p$.
To obtain two-terminal bounds, we only need to substitute $|V|-1$ with $l$ (cardinality of a minimal cardinality pathset). These bounds are useful only if $p$ is very near zero or one. In general, they are extremely weak bounds [24].

### 4.1.2. Ball-Provan bounds

In order to find a remedy to the inaccuracy of these bounds, Ball and Provan developed bounds based on the second representation of the reliability polynomial (see Equation 2.11):

$$R_V^p = \sum_{i=0}^{q} F_i p^{q-i} (1-p)^i.$$

Factoring out the nonzero terms $p^{|V|-1}$ and rewriting [15], yields:

$$R_V^p = p^{|V|-1} \sum_{i=0}^{q} H_i (1-p)^i, \text{ where } H_i = \sum_{k=0}^{i} (-1)^{i-k} \binom{q-|V|+1-k}{i-k} F_k.$$

According to [15], Ball and Provan then use a theorem of Stanley to obtain lower and upper bounds on the coefficients $H_i$ which in turn bound the $F_i$. They also conceived a polynomial time implementation of these bounds in [9]. The importance of these bounds is that they embody linear constraints on the $F_i$ with $F_i \geq \underline{F_i}$ and $F_i \leq \overline{F_i}$. By bounding the coefficients in this way, we obtain the *Ball-Provan bounds* for the all-terminal reliability:

$$\sum_{i=0}^{q} \underline{F_i} p^{q-i}(1-p)^i \leq R_V^p \leq \sum_{i=0}^{q} \overline{F_i} p^{q-i}(1-p)^i. \tag{4.1}$$

### 4.1.3. Lomonosov-Polesskii bounds

Unlike Ball and Provan, Polesskii [55] used a completely different strategy. He developed lower bounds based on a general graph theoretical result from Tutte [75] and Nash-Williams [53] which states that each graph has at least $\lfloor \frac{c}{2} \rfloor$[1] edge-disjoint spanning trees. Additionally, Polesskii noted that the network failure probability cannot be larger than the product of probabilities that each member of any set of edge-disjoint spanning trees fails. So each of the edge disjoint spanning trees must fail before the network fails. From those facts Polesskii deduced the lower bound

$$R_V^p \geq 1 - (1 - p^{|V|-1})^{\lfloor \frac{c}{2} \rfloor} \tag{4.2}$$

which was further improved by Lomonosov and Polesskii [46] to obtain the lower all-terminal reliability bound

$$R_V^p \geq |V| \left(1 - (1-p)^{\lfloor \frac{c}{2} \rfloor}\right)^{|V|-1} - (|V|-1) \left(1 - (1-p)^{\lfloor \frac{c}{2} \rfloor}\right)^{|V|}. \tag{4.3}$$

Lomonosov and Polesskii [47] also developed an upper bound. For that purpose, they defined a cut basis of a network graph as a set of $|V|-1$ minimal edge cutsets $\{L_1, L_2, \dots, L_{|V|-1}\}$ which means that every two nodes of the graph are disconnected by one of the edge cutsets $L_k,\ k \in \{1, 2, \dots, |V|-1\}$. After Gomory and Hu [22], a cut basis of a graph can be found in polynomial time. For the network graph to operate, at least one of the edges in every cut $L_k$ must be working.
Since $\mathbb{P}(\text{"}\textit{At least one edge from } L_k \textit{ is working}\text{"}) = 1 - \mathbb{P}(\text{"}\textit{All edges from } L_k \textit{ are failed}\text{"})$, we obtain the following upper bound

$$R_V^p \leq \prod_{k=1}^{|V|-1} \left(1 - (1-p)^{|L_k|}\right). \tag{4.4}$$

Although the construction of the Lomonosov-Polesskii bounds seems to be less sophisticated than this of the Ball-Provan bounds, the Lomonosov-Polesskii bounds however occasionally improve on the Ball-Provan bounds. In fact, for varying network samples and $p$-values these two different bounds improve on each other [15]. Colbourn and Harms [15] further combined the Lomonosov-Polesskii and Ball-Provan bounds by linear programming in order to obtain some improvements on the reliability bounds.

## 4.2. Bounds for networks with arbitrary edge failure probabilities

The reliability polynomial is no longer applicable when edges operate with different probabilities. In order to obtain bounds, a major technique called *edge packing* was followed.

### 4.2.1. Edge-packing

Given a graph $G = (V, E)$, $G$ is partitioned into $k$ partial graphs[2] $G_1 = (V, E_1), \dots, G_k = (V, E_k)$, where $E_1, \dots, E_k$ are pairwise disjoint. $G_i$ operates independently from $G_j$ for

[1] $c :=$ cardinality of a minimal cardinality cutset.
[2] A partial graph of $G := (V, E)$ is a graph $G' := (V, E')$ with $E' \subseteq E$.

$i \neq j$. Since there are operational states of $G$ where no $G_i$ is operational, we obtain the following lower bound:

$$R_K(G) \geq 1 - \prod_{i=1}^{k} (1 - R_K(G_i)). \tag{4.5}$$

For obtaining all-terminal bounds, Polesskii [55] suggested to edge-pack G with *minpaths*. In this case a maximal cardinality set of edge-disjoint spanning trees can be efficiently found by Edmonds's matroid partition algorithm [19]. The result for Equation 4.2 can then be similarly extended for arbitrary edge failures in order to obtain lower bounds. Edmonds's matroid partition algorithm does not necessarily deliver the best edge-packing bound by minpaths. In fact, the complexity of finding the best edge packing for all-terminal case remains open.
Brecht and Colbourn [71] established edge-packing lower bounds for the two-terminal reliability. Since Menger's theorem states that the maximum number of edge-disjoint *s*-*t*-paths equals the cardinality of a minimum *s*-*t*-cut, flow techniques [38] are used to find a maximum edge packing. However, finding the best edge-packing by *s*-*t*-paths is NP-hard (see Raman [59]). The situation looks similar for the k-terminal problem, since determining the maximum number of Steiner trees in an edge-packing is NP-hard [14]. Unfortunately, no good heuristics are known for finding a good edge packing for the k-terminal case.
Turning to upper bounds, $G$ can be edge-packed by cutsets $C_1, \ldots, C_k$. The failure of $G$ is caused by a failure of any cutset and since $G$ can fail, even if no cutset in the packing is failed, the following inequality holds:

$$R_K(G) \leq \prod_{i=1}^{k} \left(1 - \prod_{e \in C_i} (1 - p_e)\right). \tag{4.6}$$

For bounding the two-terminal reliability, a maximal number of edge-disjoint *s*-*t*-cutsets must be found. This is at the same time the dual to Menger's theorem (The maximal number of edge-disjoint *s*-*t*-cuts equals the length of a shortest *s*-*t*-path) and can be performed efficiently in polynomial time [78]. However, for the all- and k-terminal case, finding a maximum packing by mincuts is NP-hard [14]. Colbourn [14] compared the edge-packing strategy with the Lomonosov-Polesskii and Ball-Provan bounds for the two- and all-terminal case with equal edge failure. He found that for the two-terminal case, edge packing clearly affords better results. However, turning to the all-terminal case, the edge packing bound is very poor in comparison to the other two stategies.

## 4.3. The Dotson-Gobien algorithm

The results above reveal that the polynomial time bounding methods still do not yield bounds which satisfy the prescribed accuracy in the field of reliability engineering - it is quite common to assume a relative accuracy in the order of magnitude of $10^{-1}$. According to Ball [7], finding reliability bounds is a time/accuracy trade-off and hence it is very unlikely that such polynomial time algorithms exist. To achieve more accurate bounds, we have to resort to approaches with exponential complexity. In this section, we present the workings of the DG (Dotson-Gobien) method [18]. It exactly computes the two-terminal reliability and in case of premature interruption, we obtain at least some lower and upper bounds. The efficiency of this method was empirically shown in a comparison with three other algorithms by Yoo and Deo [86]. The DG is based on set theoretical partition of the sample space $\{0,1\}^q$ into disjoint sets - in case of equal edge failure probabilities, the sets obtained in the first partitioning steps are the most probable ones. Yoo and Deo also empirically found that the number of subproblems processed is only a small fraction of the total number of subproblems possible. In addition to that, the DG bounds converge fast, since the most probable path- and cutsets appear at the beginning - assuming that all edges fail with equal probability. Hence, in practice we can obtain good bounds after a fraction of the required computation time. At this point we give two different ways to derive the DG approach: the first way is set-theoretically based and the second is based on the factoring method.

### 4.3.1. Derivation

Dotson and Gobien defined an *elementary event* to be a state vector $\vec{x} \in \{0,1\}^q$ which indicates the state of each system component. A *full event* is recursively defined as either an elementary event or the union of two events that differ only in the status of exactly one system component. So a full event is composed of a union of elementary events. The intersection of two full events $A$ and $B$ equals the set of elementary events that are both in $A$ or in $B$.
Let $P$ be a path $P = [e_1, e_2, \dots, e_r]$ of length $r$ ($r \in \mathbb{N}$) in a network graph $G$ from $s$ to $t$ and $e_i \in E$ for $1 \leq i \leq r$. $P$ is a full event and can be expressed as the intersection of full events $e_i$ (component $c_i \in \mathcal{C}$ is working), where $e_i$ is itself a union of $2^{q-1}$ elementary events

$$P = \bigcap_{i=1}^{r} e_i.$$

The complement of $P$ is obtained by De Morgan's law

$$\overline{P} = \bigcup_{i=1}^{r} \overline{e_i}.$$

By applying the SDP equation (Equation 3.3), the complement of $P$ can be expressed as a union of *mutually disjoint* sets or full events

$$\overline{P} = \{\overline{e_1}\} \cup \{e_1 \cap \overline{e_2}\} \cup \ldots \cup \{e_1 \cap e_2 \cap e_3 \ldots \cap \overline{e_r}\}. \tag{4.7}$$

Each of the $r$ subproblems can be further partitioned in the same way by searching for *s-t-paths*. For each subproblem we have to look for the *topologically shortest* path to keep the number of subproblems low. This is done by breadth-first-search since all edges have length one.
Alternatively, a similar representation of Equation 4.7 can be derived by recursively applying the factoring theorem for each of the $r$ edges in path $P$ (see Deo and Medidi [17]). Choosing the first keystone edge $e_1 \in P$ and define the failure probability of component $e_i$ to be $q_i := 1 - p_i,\ i = 1 \leq i \leq r$, we have:

$$R_{s,t}(G) = p_1 \cdot R_{s,t}(G * e_1) + q_1 \cdot R_{s,t}(G - e_1),$$

where $*/-$ stands for a contraction/deletion of all edges $e_1$. Then the term $R_{s,t}(G * e_1)$ will again be expanded by factoring on edges $e_2$ which is analogously defined. Overall it follows that:

$$\begin{aligned} R_{s,t}(G) &= q_1 \cdot R_{s,t}(G - e_1) \\ &+ p_1 q_2 \cdot R_{s,t}(G * e_1 - e_2) \\ &+ \dots \\ &+ p_1 p_2 \cdots p_{r-1} q_r \cdot R_{s,t}(G * e_1 * e_2 * \dots * e_{r-1} - e_r) \\ &+ \prod_{k=1}^{r} p_k \end{aligned} \tag{4.8}$$

So we have $r$ subproblems or subgraphs deduced from path $P$.
Again, for each subproblem this equation can be recursively applied. In each subgraph, reliability preserving reductions (Section 3.2) can be performed if possible [17]. The above Equation 4.8 can be illustrated by a recursion tree, where the tree nodes correspond to the subproblems and the tree edges are labeled with the edges to be contracted or deleted.
Suppose $\mathcal{S}$ to be a *disjoint exhaustive success collection* of success events $S_i,\ 1 \leq i \leq |\mathcal{S}|$ (i-th $s$-$t$ connecting subgraph) such that if an elementary event connects $s$ and $t$, then it is contained in some $S_i \in \mathcal{S}$. The terminal-pair reliability of G is then represented by

$$R_{s,t}(G) = \sum_{i=1}^{|\mathcal{S}|} \mathbb{P}(S_i).$$

The last addend of equation 4.8 is the probability of the first success event $S_1$, thus $\mathbb{P}(S_1) = \prod_{k=1}^{r} p_k$. All the other $\mathbb{P}(S_i),\ 2 \leq i \leq |\mathcal{S}|$ are the probabilities of the remaining $s$-$t$ paths found in the subgraphs multiplied with their appropriate probability terms. Analogously it holds for the *disjoint exhaustive failure collection* $\mathcal{C} := \{C_i,\ 1 \leq i \leq |\mathcal{C}|\}$

$$R_{s,t}(G) = 1 - \sum_{i=1}^{|\mathcal{C}|} \mathbb{P}(C_i),$$

where the $\mathbb{P}(C_i)$ terms are the probabilities for the $s$-$t$ cuts. For $u < |\mathcal{S}|$, $v < |\mathcal{C}|$ and $u, v \in \mathbb{N}$ the lower and upper bounds for the reliability are

$$\sum_{i=1}^{u} \mathbb{P}(S_i) \leq R_{s,t}(G) \leq 1 - \sum_{i=1}^{v} \mathbb{P}(C_i).$$

Following this inequation, the lower bound increases everytime a new $s$-$t$ path has been found and the upper bound respectively decreases for every additional $s$-$t$ cut. It becomes an equation as soon as all $s$-$t$ cuts and paths are found.

### 4.3.2. The approach

At this point, we give a brief description of the DG method with reductions. First we try to reduce the input graph as far as possible. Then we look for the shortest $s$-$t$ path of length

$l$. In case no $s$-$t$ path can be found, we subtract the probability of the cut[3] from the current upper bound and return from the recursion afterwards. The recursion also ends, when $s$ and $t$ are merged into one single vertex. Otherwise, the found $s$-$t$ path contributes to the lower bound and we can further derive $l$ subproblems by factoring on the edges - as shown above - contained in the shortest $s$-$t$ path. Each of the subproblems is then again processed as described.

### 4.3.3. Example

To illustrate the workings of the DG method in conjunction with series-parallel reductions, we performed this calculation on the 2x3 grid network (see Figure 3.5).
We start by trying to reduce the grid network $G$: two series reduction are conducted on edges $e5,\ e2$ and $e0,\ e4$ resulting in edges $e_{r_1}$ and $e_{r_2}$ respectively. Since no further series or parallel reductions can be performed, we look for a shortest $s$-$t$ path. One shortest $s$-$t$ path is for example $e_3,\ e_{r_1}$. As a result, we obtain two subproblems: $G_1 := G - e_3$ and $G_2 := G * e_3 - e_{r_1}$. The lower bound obtained after the first iteration equals $p_{e_3}p_{e_{r_1}}$ and the upper bound equals one, since there are no cuts. The computation ends after series and parallel reductions have been performed on the two subgraphs $G_1$ and $G_2$. Assume that the two subgraphs have been reduced to one single edge $e_{r_3}$ and $e_{r_4}$ respectively. The resulting two-terminal reliability is then expressed as

$$R_{s,t} = p_{e_3}p_{e_{r_1}} + q_{e_3}p_{e_{r_3}} + p_{e_3}q_{e_{r_1}}p_{e_{r_4}}.$$

We note that in the course of the DG method, subgraphs with redundant edges may occur. These edges do not need to be removed, as we look for the topologically shortest $s$-$t$ path at each recursive function call, and hence redundant edges cannot lie on the shortest path. So we would never factor on an irrelevant or redundant edge.

### 4.3.4. Required data structures

In the original DG approach [18], the subgraphs are iteratively constructed with the help of an event queue. The event queue contains the respective sequences of edges after which the original graph is partitioned. Hence, the size of the accumulated event queue depends on the number of upcoming subproblems. To reduce their number, Deo and Medidi [17] incorporated reductions and instead of event queues, they directly store the emerging subgraphs as adjacency matrices. In either case, there is an overdemand for memory: the event queues contain redundant information by sharing the same sequence of precedent edges. Furthermore, storing all subgraphs can lead to a dramatic increase in memory consumption which could be expected as this algorithm is of exponential complexity. In other words, the accuracy of the computed bounds is restricted by the size of the available memory. Hence, in Chapter 7 we will address the task of optimally storing the essential information without significantly deteriorating the computation time.

[3]The cut-term obtained at a recursion step is a sequence of edges to be contracted or removed. This sequence is composed of edges that were contracted or removed in the previous recursion steps. Thus, following the path upwards to the root of the recursion tree one obtains the sought sequence.

### 4.3.5. Considering node failures

Torrieri developed a method that accounts for node failures [74]. This method can be embedded in the DG approach or any approach that generates symbolic path or cut terms. The additional cost for considering node failures is only linear in the number of edges. First an undirected network with unreliable nodes is transformed into a directed network with perfect nodes by replacing each undirected edge with two anti-parallel directed edges. Then the DG approach is performed on the transformed network with perfect nodes to yield cut and path terms. Each of those terms are appropriately manipulated during the computation. For detailed information, we refer to [74].
Unfortunately, Torrieri's method yields incorrect results for some undirected networks. However, this error has been identified and rectified by Chen et.al. [13]: instead of assigning the same label to two anti-parallel edges, they must be unambiguously assigned to distinct labels.

# Part III.

# Improvements

# 5. Improving the Kuo-Lu-Yeh Approach

One of the key ideas making the KLY method so efficient is the fast identification of reliability-isomorphic subgraphs. However, the details for uniquely representing and recognising the reliability graphs are concealed. In the following, we will explain our new data structure for uniquely describing the graph model. From this we can derive the needed criteria to unambiguously identify reliability isomorphic subgraphs. Furthermore, we show how the KLY method can be significantly improved by removing redundant biconnected components.

### Representation of the stochastic graph

The initial network graph is stored as a Boolean adjacency matrix $M$, with number of columns/rows equal to the number of nodes. Each edge in the initial graph is a Boolean variable indexed by a unique $i \in \mathbb{N}$. A pair of nodes is unambiguously assigned to an edge in a global hashmap *n1n2edge*. We exploit the symmetric property of $M$, representing an undirected graph with $N$ nodes, by only storing the entries of the lower or upper triangle part in a Boolean vector $v_{Bool}$ of length $\frac{N(N-1)}{2}$. $v_{Bool}$ has full length $N^2$ for directed graphs. Initially we set the node index equal to the column/row index. This index mapping is captured by an array $col2n$ (column index $\rightarrow$ node index) from which the reverse mapping $n2col$ can be deduced. In the course of the algorithm, nodes and their adjacent edges are deleted so that according to their deletion the mapping has to be adapted as follows: Assuming the node with index $i \in \mathbb{N}$ is deleted, column and row of $M$ with index $j := n2col[i]$ are therefore removed. $col2n$ must be reshifted according to the following rule: $\forall k \geq i : col2n[k] = col2n[k+1]$. Thereupon the last entry of $col2n$ is deleted and $n2col$ respectively adapted.

### Recognition of reliability isomorphic graphs

Two subgraphs $G$ and $G'$ are reliability isomorphic with respect to the two-terminal measure if:

- the position of terminal node $s$ is equal ($t$ remains fixed) and
- $v_{Bool}$ is equal and
- $col2n$ is equal.

These three criteria serve as key values for the above mentioned global hashmap.

## 5.1. Recognition and deletion of redundant biconnected components

A biconnected undirected graph is a connected graph that is not broken into disconnected pieces by deleting any single vertex. This means that a biconnected graph has no articulation vertices. An articulation vertex is defined as a vertex whose removal splits a connected graph into two connected subgraphs (see [28]). This property prevents the graph's disconnection upon removal of any single edge and can be regarded as a two-fold redundancy. So at least two edges must be removed to disconnect the graph. As mentioned in Section 3.4, only redundant nodes unequal $s$, $t$ with degree one are removed in the KLY algorithm. By recognising and removing redundant biconnected components, the size of subgraphs can shrink immensely. In summary: less time is needed for hashing the subgraphs. In addition to that, fewer subproblems, $G_{node}$, will evolve leading to a lower memory consumption since there will be fewer entries for the global hashmap. Nevertheless, this requires the occurence of such redundant structures. In case they do not occur, the overhead spent is only linear in the size of the graph [28]. In our experiments we will show that these redundant structures often occur for many network examples and that their deletion generally leads to a lower memory consumption and for large graphs, depending on their structure, to a noticeable speedup. In the following section we will illustrate the removal of redundant biconnected components with respect to the terminal nodes.

### 5.1.1. Transformation to the biconnected component tree

Figure 5.1 shows an example reliability graph $G$ having nine biconnected components [1] and five articulation vertices ($a1 - a5$). With Hopcroft's and Tarjan's algorithm implemented in function $articpointDFS(vertex)$, these components can be found in linear time by traversing $G$ in depth-first-search order. For $articpointDFS(s)$ we obtain for each articulation vertex $a$ its appropriate biconnected components $c$ returned as $bcmap$ (in Algorithm 5): $(a1 : c1, c2)$, $(a2 : c3)$, $(a3 : c4, c5, c6)$, $(a4 : c7)$, $(a5 : c9, c8)$. The assignment of components to their articulation node depends on the order the articulation nodes are visited. Components which were already assigned to an articulation node are no longer assigned to another articulation node which shares the same biconnected component. In $c_i, 1 \leq i \leq 9$, all the corresponding nodes of $G$ are stored. In order to identify and delete the redundant components, we need to transform $G$ to a biconnected component tree ($bctree$), $T_G$ , created by Algorithm 6. At line 1, $bctree$ is initialised with the number of nodes it will consist of. The source node and each of the biconnected components are nodes of the $bctree$. Starting with the source node (index set to 0), edge connections are built to its adjacent components. In each of the components we search for nodes which are articulation points. If there exists an articulation point $a$ in component $c$ then edges are bridged from node $c$ to all components belonging to $a$. The tree is hence established in depth-first-search manner by Algorithm 7. After the $bctree$ has been constructed from the original graph and the location of terminal $t$ has been determined in the tree (line 5-6 of Algorithm 5), we remove redundant nodes - unequal $s, t$ and with degree one - from the $bctree$. The redundant components are collected in list $comps2delete$ and finally all nodes belonging to

[1]A single edge is regarded as biconnected.

the collected components are deleted from the original graph $G$. Graph $G$ is disposed of 6 components (see bottom of Figure 5.1).

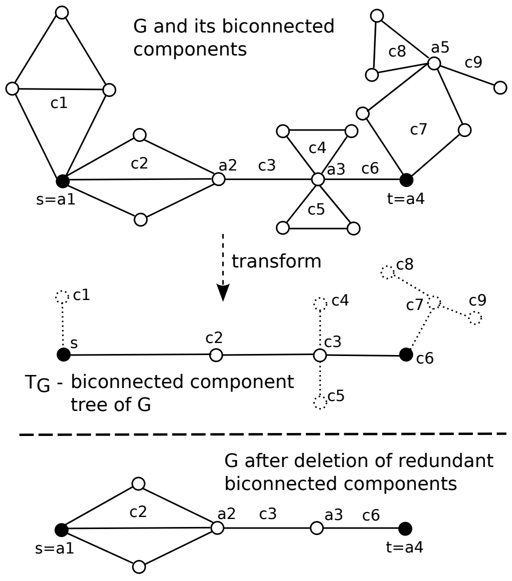

G and its biconnected components
c1
c2
s=a1
a2
c3
c4
a3
c5
c6
t=a4
c7
c8
a5
c9
transform
c1
s
c2
c4
c3
c5
c6
c7
c8
c9
TG - biconnected component tree of G
G after deletion of redundant biconnected components
c2
s=a1
a2
c3
a3
c6
t=a4

Figure 5.1.: Deletion of redundant biconnected components.

### 5.1.2. A case study

The pseudo code for the path constructing function is listed in Algorithm 2. In comparison to [85], merely the line for removing redundant nodes, line 11 of Algorithm 2, must be substituted by $removeRedundantBicomp$ (Algorithm 5).
Since redundant nodes with degree one are also biconnected components, they are implicitly considered for removal. The KLY algorithm is applied with our proposed extension to the example graph G0 in Figure 5.2 which shows the obtained EED. There are eight $G_{node}$ which are numbered according to the order they have been visited in the course of the algorithm. Without the proposed extension we would have obtained sixteen $G_{node}$ (hashmap entries or subproblems). We have five hits at EED node G3 since there are six incoming edges. The removal of the biconnected components is indicated by dotted lines. As one can observe from this example, the removal of biconnected components leads to a dramatic decrease of the size of the subgraphs and at the same time the number of subproblems. The number of hits for isomorphic graphs are fewer compared to the original approach. However, the hit ratio - defined as $\frac{Hit}{|G_{node}|+Hit}$ - of the extended approach is 5 percent higher. Adhering to the breadth-first-search variable ordering, we obtain 30 nodes for the resulting OBDD. Finally, this OBDD is evaluated by the $Prob$ function [85] to obtain the reliability value. The results for G0 can be found in Table 8.5 Nw.1 for edges assumed to fail with probability of 0.1.

Algorithm 5: removeRedundantBicomp

```
Require: $currentsource$
1: Map $bcmap = articpointDFS[currentsource]$
2: if $noOfcomps == 1$ then
3:     return //graph is biconnected
4: end if
5: Graph $bctree = transform(bcmap, currentsource)$
6: Determine the location of terminal $t$ in the $bctree$
7: repeat
8:     $H := \{c \in bctree | c \notin \{s, t\} \wedge deg(c) == 1\}$
9:     for all node $c \in H$ do
10:        remove $c$ from $bctree$
11:        $comps2delete.add(c)$ //Collect $c$ in a list
12:    end for
13: until $H = \emptyset$
14: for all node $c \in comps2delete$ do
15:    for all node $n \in c$ do
16:        G.$removeNode(n)$
17:    end for
18: end for
```

Algorithm 6: transform

```
Require: $bcmap, currentsource$
1: Graph $bctree(noOfcomps + 1)$
2: Map $m = bcmap[currentsource]$
3: for all component $c \in m$ do
4:     $bctree.addEdge(0, c)$ //$currentsource := 0$
5:     List $ls = m[c]$ //set of all nodes in $c$
6:     for all node $n \in ls$ do
7:         if !$bcmap[n].empty()$ then
8:             dfs($bctree$, $bcmap$, $n$, $c$)
9:         end if
10:    end for
11: end for
12: return $bctree$
```

Algorithm 7: dfs

```
Require: $bctree, bcmap, articpoint, c_{previous}$
 1: Map $m = bcmap[articpoint]$
 2: for all component $c \in m$ do
 3:     $bctree$.addEdge($c_{previous}$, $c$)
 4:     List $ls = m[articpoint]$
 5:     for all node $n \in ls$ do
 6:         if !$bcmap[n]$.empty() then
 7:             dfs($bctree$, $n$, $c$)
 8:         end if
 9:     end for
10: end for
```

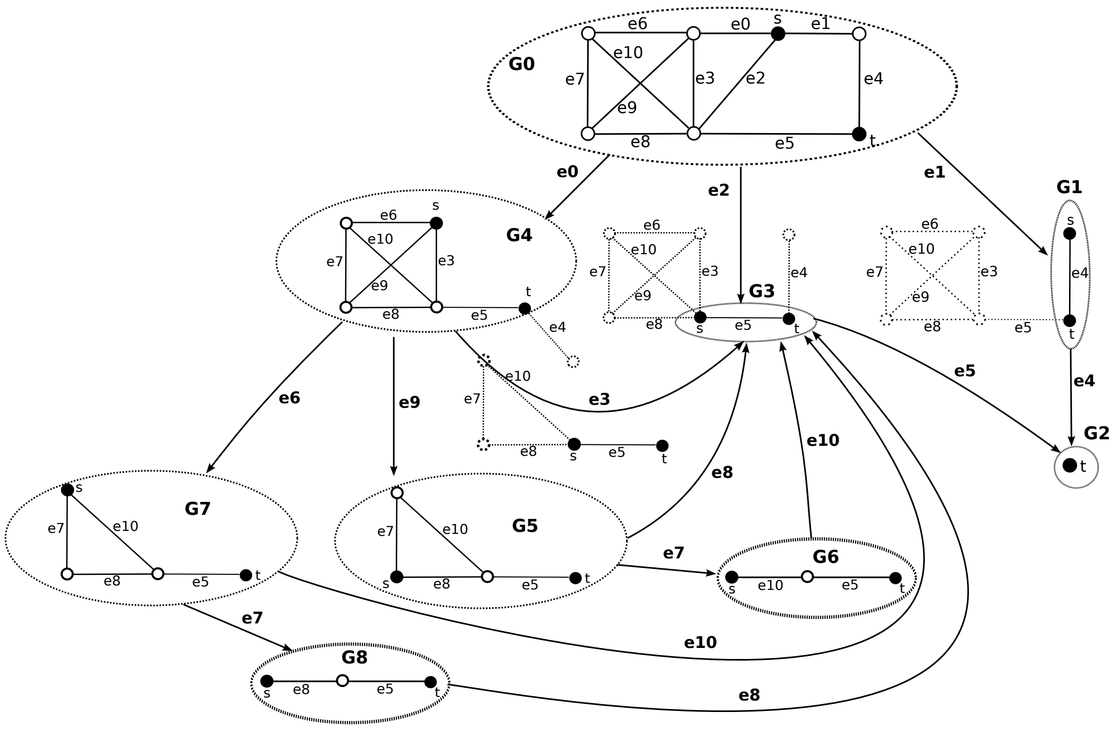


Figure 5.2.: Edge Expansion Diagram for Nw.1.

# 6. Improving the Decomposition Approach

We have outlined in Section 3.5 that the complexity of the decomposition approach notably depends on the size of the maximal boundary set $F_{max}$. Since $|F_{max}|$ results from the process of variable ordering, we must focus on conceiving variable ordering techniques which preferably lead to low $|F_{max}|$. Finding a graph decomposition which leads to the smallest $|F_{max}|$ corresponds to the path- or tree-width problem, also called partial $k$-tree problem: given a graph $G$, find a tree decomposition for $G$ of minimal tree-width.

**Definition 6.1.** *According to [60], a tree decomposition of a graph $G = (V, E)$ is a family $(X_i : i \in I)$ of subsets of $V$, together with a tree $T = (I, F)$ whose node set corresponds to $I$ and $F$ is the edge set of $T$, such that*

1. $\bigcup_{i \in I} X_i = V$,
2. $\forall (v, w) \in E, \exists i \in I$ *with* $v, w \in X_i$ $(i \in I)$,
3. $\forall i, j, k \in I$, *if $j$ is on the path in $T$ from $i$ to $k$ then* $X_i \cap X_k \subseteq X_j$.

*The tree-width of a tree decomposition $T$ is defined as $W(T) := \max_{i \in I} |X_i| - 1$. The tree-width is defined such that there is no more than $W(T)$ common vertices between $X_i$ and $X_j$, with $(i, j) \in F$, unless $X_i = X_j$. The tree-width of a graph, $TW(G)$, is the smallest tree-width over all its tree decompositions: $TW(G) = \min_{T \text{ is a tree decomposition of } G} W(T)$.*

The problem of finding a tree decomposition of smallest tree-width is motivated by the fact that once a tree decomposition for a graph $G$ with bounded tree-width $k \in \mathbb{N}$ has been determined, many NP-hard problems for general graphs $G$ can be solved in time linear to the size of $G$ but exponential in $k$ or $|F_{max}|$. Unfortunately, finding a tree decomposition with smallest tree-width is an NP-complete problem [4]. A lot of research has been done on finding tree-decompositions with minimal tree-width. A noteworthy result is due to Bodlaender [10]: for fixed $k$, he found a linear time algorithm which decides whether a graph $G$ has bounded tree-width of at most $k$, and if so, it finds a tree-decomposition of $G$ with tree-width at most $k$. However, the constant factor $k$ of the algorithm is very large - much too large for practical purposes. In addition, the algorithm in its present form is probably not practical.
Thus, Carlier, Hardy and Hermann [12, 23, 25] suggested the bfs-ordering as a good alternative variable ordering for the decomposition method. Apart from this, Lucet proposed in her PhD thesis [48] some other heuristics for obtaining low $|F_{max}|$ values. However, these heuristics have not yet been shown to deliver better results than the bfs heuristic. In this chapter, we propose another heuristic which yields lower $|F_{max}|$ values for a large variety of different graph structures. The new heuristic has a worst case complexity of $\mathcal{O}\big(|V| \cdot |E| \cdot |F_{max}| \cdot N_{max}\big)$, where $N(F_k) := \{v \notin F_k : (u, v) \in \overline{E_k},\ u \in F_k\}$[1] is the set of

[1] $\overline{E_k}$ is the complementary edge set in the k-th level, defined in Section 3.5.4.

nodes adjacent to frontier set $F_k$ and $N_{max} := \max_k |N(F_k)|$ is the maximal neighborhood among all frontier sets. In order to conceivably improve performance, it can also be applied to other BDD-based methods, such as the KLY method.
The experiments in Section 8.1 show that the new heuristic finds for many example networks - especially unregular networks - edge orderings with much lower $|F_{max}|$ than the currently used bfs-heuristic. To get a better understanding of the evolvement of the new heuristic, we first explain the derivation of new partitions during the decomposition procedure. This non-trivial derivation was omitted by [23]. As stated in Section 3.5, we provide an example where a false-leaf is created when an edge is merged during factoring. At the end of this chapter the new heuristic is compared to the current bfs-heuristic by means of three sample networks.

## 6.1. Derivation of new partitions

Based on an available partition, two new partitions are derived by edge-contraction or edge-deletion.
Let $e_1, e_2, \dots, e_q$ be an ordering of edges for a graph $G$. We say that this edge ordering has a connected property if for each $k \in \{1, 2, \dots, q\}$, the edge set $E_k$ is connected, whereas $E_k := \{e_1, e_2, \dots, e_{k-1}\}$ and $\overline{E}_k := \{e_k, \dots, e_q\}$. Since the bfs and the new heuristic have an edge ordering with connected property, we obtain at most one new node which is added to the previous boundary set $F_{k-1}$ at each factoring step. So we are confronted with two cases when factoring on edge $e := (n_1, n_2)$: Case I. where a new node $n_2 \in F_k$ is added to $F_{k-1}$ and Case II. where no new nodes are added to $F_{k-1}$. For Case I., there are two sub-cases and for case II. there are four sub-cases to distinguish. They are listed in Table 6.1. For example, Sub-case a) means that $n_1$ leaves the previous frontier set $F_{k-1}$ in the next iteration $k$.
The four sub-cases a) to d) lead to ten elementary cases for each contraction and deletion operation (see Tables 6.2, 6.3, 6.4, 6.5). For Tables 6.3, 6.5 we have omitted Sub-case d) which is analog to a) when interchanging the roles of $n_1$ and $n_2$.
Starting with the edge-contraction accompanied by Case I.a, we have to delete node $n_1$ from the block of the current partition. The block wherein $n_1$ is contained is named $b(n_1)$. Subsequently, $n_2$ is added to $b(n_1)$ which may cause a change in the block order. To finally obtain the new partition, the block is, if required, rearranged inside the current partition according to the sorting defined in Section 3.5.6. The proceeding for Case I.b is similar, except that the blocksize of $b(n_1)$ increases by one. Consequently, the position of $b(n_1)$ either remains or is shifted to the right within its current partition.
Facing Case II., we have to distinguish whether $n_1, n_2$ are in the same block or in different blocks. For the latter case, the different blocks are merged into one block which is then placed into the current partition according to the defined sorting. By deleting nodes $n_1$ or $n_2$ from the blocks, we must account for the case that the respective blocks may be empty - or decay - after node removal. In this case, we do not need to merge the blocks (possibly merge blocks for II.a and II.c). We note that if a decaying block is a marked block, then the child of the current BDD-partition-node must be a BDD false-leaf.
For the deletion operation with regard to Case I. (see Table 6.4), we have to create a new block for node $n_2$ and appropriately place the new block into the current partition. Other-

wise, all other partition manipulating operations are similar to those for contraction.

Table 6.1.: Case-by-case analysis for deduction of new partitions

| | **I. New frontier node $n_2$:** $n_1 \in F_{k-1}, n_2 \in F_k$ | **II. No new frontier node:** $n_1, n_2 \in F_{k-1}$ |
|---|---|---|
| **a)** $n_1 \notin F_k, n_2 \in F_k$ | relevant | relevant |
| **b)** $n_1, n_2 \in F_k$ | relevant | relevant |
| **c)** $n_1, n_2 \notin F_k$ | irrelevant | relevant |
| **d)** $n_1 \in F_k, n_2 \notin F_k$ | irrelevant | relevant |

Table 6.2.: Contract I.

| | |
|---|---|
| **a)** | delete $n_1$, add $n_2$ to $b(n_1)$ & rearrange $b(n_1)$ |
| **b)** | add $n_2$ to $b(n_1)$ & rearrange $b(n_1)$ |

Table 6.3.: Contract II.

| | $n_1, n_2$ **are in the same block** | $n_1, n_2$ **are in different blocks** |
|---|---|---|
| **a)** | delete $n_1$ & check if $b(n_1)$ decays, sort blocks | delete $n_1$ & check if $b(n_1)$ decays, possibly merge blocks & sort blocks |
| **b)** | return current partition | merge blocks & sort blocks |
| **c)** | delete $n_1, n_2$ & check if $b(n_{1/2})$ decays, sort blocks | delete $n_1, n_2$ & check if $b(n_1)$ or $b(n_2)$ decay, possibly merge blocks & sort blocks |

Table 6.4.: Delete I.

| | |
|---|---|
| **a)** | delete $n_1$ & check if $b(n_1)$ decays & create new block $b(n_2)$ & sort blocks |
| **b)** | create new block $b(n_2)$ & sort blocks |

Table 6.5.: Delete II.

| | $n_1, n_2$ **are in the same block** | $n_1, n_2$ **are in different blocks** |
|---|---|---|
| **a)** | delete $n_1$ & check if $b(n_1)$ decays, sort blocks | delete $n_1$ & check if $b(n_1)$ decays, sort blocks |
| **b)** | return current partition | return current partition |
| **c)** | delete $n_1, n_2$ & check if $b(n_{1/2})$ decays, sort blocks | delete $n_1, n_2$ & check if $b(n_1)$ or $b(n_2)$ decay, sort blocks |

### 6.1.1. False-leaf when contracting

Now we have accumulated enough insight in order to explain by means of an example in Figure 6.1 the possible occurence of a BDD false-leaf when contracting an edge. For the terminals 0 and 5, the factoring order is $e_0, e_1, e_2, e_3, e_4, e_5$. At level three of the BDD, the frontier set equals $\{2, 3, 4\}$. A possible setup of a partition could be [23]*[4]. Contraction of edge $e_4$ results in the decay of the marked block [23]*. Consequently, terminal node 0 is disconnected from the boundary set and we have to link the current BDD-node of partition [23]*[4] to a false-leaf.

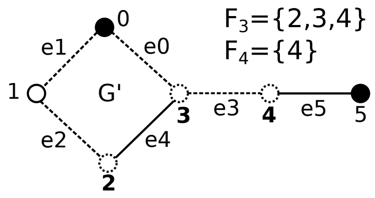


Figure 6.1.: Example for false-leaf when contracting edge.

## 6.2. A new heuristic for finding a good variable order

We have stated that the problem of finding the tree or path decomposition with the smallest tree- or pathwidth is NP-hard for an arbitrary graph [4]. Hence, one must resort to heuristics. To obtain a substantial improvement of the approach, our goal is to find a better heuristic than the currently proposed breadth-first-search variable ordering. For all kinds of graph structures, the new heuristic should preferably find a variable ordering which leads to a maximal frontier with lower or at most equal size in comparison to the bfs-heuristic. In case the cardinality of the maximal frontier set is equal for both heuristics, the cardinalities of all frontier sets $F_k$ obtained from the new heuristic should be less than or equal to those from the bfs-heuristic.
At each level $k$ of the factoring order (the number of levels equals the number of edges of the input graph $G$), we have a certain set of frontier nodes $F_k$. The cardinality of $F_k$ highly depends on the previous choices of edges to be factored and is at least two. To keep the frontier size as small as possible, we must focus on adding as few vertices as possible to $F_k$ and getting rid of as many vertices as possible from $F_k$ at each iteration of the edge choice (see Algorithm 8). By choosing an edge to factor, at most two new vertices can be added to the current frontier $F_k$. This is done by choosing an edge which is not adjacent to $F_k$. Hence, a better choice must be an edge adjacent to $F_k$. Each time an edge is chosen, it is deleted from $G$. Algorithm 8 ends when all edges are deleted from $G$. The candidate for our *first priority choice* must be an edge which connects two vertices $n_1, n_2 \in F_k$. In case this choice can be made, the chosen edge is added to the variable order list *order* and deleted from $G$ (see Algorithm 8 line 8). The size of $F_k$ would at least remain constant or at most decrease by two. Everytime a choice is made, we subsequently have to check whether the affected nodes $n_1, n_2$ must be added or removed from $F_k$ and subsequently update $F_k$. This fact is considered in lines 9, 15 and 20 of Algorithm 8. The maximal size of $F_k$ is updated after each iteration in line 5.
Otherwise, if the set of $F_k$-adjacent edges does not contain any edge $e = (n_1, n_2)$ with $n_1, n_2 \in F_k$, we decide for an edge $e = (n_{min}, n_2)$ adjacent to the frontier node of lowest degree $n_{min} := \underset{n \in F_k}{\arg\min} \deg(n)$, where the neighborhood of $n_2$, $N(n_2)$, contains at least one node from the current frontier set $F_k$ excluding $n_{min}$ (*second priority choice*, see Algorithm 8 line 12). If there are several nodes $n_{min}$ meeting the second priority and having the same degree, we randomly decide for one of them. Otherwise, if there is no edge which meets the second priority, we make our choice according to the *third priority*: among the set of frontier nodes with minimal degree $J$, we choose the egde with end vertex $n_2$ which is from the neighborhood of $J$ and has minimal degree (see Algorithm 8 line 18). If the choice is not clear, we again randomly decide for one possibility.

The reason behind this choice is that we expect nodes that are still contained in $F_k$ and nodes that will be added to $F_k$ to leave $F_k$ as early as possible in one of the following iterations. This also means that we needed at least $deg(v) - 1$ more edge decisions for a recently added node $v$ to leave the frontier set.
Finally, the variable order is returned as an ordered list of edges $order$. The variable order and the size of the maximal frontier set $|F_{max}|$ depends on the initial input node $n$. So $findVarOrder$ is conducted for all nodes of $v \in G$ and the order is taken for the initial node $n$ that gives the lowest size for $F_{max}$.

### 6.2.1. Complexity discussion

In Algorithm 8, we decide $|E|$ times for an edge. In the worst case, there are no matches for the first and second priority choice in each iteration $k$. Since each of the three priority choices implies the analysis of the neighborhood $N(F_k)$ of boundary set $F_k$, we have a worst case complexity of $O\big(|F_k| \cdot |N(F_k)|\big)$ for each iteration $k$, where $N(F_k) := \{v \notin F_k : (u, v) \in \overline{E_k},\ u \in F_k\}$. Define $N_{max}$ to be the maximal cardinality neighborhood among all frontier sets: $N_{max} := \max_k |N(F_k)|$. Overall, the worst case runtime complexity is $O\big(|V| \cdot |E| \cdot |F_{max}| \cdot N_{max}\big)$. Note that $N_{max} \leq |V| - |F_{k'}|$, with $k' := \arg\max_k |N(F_k)|$.

### 6.2.2. Application on Example Graphs

In the following, we compare the results of both heuristics based on two general regular networks (a Fan network and a complete N-node network, and an irregular network), the ARPANET from 1979 (consisting of 59 nodes and 71 edges, see [70]).

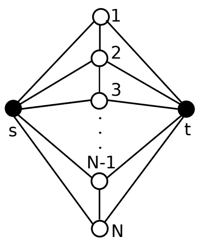


Figure 6.2.: Fan network

**Fan network** For the regular graph called the "Fan" graph (see Figure 6.2), the new heuristic finds a variable ordering of $|F_{max}| = 3$ with regard to the initial nodes $n \in S := \{s, t, 1, 2, N-1, N\}$. Otherwise, the new heuristic yields $|F_{max}| = 4$ for all $n \in V \setminus S$.
For $n \in \{s, t\}$ the bfs-heuristic yields $|F_{max}| = N$ or $|F_{max}| = N + 1$. With respect to initial nodes in $V \setminus \{s, t\}$, the bfs-heuristic yields $3 \leq |F_{max}| \leq N + 2$ depending on the order of the bfs-traversal.
For the bfs-heuristic, we have a huge difference of $N - 1$ for $|F_{max}|$ between the worst and the best case, while the difference of values for $|F_{max}|$ is only one for the new heuristic. For this particular network, we obviously obtain a better variable ordering with the new heuristic.

Algorithm 8: $findVarOrder$

```
Require: node $n$, Graph $G$
 1: Choose an edge $(n, n') := e \in E$ with $n' \in \{v \in N(n) : \arg\min \deg(v)\}$
 2: order.add($e$), G.delete($e$)
 3: $F_0$.add($n$), $F_0$.add($n'$)
 4: while $F_k \neq \emptyset$ do
 5:     Update size of $F_{max}$
 6:     //1st priority choice
 7:     for all edges $(n_1, n_2) =: e \in \overline{E}_k$ with $n_1, n_2 \in F_k$ do
 8:         order.add($e$), G.delete($e$)
 9:         Update $F_k$
10:     end for
11:     //2nd priority choice
12:     Choose an edge $e = (n_1, n_2)$ with $n_1, n_2$ from
        $H := \Big\{n_1 \in F_k : \arg\min \deg(n_1) \cap \big(F_k \setminus \{n_1\}\big) \cap N(n_2)\Big\}$
13:     if $\big(F_k \setminus \{n_1\}\big) \cap N(n_2) \neq \emptyset$ then
14:         order.add($e$), G.delete($e$)
15:         Update $F_k$
16:     else
17:         //3rd priority choice
18:         Choose an edge $e = (n_1, n_2)$ with
            $n_1 \in J := \{v \in F_k : \arg\min \deg(v)\}$, $n_2 \in \{v \in N(J) : \arg\min \deg(v)\}$
19:         order.add($e$), G.delete($e$)
20:         Update $F_k$
21:     end if
22: end while
23: return order
```

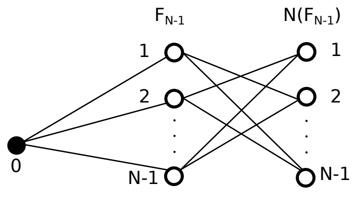


Figure 6.3.: N-node complete graph (two-level bfs recursion)

**Complete Graph** Applying the bfs heuristic to a complete N-node graph yields a variable ordering which leads to a sequence of frontier sets with the following cardinalities:

$$\underbrace{2,3,\ldots,N-1}_{N-2\ levels},\underbrace{N-1,\ldots,N-1}_{N-2\ levels},\underbrace{N-2,\ldots,N-2}_{N-3\ levels},\ldots,4,4,4,3,3,2 \tag{6.1}$$

So we have in level one of the BDD-tree a frontier set $F_1$ with size two, in level two $|F_2| = 3$ and so forth, whereas the last level is omitted, since the size of the last frontier set equals zero. The accomplishment of this sequence is clarified by Figure 6.3. It shows the two recursion levels of the bfs. In the first level N-1 edges are consecutively chosen. Since they are deleted together with node 0 (after being visited), each of the remaining N-1 nodes has N-2 neighbors. Again, N-2 edges which are adjacent to one of the remaining nodes are chosen and deleted from the graph. In the same way the choices are made for the rest. Applying the new heuristic to the complete graph, yields the reverse order of sizes of frontier sets. In this sense, Expression 6.1 is mirrored. This gives $|F_1| = 2$, $|F_2| = 3$, $|F_3| = 3$, and so on. The mirroring fact is shown for a complete 10-node graph in Figure 6.4. In total, the sum of frontier sizes are equal and in addition to that the maximal frontier size is N-1 for both. One would be left with the impression that both heuristics should lead to the same performance of the decomposition algorithm. However, this is not the case: though frontier sizes in one of the first levels are higher for the bfs heuristic, the number of possible children in each BDD-level $l$, $l \in \mathbb{N}$, are limited to $2^l$ due to the binary structure of the BDD. In contrast to this, the higher frontier set cardinalities obtained by the new heuristic for the last levels $l$, can lead to a dramatic growth of the BDD width, since $l$ has increased. This fact is confirmed by measurements for the complete 10-node graph in Section 8.1. So for complete graphs, it is better to utilize the bfs heuristic.

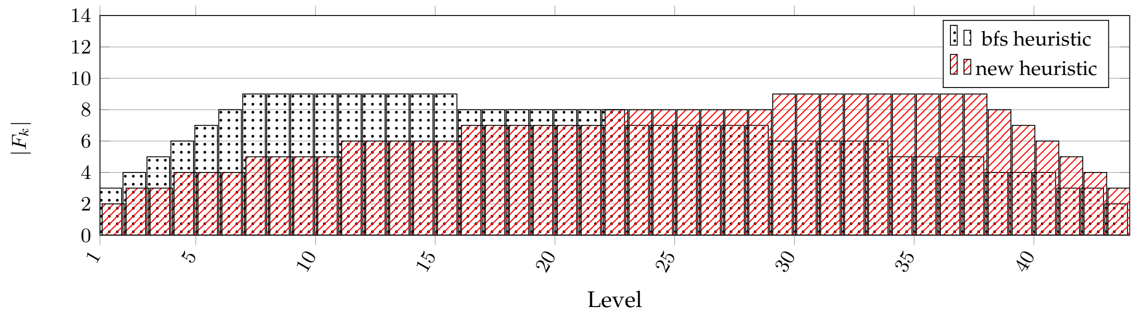


Figure 6.4.: Frontier size in each BDD level for the complete 10-node network

**ARPANET 1979** Nevertheless, we have found that the new heuristic generally delivers much better results for irregular graph structures such as the ARPANET 1979[2]: applying Algorithm 8 to node 30 we obtain $|F_{max}| = 6$. Otherwise, by using the bfs-heuristic we obtain $|F_{max}| = 16$. Considering that for $|K| = 2$ there are at most 810 different partitions for $|F_{max}| = 6$ (see Table 8.6) and around 41,92 billion partitions for $|F_{max}| = 16$, one can roughly guess the huge difference in time and memory requirement claimed by

[2]The graphical representation of the ARPANET 1979 is given in [70].

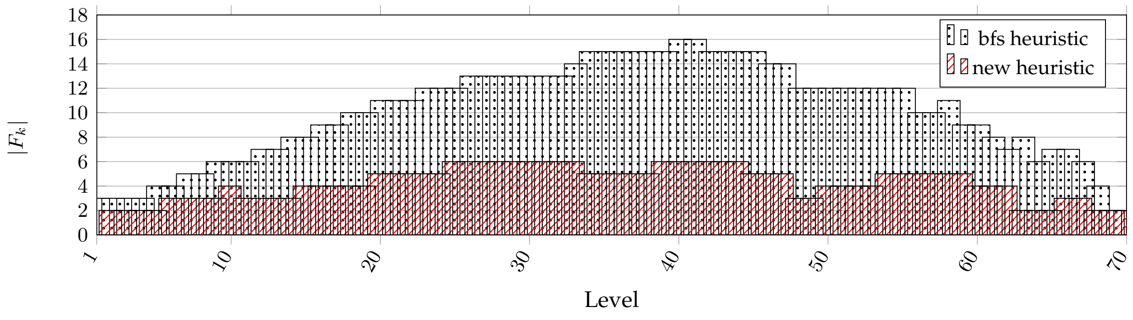


Figure 6.5.: Frontier size in each BDD level for the ARPANET 1979

the computation. Not only $|F_{max}|$ is vastly larger, also the cardinalities for all frontier sets are significantly higher for the bfs heuristic (see Figure 6.5). Consequently, this inevitably leads to a huge difference in runtime and memory demand approved by results which can be found in Section 8.1. Furthermore, we have observed that in most cases of regular graph structures - such as grid networks - it is of no matter which heuristic to use . For those structures we cannot expect any improvements since both heuristics deliver variable orderings which have the same size of $F_{max}$. Moreover, we have not yet found any counterexamples where the new heuristic produces variable orderings with higher $|F_{max}|$ than the current bfs-heuristic. Except for complete graphs, the new heuristic is consequently at least as good as the bfs-heuristic and finds for many graphs - especially irregular graphs - a better variable ordering with a significantly lower cardinality of $F_{max}$.

# 7. Improving the Dotson-Gobien Algorithm

In Section 4.3.4 we have addressed the maintenance of different data structures for the DG method. Certain drawbacks of them can pose a serious issue in terms of the DG method's expectantly high memory demand. For being able to cope with inputs of larger dimensions and additionally obtain more accurate bounds, we will show how to keep the memory consumption of the DG method with reductions as low as possible. To remedy the redundant information stored in the event queue, the lack of reductions of the original approach and Deo and Medidi's high memory demand for the storage of each evolving subgraph, we propose the use of a so-called *delta tree*. It keeps track of all changes made to the original graph due to reductions and partitioning.
Even though memory consumption is kept as low as possible, the limitation is soon reached by large graph sizes due to the exponential growth of this problem. Another key idea is to migrate the *delta tree* to hard disk. The data to be written is arranged in a certain way in order to comply with the hard disk's sequential read and write access (see 7.1.1).

## 7.1. Obtaining better bounds with the $\Delta$tree

All the intermediary results of this method can be stored in a recursion tree called a *delta tree*, $T_\Delta$, whose structure is as follows: starting with the root node, we store all reductions performed on the original input graph. In general, each node of the tree stores all consecutively performed reductions on a certain subgraph. The number of child nodes equals the length of the shortest $s$-$t$ path found at the parent node. The edges connecting the parent node with the child nodes contain the information for partitioning the respective subgraph represented by the parent node. In the course of the algorithm the tree emerges level by level according to breadth-first-search order. Each leaf of the tree represents a subgraph or task to be processed. This subgraph can be reconstructed by tracing back the *delta path*, $P_\Delta$, from leaf to root. Apart from the subgraph, the appropriate edge probability map, *EdgeProbMap* (epm), and the accumulated path/cut terms can be reconstructed by the help of $P_\Delta$. Below, we show how the relevant information can be stored in $T_\Delta$.
Each series or parallel reduction involves two edges, $e_1$ and $e_2$, whereas the first one's probability $p_{e_1}$ is readjusted with respect to the performed reduction: $p_{e_1} := p_{e_1} \cdot p_{e_2}$ (for series-) or $p_{e_1} := p_{e_1} + p_{e_2} - p_{e_1} \cdot p_{e_2}$ (for parallel reduction). Edge $e_2$ will be contracted in case of a series reduction and deleted in case of a parallel reduction. In general, the contraction of an edge $e$ contains the following steps: first delete $e$, then merge the border nodes of $e$ to one node. In both cases (series- and parallel), edge $e_2$ is removed from the graph. Edge $e_1$ remains in the graph and $p_{e_1}$ is captured in the *EdgeProbMap*. The operation $\circ$ is introduced for distinguishing a contraction ($\circ = +$) from a deletion ($\circ = -$). To distinguish $e_1$ and $e_2$, any reduction term, $red$, comprises a semicolon. Its string representation

is as follows:

$$red := "e_1; \circ e_2"$$

Any $T_\triangle$-node $n$ of a graph containing $l$ reductions is represented by:

$$n := "red'_1 red'_2 \dots red'_l"$$

The reductions are separated by an acute accent.
Based on the shortest path of length $r$, the $r$ subproblems are each derived by edge deletion and contraction operations. All edges which are to be contracted, are standing upfront followed by the last edge $e_r$ which is to be deleted (by the − operation). Again those edges can be separated by a semicolon. Suppose we have found a path of length $r$ at a node $n$ in the *delta tree*, then the $r$ *delta tree* edges $e_\triangle$ emerging from parent node $n$ are defined as follows:

$$e^1_\triangle := " - e_1" \quad e^2_\triangle := "e_1; -e_2" \quad \dots \quad e^r_\triangle := "e_1; e_2; \dots ; e_{r-1}; -e_r"$$

Every *delta path* $P_\triangle$ of recursion level $m$ is an alternating sequence of $T_\triangle$-nodes $n_i$ (commencing with root $n_0$) and $T_\triangle$-edges $e^i_\triangle$, $0 \leq i \leq m$:

$$P_\triangle := "n_0 > e^0_\triangle > n_1 > e^1_\triangle > \dots > e^{m-1}_\triangle > n_m > e^m_\triangle"$$

### 7.1.1. Memory-efficient implementation

In this part we describe the whole modified algorithm which generates an output file *FNext* by taking an input file *FPrev* at each recursion level. After initialising the input graph, the files and all appropriate maps in Algorithm 9, Algorithm 10 is invoked for processing the initial graph. There we first check the connectivity of the graph. If the graph is connected, we look for possible reductions in line 12. The changed probabilities due to reductions are updated in *epm* at line 13. Additionally, the performed graph manipulations caused by reductions are captured in a string as described above and again this string is contributed to *line* (line 14). Furthermore, *line* is enriched with the respective subproblems according to the shortest path $sp$. Finally, *line* is written to *FNext*.
Now we define the rules for writing $T_\triangle$ to *FNext*. Each line in *FNext* (*linebranch*) represents a branch of $T_\triangle$ comprising $k$ leafs. The string representation therefore is defined as follows

$$linebranch := "n_0 > e^0_\triangle > n_1 > e^1_\triangle > \dots > e^{m-1}_\triangle > n_m : e^{sub_0}_\triangle, e^{sub_1}_\triangle, \dots, e^{sub_k}_\triangle"$$

Hereby the $k$ *delta tree* edges $e^{sub}_\triangle$ are separated by a comma and attached by a colon. In Algorithm 10 the first part of *linebranch* - to the left of the colon - is aggregated by *aggregateLeaf(e)* with the respective subproblems. At the end of the for-loop the completed *linebranch* is written to file *FNext*. After all *linebranches* of the current $T_\triangle$ level have been written, the *FNext* file is renamed to *FPrev* and then taken as input for the next recursion in Algorithm 11. A *delta path* (task $P_{\triangle_i}$) with $i \in \{0, 1, \dots, k\}$ is derived from *linebranch* and is represented as follows

$$P_{\triangle_i} := "n_0 > e^0_\triangle > n_1 > e^1_\triangle > \dots > e^{m-1}_\triangle > n_m > e^{sub_i}_\triangle"$$

In Algorithm 11, each $P_\triangle$ deduced from *linebranch* is processed by Algorithm 12 which reads $P_\triangle$ and reconstructs the subgraph and the appropriate accumulated terms *acc*. At the end of each level of $T_\triangle$, the current bounds are printed out.

Algorithm 9: Main

```
1: $Init()$; //Initialize $inputGraph, epmInit, FNext, FPrev$
2: $FPrev = computeRel(inputGraph,$new $List(), epmInit, "", FNext)$;
3: $FNext =$ new File;
4: $bfsLevel(FPrev)$;
```

Algorithm 10: computeRel

```
Require: RBD $Graph$,List $acc$,EdgeProbMap $epm$,String $line$,File $FNext$
1: bool $PorC$; //Condition variable for determining path/cut
2: bool $b = Graph.findPath()$;
3: if $b == false$ then
4:    double $tmp = computeProduct(acc)$;
5:    if $PorC == true$ then
6:       $Paths = Paths + tmp$;
7:    else
8:       $Cuts = Cuts + tmp$;
9:    end if
10:   return
11: end if
12: SPRed $red = Graph.reduce()$ //Preprocessing: Reduce graph.
13: $epm = red.getEdgeProbMap()$; //Update edge probabilities
14: $line = line + red.getString()$; //Extend line with reduced terms
15: List $sp = BFS.shortestPath(Graph)$; //Find shortest path
16: for each $e \in sp$ do
17:    $line = line + aggregateLeaf(e)$;
18: end for
19: $FNext.write(line)$;
20: return $FNext$
```

Algorithm 11: bfsLevel

```
Require: File $FPrev$
1: for each $linebranch \in FPrev$ do
2:    //proceed each subproblem (path)
3:    for each $sub \in linebranch$ do
4:       RBD $rbd = readTaskBranch(sub)$;
5:       $FNext = computeRel(rbd, accumlation, epm, line, FNext)$;
6:    end for
7: end for
8: if $FNext.IsEmpty()$ then
9:    return
10: end if
11: $FPrev = FNext$;
12: $FNext =$ new File;
13: print "upper bound for Unreliability = $1 - Paths()$";
14: print "lower bound for Unreliability = $Cuts()$";
15: $bfsLevel(FPrev)$;
```

Algorithm 12: readTaskBranch

```
Require: String $sub$
1: String[] $deltapath = sub.split(>)$;
2: for each $i \in deltapath$ do
3:    $processNode(deltapath[i])$; //reductions
4:    $i = i + 1$;
5:    $processEdge(deltapath[i])$; //merge&delete operations
6: end for
7: $epm.update()$; //recreated EdgeProbMap
8: $acc.update()$; //recreated accumlation List
9: return $rbd$; //reconstructed graph
```

# Part IV.

# Measuring and Comparing

# 8. Evaluation on different network structures

To prove the relevance of our extensions and modifications, we have compared our approaches to the original approaches based on 27 example networks (Figure 8.1) and 5 randomized sets of networks. Some of the networks are taken from [85, 36, 25] in order to allow for comparison. Those are networks 2-8, 10, 11, 13, 14, 16, 17, 26. The other networks are well-known examples from the literature. For example Nw.22 is known as the street network with parameter N. This network has N horizontal edges in each row and N-1 edges in each column. Nw.18 is the K4 ladder of dimension 21 (having 21 vertical edges). Its reliability can be determined analytically for any dimension by method [72]. Also the ARPANET 1979 - one of the world's first operational packet switching networks, the progenitor of what was to become the global Internet is taken into consideration (see [70]). To cover as many different graph structures as possible and to realise a fair comparison, we have additionally created different sets of random network structures (see Nw. 28-32).

**Randomized network structures** Based on the work in [77], we have used the GenGraph tool to generate randomized networks whose degree distribution follows the power-law. According to [77], the power-law distribution for the node degree seems to give a good replication of the Internet or real world communication network structures. The generated graphs depend on a set of four parameters: N (the number of nodes), $\alpha$ (the exponent of the power law distribution), mindeg (the minimal node out-degree) and maxdeg (the maximal out-degree). We have chosen the following parameter configuration for

- Nw.28 : $|V| = 130$, $\alpha = 4$, mindeg=2, maxdeg=3
- Nw.29 : $|V| = 66$, $\alpha = 3$, mindeg=3, maxdeg=4
- Nw.30 : $|V| = 45$, $\alpha = 2$, mindeg=2, maxdeg=4
- Nw.31 : $|V| = 15$, $\alpha = 4$, mindeg=5, maxdeg=8
- Nw.32 : $|V| = 100$, $\alpha = 3$, mindeg=3, maxdeg=8

In the field of fault-tolerant systems, it is realistic to assume double, triple or quad redundancy justifying our choice for mindeg and maxdeg in Nw. 28-30. The parameter $\alpha$ determines the interlacement of the network: the graph structure turns out to be more "planar" and less complex with increasing $\alpha$. Since the network gets more complex with a lower $\alpha$, we had to decrease the number of nodes to maintain the computation feasible in a short time. Each result listed for Nw. 28-32 is an average made up of over 100 randomized samples $\times$ number of all different s-t-pairs, that equals $100 \times \left(\sum_{i=1}^{|V|-1} i\right)$. To give an initial impression of the randomized graphs' appearance, we have illustrated in Figure 8.2 four representative samples taken from the generation. Nw. 32 is not depicted, however, we can imagine it to look like a highly interlaced ball of wool.

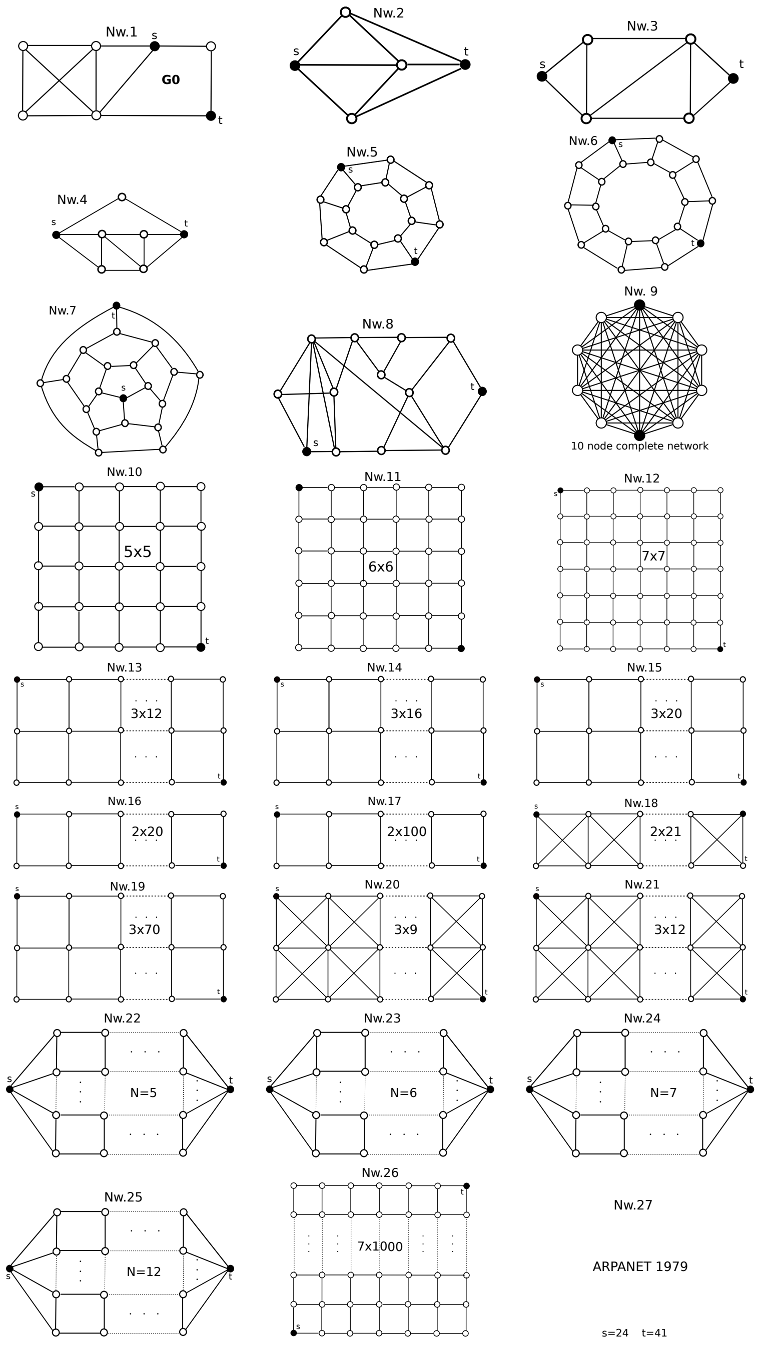


Figure 8.1.: Benchmark Networks 1-27.

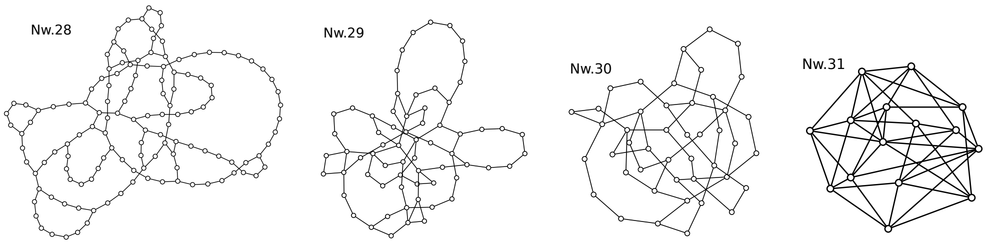


Figure 8.2.: Samples for the randomized networks 28-31

The KLY and decomposition approaches were implemented in the C++ programming language and the Dotson-Gobien approach was implemented in JAVA. The experiments were run on an Intel Xeon 5670 (Westmere EX) 2.93GHz machine with 12M L3 Cache and 40GB of RAM. For comparability reasons, all edges are assumed to fail with probability of 0.1. In fact, the failure probabilities can be arbitrary and in addition they do not have any influence on the performance of the algorithms. First we do a pre-post comparison for each method regarding their runtime and memory demand. Then all three approaches are contrasted against one another.

## 8.1. Results after applying modifications

In the tables listed underneath, index two is assigned to the original approaches whereas the modified new approaches bear index one. The computed reliability values, network parameters $|V|$, $|E|$ and number of ROBDD nodes for the KLY and number of OBDD nodes for the decomposition approach (abbreviated with "D") are shown in Table 8.1. Tables 8.1 and 8.4 are of relevance for the KLY and decomposition approach. Separately, the results are found for each of them in Tables 8.2 and 8.3. The measurements for the Dotson-Gobien approach are listed in Tables 8.5 and 8.6.

### 8.1.1. Results for the KLY approach

For effectively applying logic operations on ROBDDs, we have incorporated the BuDDy BDD library (from [44]) in the KLY approach. The size of the ROBDD is determined by bfs variable ordering and since the order of this edge traversal is ambiguous, we might not obtain the same ROBDDs as in [85] for the same input networks. However, according to the bfs-variable ordering, the resulting ROBDDs are the same for either of our implemented approaches. In the result Tables 8.1, 8.2, 8.4 our new approach bears index 1 and the original KLY approach index 2.
The number of $G_{node}$/Hits, runtimes and speedup are illustrated in Table 8.2. Examples which we could not compute within an acceptable amount of time/memory are marked with "-". Table 8.4 exposes the peak memory consumption and the relative memory consumption $\frac{m1}{m2}$ for each example network. Networks where no redundant biconnected components occur and hence no improvement with our new approach could be gained, are underlined. The memory consumption is equal for both methods when applied to this kind of network. Yet, the runtimes are expected to be slightly shorter for the original ap-

Table 8.1.: Reliability values and number of BDD-nodes for $KLY$ and $D_2$ using bfs heuristic, and $D_1$ using new heuristic

| Nw | $\lvert V\rvert$ | $\lvert E\rvert$ | $\lvert BDD_{KLY}\rvert$ | $\lvert BDD_{D_1}\rvert$ | $\lvert BDD_{D_2}\rvert$ | Reliability |
|---|---|---|---|---|---|---|
| ***1*** | 7 | 11 | 26 | 18 | 63 | 0.979257 |
| ***2*** | 5 | 8 | 23 | 31 | 31 | 0.997632 |
| ***3*** | 6 | 9 | 16 | 22 | 22 | 0.977184 |
| ***4*** | 7 | 11 | 31 | 27 | 42 | 0.995665 |
| ***5*** | 16 | 24 | 450 | 361 | 667 | 0.995553 |
| ***6*** | 20 | 30 | 558 | 491 | 821 | 0.994395 |
| ***7*** | 20 | 30 | 3,995 | 3,165 | 6025 | 0.99712 |
| ***8*** | 13 | 23 | 275 | 155 | 412 | 0.987428 |
| ***9*** | 10 | 45 | 74,885 | 146,987 | 58,291 | 1.000000 |
| ***10*** | 25 | 40 | 1,147 | 2,104 | 1,694 | 0.975557 |
| ***11*** | 36 | 60 | 4,969 | 10,121 | 7,227 | 0.975645 |
| ***12*** | 49 | 84 | 20,752 | 46,795 | 29,821 | 0.975659 |
| ***13*** | 36 | 57 | 316 | 474 | 474 | 0.961730 |
| ***14*** | 48 | 76 | 436 | 654 | 654 | 0.956266 |
| ***15*** | 60 | 97 | 556 | 834 | 834 | 0.950832 |
| ***16*** | 40 | 58 | 114 | 168 | 169 | 0.784482 |
| ***17*** | 200 | 298 | 594 | 888 | 889 | 0.304293 |
| ***18*** | 42 | 101 | 372 | 563 | 564 | 0.995652 |
| ***19*** | 210 | 347 | 2,056 | 3,084 | 3,084 | 0.885461 |
| ***20*** | 27 | 74 | 1,495 | 1,754 | 2,249 | 0.997996 |
| ***21*** | 36 | 101 | 1,996 | 2,480 | 2,951 | 0.997996 |
| ***22*** | 22 | 41 | 1,686 | 2,411 | 2,466 | 0.999909 |
| ***23*** | 32 | 61 | 8,610 | 11,534 | 12,460 | 0.999987 |
| ***24*** | 44 | 85 | 41,904 | 52,936 | 60,057 | 0.999998 |
| ***25*** | 71 | 265 | - | 81,998,602 | 109,023,610 | 1.000000 |
| ***26*** | 7,000 | 12,993 | - | 12,968,704 | 12,951,730 | 0.975357 |
| ***27*** | 59 | 71 | 917,328 | 2,880 | 14,469,848 | 0.956042 |
| ***28*** | 130 | 140 | 122,004 | 15,312 | 1,587,344 | |
| ***29*** | 66 | 85 | - | 245,615 | - | |
| ***30*** | 45 | 59 | 38,067 | 8,950 | 967,400 | |
| ***31*** | 15 | 44 | 365,337 | 39,707 | 493,739 | |
| ***32*** | 100 | 210 | - | - | - | |

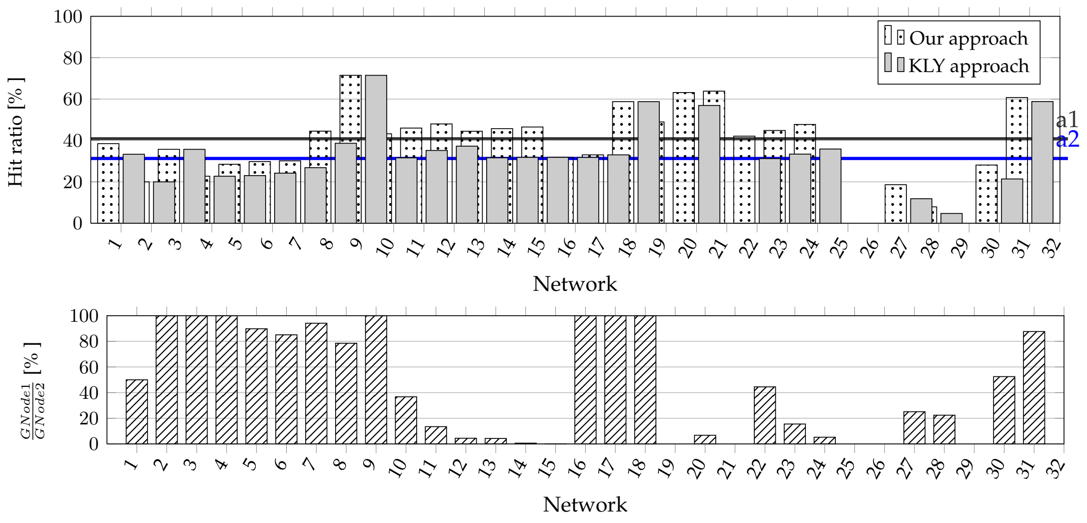


Figure 8.3.: Hit ratio and ratio of hashmap sizes

Table 8.2.: Comparison of results (original KLY vs. improved KLY)
$t_1$ := runtime of the improved KLY method,
$t_2$ := runtime of the original KLY method,
$\frac{t_2}{t_1}$ := speedup

| **Nw** | **GNode1** | **GNode2** | **Hit1** | **Hit2** | $t_1$ | $t_2$ | $\frac{t_2}{t_1}$ |
|---|---|---|---|---|---|---|---|
| ***1*** | 8 | 16 | 5 | 8 | <1ms | <1ms | 1 |
| ***2*** | 8 | 8 | 2 | 2 | <1ms | <1ms | 1 |
| ***3*** | 11 | 11 | 5 | 5 | <1ms | <1ms | 1 |
| ***4*** | 12 | 12 | 5 | 5 | <1ms | <1ms | 1 |
| ***5*** | 123 | 137 | 49 | 41 | <1ms | <1ms | 1 |
| ***6*** | 205 | 241 | 87 | 77 | <1ms | <1ms | 1 |
| ***7*** | 577 | 613 | 249 | 225 | 0.02 | 0.02 | 1 |
| ***8*** | 106 | 135 | 85 | 85 | <1ms | <1ms | 1 |
| ***9*** | 1,025 | 1,025 | 2,568 | 2,568 | 0.32 | 0.31 | 0.97 |
| ***10*** | 774 | 2,106 | 589 | 973 | 0.04 | 0.06 | 1.5 |
| ***11*** | 7,849 | 58,226 | 6,686 | 31,606 | 0.99 | 2.53 | 2.56 |
| ***12*** | 137,899 | 3,112,115 | 127,282 | 1,847,623 | 30.59 | 206.62 | 6.75 |
| ***13*** | 725 | 16,958 | 580 | 7,842 | 0.08 | 0.69 | 8.63 |
| ***14*** | 1,629 | 272,614 | 1,372 | 127,313 | 0.29 | 15.1 | 52.07 |
| ***15*** | 3,077 | 4,363,912 | 2,676 | 2,040,215 | 0.83 | 385.45 | 464.4 |
| ***16*** | 75 | 75 | 35 | 35 | <1ms | <1ms | $1-\epsilon$ |
| ***17*** | 395 | 395 | 195 | 195 | 0.33 | 0.26 | 0.79 |
| ***18*** | 231 | 231 | 329 | 329 | 0.04 | 0.03 | 0.75 |
| ***19*** | 119,277 | - | 114,376 | - | 410.64 | - | - |
| ***20*** | 61,594 | 912,957 | 105,568 | 1,206,264 | 8.24 | 51.86 | 6.27 |
| ***21*** | 1,206,586 | - | 2,128,876 | - | 246.43 | - | - |
| ***22*** | 1,020 | 2,292 | 741 | 1,041 | 0.06 | 0.06 | 1 |
| ***23*** | 10,629 | 68,504 | 8,639 | 34,386 | 1.12 | 2.6 | 2.32 |
| ***24*** | 189,702 | 3,629,099 | 173,264 | 2,030,453 | 35.94 | 212.54 | 5.91 |
| ***25*** | - | - | - | - | - | - | - |
| ***26*** | - | - | - | - | - | - | - |
| ***27*** | 402 | 1600 | 92 | 215 | 3.35 | 3.45 | 1.03 |
| ***28*** | 623 | 2,783 | 78 | 200 | 0.52 | 1.04 | 2.0 |
| ***29*** | - | - | - | - | - | - | - |
| ***30*** | 844 | 1,613 | 308 | 409 | 0.12 | 0.12 | 1 |
| ***31*** | 10,531 | 12,017 | 16,306 | 17,155 | 3.46 | 3.28 | 0.95 |
| ***32*** | - | - | - | - | - | - | - |

proach since the overhead does not pay off for the new approach. In the worst case (see Nw.18), the new approach takes 1.33 times longer than the original approach. However, the additional time is in absolute values only 100 ms for a relatively large network. For smaller ones such as Nw.2, 3 and 4, the overhead spent is negligible. In most of the samples we can observe a significant speedup and reduction of memory consumption by applying our new approach. This becomes especially more significant for larger-sized networks. A remarkable speedup was achieved for Nw.15 with 464 times. Furthermore, our new approach needs only 0.3 percent of the memory claimed by the original one. Comparing the results with Nw. 13 and 14 one should expect these factors to rise for higher dimensions of this grid network. For Nw.19 it was not possible to finish the computation with the original approach in an acceptable time frame and without exhausting the memory.
A similar conclusion can be drawn for the other recursive network structures Nw.10-12 and Nw.20-24. Even more, Nw. 25 and 26 are unfeasible for the original and the modified approach. In general, we indicated a much higher number of $G_{node}$ for the original approach coupled with a higher number of hits. The number of $G_{node}$ can be regarded as the number of misses since a miss leads to a new hashmap entry. A large difference between the number of $G_{node}$ indicates that either numerous or large-sized biconnected components were removed in the course of the algorithm.
We can justify the hahsmap's value of benefit by means of the above-defined hit ratio $\frac{Hit}{|G_{node}|+Hit}$. The hit ratio for our new approach is for most examples significantly higher than for the original one (upper graph of Figure 8.3). On average, we have a hit ratio of 41 percent (a1) for the new approach and 32 percent (a2) for the original approach. For the scalable and regular network structures we notice that the hit ratio (or the benefit of the hashmap) increases linearly with the size of the networks. For Nw.13-15 the hit ratio remains even constant for the original algorithm. From these facts we conclude that the hashmap's benefit is clearly improved by using the new approach. Another fact that we can deduce from this diagram is that the hit ratio grows with the density or interlacement of the network. This can be observed for the sets of randomized graphs and especially for the 10-node complete graph where we have obtained the highest hit ratio. The diagram at the bottom of Figure 8.3 depicts the hashmap size ratio of the two approaches. For instance, in Nw.1 the hashmap size has shrunk to 50 percent with regard to the hashmap size of the KLY-algorithm. The more redundant biconnected components are removed, the more the hashmap shrinks. We find that the new approach performs better on graphs tending to be planar. For example, Nw.28 has a maximum out-degree of three and a low interlacement. The number of $G_{node}$ for the original approach is 4 times higher than the number of $G_{node}$ for the new approach. The new approach gives a speedup factor of 2 and 91.5 percent of the memory is needed. With decreasing $\alpha$ and increasing interlacement the number of $G_{node}$ for the original approach is only twice as much as the new one (see lower graph of Figure 8.3). For Nw.30 and 31 both approaches end up with the same average runtime. The memory consumption is expected to be lower since the number of $G_{node}$ differs significantly. However, compared to Nw.28 the memory savings with the new approach is proportionally lower. It can be seen that the number of BDD-nodes and the memory consumption have risen although the number of nodes and edges were decreased. This is traced back to the increased network interlacement/complexity related to a decreasing $\alpha$. To affirm this assertion, the randomized graphs in Nw.31 were given a high connectiv-

ity by fixing mindeg=5 and maxdeg=8. There, the overhead spent to find and to remove redundant biconnected components does not pay off in runtime. However, by looking at the number of $G_{node}$ we infer that with regard to Nw.28, 30 and 31 proportionally fewer redundant biconnected components occur and hence only a small amount of memory is less demanded by the new approach. Regarding runtime and memory consumption, the smaller-sized networks are of no consequence. In other words, it does not make any difference in runtime and memory consumption to compute them either with the original approach or the new one. Our measurements stress that apart from the network size, the network structure has even a higher impact on the performance of both approaches. Sparse and regular networks can be well handled by both algorithms whereas dense and irregular network structures with a high interlacement can easily lead to vast memory consumptions and long runtimes. So both approaches could not solve Nw.29 and above all Nw.32 which has a high connectivity and more than 200 edges.
In general, the memory consumption for the new approach is for every kind of network at most as much as for the original approach. In the worst case where no redundant biconnected components occur (here we do not involve the case with redundant nodes of degree one which is already covered by the original approach), the same amount of memory is consumed and the runtime of the new approach is extended by the overhead spent to detect biconnected components (see Nw.18).

### 8.1.2. Results for the decomposition approach

Unlike the KLY approach, there are two columns for the number of OBDD nodes for the decomposition approach in Table 8.1, brought off by the different strategies for determining the variable ordering. As stated above, the decomposition approach using bfs heuristic bears label $D_2$ and the use of the new heuristic is labeled with $D_1$ respectively. The different OBDD sizes are a significant evidence for differences in the performance on certain networks. In general, the OBDD sizes obtained from the new heuristic are significantly lower for unregular network structures in comparison to the bfs heuristic. The differences are especially huge when there are great variations of the $|F_{max}|$ value between both heuristics. For regular structures, we have a patchy situation. While it is better to use the new heuristic for Nw.20 and Nw.21, street networks (Nw.22-25), fan networks (see Section 6.2.2), the bfs heuristic leads to fewer BDD-nodes for complete N-node graphs or grid networks such as Nw.10-12. The reasons therefore were exemplarily discussed in Section 6.2.2 for the complete and fan network. In Table 8.3, the $|F_{max}|$ values, maximal BDD-width $W$ and the runtime are listed for both heuristics. Small differences of time ratio $\frac{t_2}{t_1}$ - with less than 1 percent - are annotated by an $\epsilon$. Again, we have marked those networks (with ') where the use of the new approach does not bring any improvements. For these cases, mostly the new heuristic yields a variable ordering with at most the same $|F_{max}|$ as the bfs heuristic. Since their $|F_k|$, $k = 1, 2, \ldots, m$ may differ, the resulting maximal BDD width $W$ or the number of BDD-nodes may also vary (see Nw.9). This is also the only network where the decomposition approach performs significantly better with the bfs-heuristic. For the small sized networks, such as Nw.1-6, where the runtimes are less than 1 ms, the improvements are not perceptible. They can be better traced on medium scaled networks such as Nw.24. The improvement scales up when the dimension of the street network is increased. Since $|F_{max}|$ is the same for both heuristics, the ratio of improvement is rela-

Table 8.3.: Comparison of results (decomposition method using bfs vs. new heuristic)
$t_1$ := runtime of the improved decomposition method,
$t_2$ := runtime of the original decomposition method,
$\frac{t_2}{t_1}$ := speedup

| **Nw** | $\lvert F_{max_1} \rvert$ | $\lvert F_{max_2} \rvert$ | $W_1$ | $W_2$ | $t_1$ | $t_2$ | $\frac{t_2}{t_1}$ |
|---|---|---|---|---|---|---|---|
| ***1*** | 3 | 5 | 4 | 12 | <1ms | <1ms | $1+\epsilon$ |
| ***2′*** | 3 | 3 | 8 | 8 | <1ms | <1ms | 1 |
| ***3′*** | 2 | 2 | 3 | 3 | <1ms | <1ms | 1 |
| ***4*** | 3 | 4 | 4 | 8 | <1ms | <1ms | $1+\epsilon$ |
| ***5*** | 4 | 5 | 27 | 72 | <1ms | <1ms | $1+\epsilon$ |
| ***6*** | 4 | 5 | 27 | 75 | <1ms | <1ms | $1+\epsilon$ |
| ***7*** | 6 | 7 | 363 | 830 | 0.01 | 0.03 | 3 |
| ***8*** | 3 | 5 | 10 | 47 | <1ms | <1ms | $1+\epsilon$ |
| ***9′*** | 9 | 9 | 17,007 | 6,670 | 0.70 | 0.23 | 0.33 |
| ***10′*** | 5 | 5 | 90 | 90 | <1ms | <1ms | 1 |
| ***11′*** | 6 | 6 | 297 | 297 | 0.04 | 0.03 | 1 |
| ***12′*** | 7 | 7 | 1,001 | 1,001 | 0.20 | 0.12 | 1 |
| ***13′*** | 3 | 3 | 9 | 9 | <1ms | <1ms | 1 |
| ***14′*** | 3 | 3 | 9 | 9 | <1ms | <1ms | 1 |
| ***15′*** | 3 | 3 | 9 | 9 | <1ms | <1ms | 1 |
| ***16′*** | 2 | 2 | 3 | 3 | <1ms | <1ms | 1 |
| ***17′*** | 2 | 2 | 3 | 3 | <1ms | <1ms | 1 |
| ***18′*** | 2 | 2 | 3 | 3 | <1ms | <1ms | 1 |
| ***19′*** | 3 | 3 | 9 | 9 | <1ms | <1ms | 1 |
| ***20*** | 4 | 5 | 37 | 141 | 0.01 | 0.01 | 1 |
| ***21*** | 4 | 5 | 37 | 141 | 0.01 | 0.01 | 1 |
| ***22*** | 5 | 5 | 90 | 90 | <1ms | <1ms | $1+\epsilon$ |
| ***23*** | 6 | 6 | 297 | 297 | 0.05 | 0.05 | $1+\epsilon$ |
| ***24*** | 7 | 7 | 1,001 | 1,001 | 0.23 | 0.27 | 1.17 |
| ***25*** | 12 | 12 | 534,888 | 534,888 | 666.20 | 936.40 | 1.41 |
| ***26′*** | 7 | 7 | 1,001 | 1,001 | 62.20 | 62.20 | 1 |
| ***27*** | 6 | 16 | 361 | 1,595,155 | 0.01 | 179.74 | 17,974 |
| ***28*** | 7 | 12 | 642 | 96,300 | 0.08 | 12.77 | 160 |
| ***29*** | 9 | 18 | 18,562 | - | 1.63 | - | - |
| ***30*** | 6 | 11 | 861 | 166,091 | 0.05 | 11.85 | 237 |
| ***31*** | 7 | 10 | 4,583 | 62,630 | 0.17 | 3.16 | 18.59 |
| ***32*** | 23 | 39 | - | - | - | - | - |

Table 8.4.: Memory consumption (KLY and decomposition approach)
$m_{KLY_1}$ := memory demand for improved KLY method
$m_{KLY_2}$ := memory demand for original KLY method
$\frac{m_{KLY_1}}{m_{KLY_2}}$ := memory consumption ratio between improved and original KLY method
$m_{D_1}$ := memory demand for improved decomposition method
$m_{D_2}$ := memory demand for original decomposition method
$\frac{m_{D_1}}{m_{D_2}}$ := memory consumption ratio between improved and original decomposition method

| Nw | $m_{KLY_1}$ | $m_{KLY_2}$ | $\frac{m_{KLY_1}}{m_{KLY_2}}$ | $m_{D_1}$ | $m_{D_2}$ | $\frac{m_{D_1}}{m_{D_2}}$ |
|---|---|---|---|---|---|---|
| ***1*** | <1MB | <1MB | $1-\epsilon$ | <1MB | <1MB | $1-\epsilon$ |
| ***2′*** | <1MB | <1MB | 1 | <1MB | <1MB | 1 |
| ***3′*** | <1MB | <1MB | 1 | <1MB | <1MB | 1 |
| ***4*** | <1MB | <1MB | 1 | <1MB | <1MB | $1-\epsilon$ |
| ***5*** | <1MB | <1MB | $1-\epsilon$ | <1MB | <1MB | $1-\epsilon$ |
| ***6*** | <1MB | <1MB | $1-\epsilon$ | <1MB | <1MB | $1-\epsilon$ |
| ***7*** | 1.75MB | 1.77MB | 0.988 | 1.85MB | 2.25MB | 0.822 |
| ***8*** | <1MB | <1MB | $1-\epsilon$ | <1MB | <1MB | $1-\epsilon$ |
| ***9′*** | 6.74MB | 6.74MB | 1 | 25.50MB | 10.21MB | 2.50 |
| ***10′*** | 1.59MB | 2.51MB | 0.633 | 1.54MB | 1.50MB | 1 |
| ***11′*** | 10.89MB | 35.60MB | 0.306 | 2.93MB | 2.54MB | 1 |
| ***12′*** | 139MB | 1717MB | 0.081 | 9.57MB | 6.34MB | 1 |
| ***13′*** | 1.7MB | 7.92MB | 0.215 | <1MB | <1MB | 1 |
| ***14′*** | 3.46MB | 98.44MB | 0.035 | <1MB | <1MB | 1 |
| ***15′*** | 5.30MB | 1755MB | 0.003 | <1MB | <1MB | 1 |
| ***16′*** | <1MB | <1MB | 1 | <1MB | <1MB | 1 |
| ***17′*** | 3.89MB | 3.89MB | 1 | 1.33MB | 1.36MB | 1 |
| ***18′*** | 1.61MB | 1.61MB | 1 | <1MB | <1MB | 1 |
| ***19′*** | 275MB | - | - | 1.66MB | 1.66MB | 1 |
| ***20*** | 35.30MB | 345MB | 0.102 | 1.42MB | 1.62MB | 0.877 |
| ***21*** | 649MB | - | - | 1.44MB | 1.64MB | 0.878 |
| ***22*** | 2.82MB | 3.22MB | 0.876 | 1.66MB | 1.66MB | $1-\epsilon$ |
| ***23*** | 12.93MB | 39.20MB | 0.330 | 2.93MB | 3.03MB | 0.967 |
| ***24*** | 257MB | 1955MB | 0.131 | 10.64MB | 11.82MB | 0.900 |
| ***25*** | - | - | - | 14.2GB | 19.9GB | 0.714 |
| ***26′*** | - | - | - | 1.8GB | 1.8GB | 1 |
| ***27*** | 65.23MB | 66.10MB | 0.987 | 1.25MB | ≈2.5GB | <0.001 |
| ***28*** | 25.91MB | 28.32MB | 0.915 | 4.00MB | 236.4MB | 0.017 |
| ***29*** | - | - | - | 39.10MB | - | - |
| ***30*** | 21.08MB | 21.83MB | 0.965 | 2.83MB | 174.80MB | 0.016 |
| ***31*** | 51.20MB | 51.90MB | 0.986 | 8.10MB | 88.51MB | 0.092 |
| ***32*** | - | - | - | - | - | - |

tively small. It already becomes significantly larger if the difference of $|F_{max}|$ would be only one. The difference in runtime and memory consumption (see Table 8.4) is then already perceptible for small sized networks, *e.g.* Nw.7. For this network, the runtime of our implementation with the new heuristic is even more than four times lower than the one from Hermann with OBDD-A2 [25], even though the architecture that Hermann has used works with a higher CPU clock rate (Intel Xeon 3.2GHz). For the medium sized unregular network - the "ARPANET 1979", the bfs heuristic yields an $|F_{max}|$ which is almost three times larger. In addition, the cardinalities for each boundary set are smaller or equal for the variable ordering achieved by the new heuristic. As a result, the differences between the runtimes and memory demands are remarkably large: the speedup is around 18,000 times and less than 0.1 percent of the memory is claimed with the new approach. The conspicuously better performance of the new approach is further approved by the results on the set of randomized graph structures. At this point we note that the decomposition approach finds its limit on the order of magnitude of $|F_{max}|$. The computation turns out to be impracticable for variable orderings where $|F_{max}| > 16$. For instance, the bfs-heuristic yields $|F_{max}| = 18$ for Nw.29 rendering the computation impratical. The same holds for Nw.32 where both methods deliver a practically too high $|F_{max}|$ value. Despite this shortcoming and under the condition that $|F_{max}|$ is not larger than 16, the new heuristic allows to compute certain networks which could not be computed before because of a too large $|F_{max}|$ obtained by the bfs heuristic. Overall, the application of the new heuristic for both unregular and regular network structures is to be favored.

### 8.1.3. Results for the modified Dotson-Gobien approach

We give two result tables for the Dotson-Gobien approach. One where a relative accuracy of ten percent for the unreliability is achieved (Table 8.5) and one where the exact result is given in case the network can be completely computed (Table 8.6). Some regular network structures of larger scale were omitted, since we can infer their results from their smaller scale representatives. Another reason is that we only want to examplarily show the poor performance of the Dotson-Gobien approach for large scale regular structures in contrast to the KLY or decomposition approach. The strength of the Dotson-Gobien approach applies more to unregular randomized structures.
The last column $d(T_\Delta)$ of Table 8.5 indicates the depth or level of the *delta tree* after which the respective bounds are obtained. The file size, the number of tasks and the average disk IO bandwidth are also listed. For Nw.1-4, the bandwidth cannot be measured since the size of the file created is too small.
It can be observed that apart from the number of components, the runtime highly depends on the structure of the network. Comparing the results of Nw.11 and Nw.22 in Table 8.5, their runtimes are roughly the same. However, Nw.22 reveals a three times higher bandwidth than Nw.11. The reason therefore lies in the different graph structures: the shortest *s*-*t* path for Nw.11 is twice as long as the one for Nw.22 which means that Nw.11 generates in general more subproblems in each level than Nw.22. This leads to a higher computation time for processing all tasks for each level. Hence, for Nw.11 more time must be spent to wait for the data to be written on hard disk leading to a lower bandwidth. In the same time, ten levels are processed for Nw.22 whereas only seven levels are finished for Nw.11. Another observation concerning the impact of the graph structure is between

Nw.24 and Nw.18: Nw.24 has 20 components fewer than Nw.18 but we needed about four times longer in order to achieve bounds with the same relative accuracy. Though we start with a lower number of subproblems (length of shortest *s*-*t* path) in Nw.24, this number still remains high after several recursion levels for many child nodes in $T_\Delta$ leading to a high number of tasks for each level. For Nw.24 272 million subproblems are stored in $21.4$ GB which means that in average 84 Bytes are needed to encode a task. Comparing the bandwidths of Nw.22-24, we notice that the average disk bandwidth drops when the network size increases. The simple reason is that it takes more time to perform graph manipulations on larger graphs and more subproblems evolve due to the increasing length of the shortest *s*-*t* path leading to a higher latency. For some networks it was not possible to obtain the exact results within two days. Those that we could finish are listed in Table 8.6. The maximal depth (or the total amount of levels) of the *delta tree* is denoted as $d_{\max}(T_\Delta)$. Note that the depth of the tree is limited by the number of edges of the original input graph since the number of edges decreases by at least one, after each recursion. $d_w(T_\Delta)$ is the broadest level of the tree. The file size of this level also indicates the maximum disk space needed for the whole computation. We can make the following observation by comparing the two tables: for large networks we only need a little fraction of the total time and also of the maximum required disk space in order to reach satisfying bounds. Most of the additional time only contributes to minor improvements of the bounds. For example, for Nw.11 the fraction of disk space required is 0.071 percent and the time spent is only 0.018 percent of the overall time. Similarly, this can be observed for Nw.23. We remark that for even smaller failure probability values, on the order of magnitude of $10^{-5}$ for highly reliable systems, the bounds would be obtained in an even shorter amount of time requiring less hard disk space.
To compare the implementation based on the delta tree and disk storage with an older implementation of the DG method [18] extended with series and parallel reductions, we have added another time column $t_2$ in Table 8.5 and Table 8.6. For Nw.13, 14, 18, 23 and 24 the memory of 2GB was exhausted before the respective bounds could be attained. For all the other networks we can observe that the runtime is shorter for the small sized Nw.1-5, but it takes significantly more time for the larger networks such as Nw.11. This is due to the hold of all subgraphs and performed reductions in memory by the DG method with reductions: for small sized problems less memory is required to store each generated subgraph. Furthermore fewer subproblems evolve. As soon as the problem reaches a certain size, more storage is needed for each subgraph and also more subproblems are to be expected. Consequently, the prohibitively rapid growth of memory demand leads to a negative impact on runtime. The measurements for the randomized structures (Nw. 28-32) are hence only carried out for the modified DG approach. We observe a decrease of of the bandwidth when the interlacement increases. It is more likely that subgraphs of the more planar (less interlaced) graphs contain reducible structures, thus more reduction data is written and fewer subproblems are generated during the computation which results in a lower latency for newly generated workload-data. On the contrary, there are less reducible structures for highly interconnected graphs so that the length of the shortest path or the number of subproblems will only decrease slowly. More time must be spent for computing new workloads or tasks leading to a lower bandwidth.
The demand for memory grows unacceptably high with the size of the networks due to the exponential nature of the terminal-pair reliability problem. This imposes a limiting factor

for reaching good reliability bounds since the computation must be interrupted because of memory shortage. By migrating the memory content to hard disk, the new method allows for coping with larger-sized networks. This method even allows interruptions since the files created until the point of interruption can be reused to continue the computation at a later time. One may assume that by migrating the memory content to hard disk, afterwards the hard disk itself might be a bottleneck. However, by having a look at the measured bandwidth values, this is definitely not the case. On the contrary, the maximal average disk bandwidth of merely 3.90 MB/s shows that there is space for exploiting even further the writing speed of today's hard disks (of around 150 MB/s). This leaves room for further parallelization.

Table 8.5.: Bounds (relative accuracy=0.1)
$t_1$ := runtime for memory-efficient DG method with reductions
$t_2$ := runtime for DG method with reductions
file(MB):= size of generated file in MB
øbw$\left(\frac{MB}{s}\right)$ := bandwidth in MB per second
$d(T_\Delta)$ := level of the delta tree after which the respective bounds are obtained

| Nw | lb | ub | $t_1$ | $t_2$ | file(MB) | #tasks | øbw$\left(\frac{MB}{s}\right)$ | $d(T_\Delta)$ |
|---|---|---|---|---|---|---|---|---|
| *1* | $2.05\cdot10^{-2}$ | $2.10\cdot10^{-2}$ | 0.008 s | 0.005 s | 0.00 | 3 | - | 3 |
| *2* | $2.37\cdot10^{-3}$ | $2.37\cdot10^{-3}$ | 0.007 s | 0.006 s | 0.00 | 0 | - | 3 |
| *3* | $2.26\cdot10^{-2}$ | $2.28\cdot10^{-2}$ | 0.009 s | 0.007 s | 0.00 | 2 | - | 3 |
| *4* | $4.33\cdot10^{-3}$ | $4.33\cdot10^{-3}$ | 0.013 s | 0.010 s | 0.00 | 0 | - | 4 |
| *5* | $4.40\cdot10^{-3}$ | $4.47\cdot10^{-3}$ | 0.282 s | 0.229 s | 0.02 | 368 | 0.30 | 6 |
| *6* | $5.58\cdot10^{-3}$ | $5.61\cdot10^{-3}$ | 0.455 s | 0.485 s | 0.01 | 1,474 | 0.13 | 7 |
| *7* | $2.86\cdot10^{-3}$ | $2.88\cdot10^{-3}$ | 2.00 s | 2.18 s | 0.53 | 8,188 | 1.04 | 8 |
| *8* | $1.25\cdot10^{-2}$ | $1.27\cdot10^{-2}$ | 0.141 s | 0.123 s | 0.01 | 233 | 0.01 | 5 |
| *9* | $1.99\cdot10^{-9}$ | $2.05\cdot10^{-9}$ | 0.68 s | 0.41 s | 0.14 | 1,464 | 0.64 | 11 |
| *10* | $2.42\cdot10^{-2}$ | $2.47\cdot10^{-2}$ | 0.86 s | 0.85 s | 0.41 | 9,171 | 1.22 | 6 |
| *11* | $2.42\cdot10^{-2}$ | $2.47\cdot10^{-2}$ | 6.91 s | 9.80 s | 5.46 | 121,622 | 1.27 | 7 |
| *12* | $2.42\cdot10^{-2}$ | $2.47\cdot10^{-2}$ | 117.64 s | - | 62.54 | 1,275,901 | 0.84 | 8 |
| *13* | $3.81\cdot10^{-2}$ | $3.85\cdot10^{-2}$ | 49.16 s | - | 54.10 | 697,592 | 0.75 | 8 |
| *14* | $4.36\cdot10^{-2}$ | $4.38\cdot10^{-2}$ | 1.39 h | - | 4,899 | 53,184,683 | 0.56 | 8 |
| *18* | $4.34\cdot10^{-3}$ | $4.41\cdot10^{-3}$ | 1.91 h | - | 9,011 | 50,958,418 | 0.78 | 10 |
| *22* | $9.12\cdot10^{-5}$ | $9.14\cdot10^{-5}$ | 7.68 s | 9.73 s | 7.26 | 97,036 | 3.90 | 10 |
| *23* | $1.32\cdot10^{-5}$ | $1.36\cdot10^{-5}$ | 166 s | - | 183.78 | 2,631,226 | 0.60 | 11 |
| *24* | $1.88\cdot10^{-6}$ | $1.91\cdot10^{-6}$ | 7.73 h | - | 21,924 | 272,012,633 | 0.40 | 14 |
| *27* | $4.36\cdot10^{-2}$ | $4.40\cdot10^{-2}$ | 1.96 s | 1.87 s | 0.69 | 3,929 | 1.67 | 8 |
| *28* | | | 65.92 s | - | 47.56 | 99,700 | 1.86 | 10 |
| *29* | | | 97.45 s | - | 138.89 | 850,055 | 2.38 | 10 |
| *30* | | | 1.67 s | - | 1.03 | 9,934 | 1.13 | 7 |
| *31* | | | 0.64 s | - | 0.07 | 1,778 | 0.15 | 7 |
| *32* | | | 190.10 s | - | 8.34 | 204,512 | 0.08 | 8 |

## 8.2. Comparing State-Of-The-Art

Comparing all three approaches, we come to the following conclusions: unlike Hardy's claim (see [23]), we have found that the $D_2$ approach does not outperform the $KLY_2$ approach in general. No fair comparison was given to these two approaches by Hardy, since the approaches were not run on an equivalent architecture. With regard to runtime and memory demand, the $D_2$ approach is, even in comparison with the $KLY_1$ approach, the uncontested clear winner for regular network structures. Networks with bounded path-

Table 8.6.: Exact results (DG method)
$t_1$ := runtime for memory-efficient DG method with reductions
$t_2$ := runtime for DG method with reductions
$d_{\max}$ := maximal depth of the delta tree
$d_w$ := broadest level of the delta tree

| **Nw** | **Unreliability** | $t_1$ | $t_2$ | $d_{\max}$ | $d_w$ | **file(MB)** | **#tasks** | **øbw**$\left(\frac{MB}{s}\right)$ |
|---|---|---|---|---|---|---|---|---|
| ***1*** | $2.07 \cdot 10^{-2}$ | 0.010 s | 0.07 s | 5 | 3 | 0.00 | 3 | - |
| ***2*** | $2.37 \cdot 10^{-3}$ | 0.007 s | 0.006 s | 3 | 1 | 0.00 | 4 | - |
| ***3*** | $2.28 \cdot 10^{-2}$ | 0.11 s | 0.08 s | 5 | 1 | 0.00 | 5 | - |
| ***4*** | $4.33 \cdot 10^{-3}$ | 0.13 s | 0.10 s | 4 | 2 | 0.00 | 7 | - |
| ***5*** | $4.45 \cdot 10^{-3}$ | 0.33 s | 0.28 s | 9 | 5 | 0.17 | 369 | 0.13 |
| ***6*** | $5.60 \cdot 10^{-3}$ | 0.56 s | 0.60 s | 11 | 7 | 0.01 | 1,474 | 0.13 |
| ***7*** | $2.88 \cdot 10^{-3}$ | 2.90 s | 3.15 s | 11 | 8 | 0.53 | 8,188 | 1.04 |
| ***8*** | $2.88 \cdot 10^{-3}$ | 0.29 s | 0.25 s | 11 | 7 | 0.17 | 306 | 0.03 |
| ***9*** | $2.00 \cdot 10^{-9}$ | 9.79 s | 5.96 s | 36 | 20 | 1.65 | 12,574 | 1.99 |
| ***10*** | $2.44 \cdot 10^{-2}$ | 12.50 s | 13.50 s | 15 | 10 | 4.59 | 51,695 | 2.36 |
| ***11*** | $2.43 \cdot 10^{-2}$ | 12.45 h | - | 25 | 16 | 20.429 | 62,358,421 | 2.43 |
| ***13*** | $3.83 \cdot 10^{-2}$ | 0.61 h | - | 22 | 12 | 521.65 | 4,135,084 | 1.25 |
| ***22*** | $9.13 \cdot 10^{-5}$ | 51.75 s | 53.52 s | 20 | 12 | 14.52 | 138,814 | 1.65 |
| ***23*** | $1.33 \cdot 10^{-5}$ | 39.55 h | - | 30 | 20 | 30,613 | 203,132,939 | 1.43 |
| ***27*** | $4.40 \cdot 10^{-2}$ | 3.23 s | 3.10 s | 12 | 8 | 0.69 | 3,929 | 1.67 |
| ***28*** | | 127.82 s | - | 13 | 10 | 47.56 | 99,700 | 1.86 |
| ***30*** | | 7.59 s | - | 13 | 9 | 2.28 | 16,662 | 1.68 |
| ***31*** | | 195.00 s | - | 29 | 19 | 49.86 | 483,366 | 1.71 |

width (maximal frontier size) are computed with extreme efficiency. For instance, the runtime for Nw.19 with the extended KLY approach is 410.640 times the runtime of the $D_2$ approach. Only a fraction of 0.6 percent of the memory demand for the $KLY_1$ is claimed by the $D_2$ approach. The runtime and memory demand for $D_2$ merely grow linearly with the length of grid networks. Comparing these two approaches with the modified DG approach, we observe the DG method's poor performance for regular structures, especially when the length of the shortest *s*-*t* path is high. While the exact result is determined within a few milliseconds and without any noticeable memory demand by the decomposition approach, the bounds of desired accuracy are reached after hours with a high hard disk space demand (see Nw.14 and 18). We note, that by using the original DG approach, these bounds would not at all be reached due to lack of memory. Even more, very large regular structures such as the street network of dimension 12 or the 7x1000 grid are even out of scope, not only for the modified DG approach but also for the $KLY_1$ approach.
On the contrary, when running the $KLY_{1/2}$ and $D_2$ approach on unregular or randomized structures, the $D_2$ approach reveals its weakness as soon as the bfs variable ordering leads to high $|F_{max}|$ values. The $KLY_1$ and $KLY_2$ approach significantly perform better than the $D_2$ approach for the two networks 7 and 27, taken from the literature. For network 27, less than one percent of the memory demand for the $D_2$ approach are claimed by the $KLY_{1/2}$ approach and in addition to this, the $KLY_{1/2}$ approach is 536 times faster. Moreover, the BDD-size is 21 times larger for the $D_2$ approach. Regarding the randomized networks, the situation between the $KLY_{1/2}$ and $D_2$ approach is similar: the difference in runtime and memory demand is enormous for medium-sized networks such as Nw.30 or Nw.28. For smaller-sized structures, where a small $|F_{max}|$ is yielded, all three approaches, $KLY_{1/2}$ and $D_2$, show on average similar performances (see Nw.31). However, for highly interlaced and large-scale structures, such as Nw.29 and 32, $KLY_{1/2}$ and $D_2$ approach find

their limits. At this point, the memory efficient DG approach proves itself valuable. It provides good bounds after a bearable amount of time. For Nw.32, consisting of 100 nodes and more than 200 edges, we needed only 8.34MB of hard disk storage to obtain bounds satisfying the required relative accuracy of 10 percent. The computations conducted by the $KLY_{1/2}$ and $D_{1/2}$ approach fail due to lack of memory and an unknown prohibitive amount of time must be spent to find the exact result.
With the new heuristic, the $D_1$ approach significantly outperforms the $KLY_{1/2}$ in every sense: both for the regular and unregular randomized structures, the runtime and memory demand are many times lower for the $D_1$ approach. For instance, Nw.29 is computable with the $D_1$ approach, whereas $KLY_{1/2}$ and $D_2$ fail. To again emphasize: the $D_1$ approach performs far better than the $D_2$ approach with respect to unregular structures. Yet, Nw.32 could only be adequately handled by the modified DG approach. For overview purposes, all five different approaches, $KLY_{1/2}$ $D_{1/2}$ and modified DG, are roughly ranked according to their performance on regular and unregular network structures (of at least medium size) in Table 8.7.

Table 8.7.: Perfomance ranking ($KLY_{1/2}$ vs. $D_{1/2}$ vs. modified DG)

| Network type | $KLY_1$ | $KLY_2$ | $D_1$ | $D_2$ | modified DG |
|---|---|---|---|---|---|
| regular | 3 | 4 | 1/2 | 1/2 | 5 |
| unregular | 3/4 | 3/4 | 2 | 5 | 1 |

### 8.2.1. Application of the new heuristic to KLY

As mentioned in Chapter 6, the new heuristic can also be applied to other BDD-based k-terminal reliability approaches. Based on the canonicity Lemma 2.6, we expect improvements for other BDD-based approaches only for those cases where a significantly smaller OBDD size is obtained from the decomposition approach with regard to the new heuristic: meaning that $|BDD_{D_1}| \ll |BDD_{D_2}|$. To exemplify the impact of the new variable ordering for the KLY approach, we additionally carried out measurements for sample Nw.27 - the ARPANET: instead of 917,328 we obtained 1031 ROBDD nodes. The runtime for $KLY_1$ is 60ms in contrast to 3.35s and for $KLY_2$ 120ms in contrast to 3.45s. Lastly, the memory consumption for $KLY_1$ and $KLY_2$ amounts to 2.64MB and 3.53MB respectively, in contrast to 63.25MB and 66.10MB. Similarly, improvements in runtime and memory demand were also achieved for the randomized structures in Figure 8.2 and some regular structures listed in Figure 8.1. More measurement results can be found in [42, 41].

# Part V.

# Dependent Component Failures

# 9. Considering dependent failures

In the literature, it is generally assumed that edges or components fail independently. Unfortunately this simplified assumption does not suffice the requirements in reality where dependencies among some components may occur. Among others, those dependencies come up due to common cause failures (*e.g.* growing neighborhood dependency) or fault propagation (see Schneeweiss [81]). Neglecting dependencies may lead to overoptimistic results in the reliability evaluation. Thus, it is a matter of great importance to extend the problem of calculating the network reliability with the additional feature of interdependent component failures. Therefore Walter [79] proposed a hybrid-solution method that combines combinatorial and stochastic models (*e.g.* Markov chains, Petri nets, Gaussian Copula). State-based stochastic models (SBM) such as Markov chains and Petri nets deduce the component interdependencies from the causal dependencies. They are complex mathematical modelling concepts that require a deep understanding from the modeler's point of view. SBM can only be used for relatively small systems since they suffer from the state-space explosion problem. To elude these disadvantages, the Gaussian-Copula representation [54] can be used as an alternative (referring to Walter et.al. [80]). It directly specifies the component dependencies instead of inferring them from the causal dependencies. In this chapter, we show how to appropriately extend the present combinatorial model for considering component interdependencies. First, the dependency relation is formalized after which a formal description of the extended problem is given. With these preliminaries, we accordingly adjust the presented state-of-the-art algorithms (DG with series-parallel reductions, KLY and decomposition approach) for using the Gaussian-Copula as stochastic model.

## 9.1. Preliminaries

According to Walter [79], the set of components $C$ can be partitioned into $k$ disjoint SIC (set of interdependent components) $C_i, 1 \leq i \leq k,\ k \in \mathbb{N}$:

$$C = C_1 \cup C_2 \cup \ldots \cup C_k.$$

The dependency is characterized as a transitive pairwise relation: if two distinct components fail dependently, they must belong to the same SIC $C_i$, for $i \in \{1, 2, \ldots, k\}$. If a component $c \in C$ fails dependently with any component from a SIC $C_i$, then $c \in C_i$. Consequently, the failure of $c$ also depends on the failure of each component in $C_i$ (transitivity). Otherwise, there are no dependencies between two components if they belong to two different SIC $C_i, C_j$ with $i \neq j$. Adhering to this definition, the k-terminal problem with dependent basic events is formulated as follows:

**Statement of the problem** Given a network graph $G := (V, E)$, its terminal node set $K$, a set of system components $C = C_1 \cup C_2 \cup \ldots \cup C_k$, $k \in \mathbb{N}$, $C_i \cap C_j = \emptyset$ for $i \neq j$ and two not-necessarily injective maps $f : E \to C$ and $g : E \to V \times V$. Each component $c \in C$ represents a random variable with two states: failed or working. The reliability for each $c \in C$ is given by $p_c$. For all $C_i \subseteq C$, where$|C_i| > 1$, there exists a corresponding stochastic model. The system's k-terminal reliability $R_K(G)$ is the probability that each pair of nodes from a selected set of nodes $K$ (terminal nodes) can communicate through at least one path of working edges.

### 9.1.1. The Gaussian Copula

A possible combination of working / failed components that lead to system operation / failure corresponds to a conjunction term whose Boolean variables are the respective working / failed components. If all components that occur in this conjunction term are independent, the probability of the conjunction term equals the product of operation/failure probabilities of the contained components.
In case of dependence, the components have to be grouped according to their SIC filiation. Before the probability of the whole conjunction term can be determined, the probability of the conjunction terms for each SIC needs to be computed. This can be carried out by a solver for the used stochastic model. As mentioned above, we make use of a copula-based stochastic model. Its application is based on Sklar's theorem [66] which relates joint distribution functions (probability of component failure combinations) to their marginal distribution functions (component failure probabilities) by means of copulas. When using the Gaussian Copula, a family of copulas, the component dependencies can additionally be specified by a $q \times q$ symmetric correlation matrix $\rho$ with $\rho_{ij} \in [0, 1]$. One only needs to know that there are some components which fail dependently and that their dependencies are of a certain strength. For $\rho_{ij} > 0$, there is a pairwise dependence between two components and independence for $\rho_{ij} = 0$. If $\rho_{ij} = 1$ for two distinct components $c_i$ and $c_j$, then the failure of $c_i$ directly implies the failure of $c_j$. Thus, $c_i$ and $c_j$ can be subsumed to one single component occuring multiple times as multiple edges in the redundancy structure. For the special case when there are no dependencies, $\rho$ equals the identity matrix. We note, that the case of antidependence ($-1 \leq \rho_{ij} < 0$) is omitted here, since we have postulated the system monotonicity which means that the failure of a component cannot increase the reliability of the whole system.
With regard to the Gaussian Copula, $\rho$ is the correlation of the margin probability mass functions transformed to a standard normal distribution. The probability of a conjunction term $CT = x_1 \wedge \ldots \wedge x_k$, where all variables $x_i$ inside $CT$ are elements of the same SIC, can be expressed by means of the Gaussian Copula $C_G$ with respect to network $G$:

$$\begin{aligned}\mathbb{P}(CT) &= C_G(\mathbb{P}(x_1), \ldots, \mathbb{P}(x_k), \rho) \\ &= \Phi^{k,\rho}(\phi^{-1}(\mathbb{P}(x_1)), \ldots, \phi^{-1}(\mathbb{P}(x_k))). \end{aligned} \tag{9.1}$$

Here, $\Phi^{-1}$ is the inverse standard normal distribution with mean zero and variance one. The conjoint probability $\mathbb{P}(CT)$ is obtained by computing $\Phi^{k,\rho}$ - the multivariate standard normal distribution with correlation matrix $\rho$ - which is expressed as a multiple integral. According to Walter [79], Monte-Carlo integrators [43] can be used for its approximation.

At this point we do not want to go further into the stochastic details, but assume that all conjunction terms can be computed as described (see Walter *et al.* [80] for further details).

## 9.2. Extension for the DG with series-parallel reductions approach

In the following, the extension of the DG method for the dependent case is described. Note that the resulting approach only allows for computing two-terminal reliability with dependent component failures. The DG method finds cuts and paths which contribute to the lower and upper bound. Those cuts and paths are conjunction terms consisting of Boolean variables from different SICs. Assume that we have found a cut. Then all edges belonging to the cut will be grouped according to their SICs. For those SICs that have a cardinality greater than one, the grouped Boolean term is stored in a distinguished map called *cutDependent*. The probabilities of the Boolean variables, with SIC size equal to one, are multiplied and the result stored in another map called *cutIndependent*. Both map entries obtain the same index $i$ for the $i$-th cut so that later on the probabilities for the dependent Boolean terms can be queried from a stochastic-based solver such as the Gaussian Copula solver. Now, since all probabilities are given in numeric values, the whole probability of the found cut can be determined by multiplication. Analogously, the same procedure is applied to paths. We refer to [40] for implementational details.
Regarding reductions, we restrict the DG approach to series-parallel reductions. The reason for this is that we want to preserve the DNF for Boolean terms when undertaking reductions between dependent components. Subsequently, the DNF terms are further decomposed by using the inclusion-exclusion method into a sum of disjoint conjunction terms (see Walter *et al.* [80]). The probabilities of those terms can then be obtained from the Gaussian Copula. Other established reduction methods such as the polygon-to-chain [37] or triangle reductions [29] are excluded, since their application does not preserve the DNF.

#### Rules for series and parallel reductions

Under the consideration of dependencies among certain system components we want to sum up the rules and heuristics for series and parallel reductions proposed in one of our works [39]:

- Reductions with multiple components are not allowed except multiple components become unique in the course of the algorithm.
- Reductions can only be performed among components which are a SIC of their own (SIC size equals one) and among components which are from the same SIC (with SIC size larger than one). We call the first case *independent reduction* and the latter *dependent reduction*.

In case of an independent reduction, the probability of the edge resulting from the reduction will be re-adjusted according to the rules for a regular series or parallel reduction (see Section 3.2). In case of a dependent reduction, we relabel one of the two affected edges,

$e_1$ or $e_2$, with a capital letter $R_j$ whereby j stands for the j-th dependent reduction. *W.l.o.g.* we label $e_1$ with $R_j$ and delete $e_2$. $R_j$ comprises the concatenated expression of the two affected edges. Here we introduce the labeling function $l : E \rightarrow B$ where $B$ stands for a boolean expression in DNF - initially, the edges assigned to a dependent component are labeled with the Boolean variable of the respective component. To be more precise, for edges $e_1, e_2 \in C_k,\ k \in \mathbb{N}$ , we now define $R_j = l(e_1) \wedge l(e_2)$ for a series reduction and $R_j = l(e_1) \vee l(e_2)$ for a parallel reduction. Hereafter $e_2$ will be deleted and $l(e_1) = R_j$. For details of the algorithmic approach we refer to [39].

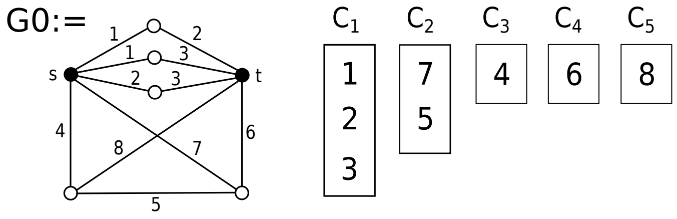


Figure 9.1.: Initial graph.

### 9.2.1. Case study

Now we describe the workings of the introduced extensions for the DG approach by means of the example in Figure 9.1. The set of components $C$ is made up of five SICs: $C_1$ and $C_2$ are the only two SICs with size larger than one. All other components beyond $C_1$ and $C_2$ are regarded as independent. In accordance with these, $\rho$ is an $8 \times 8$ correlation matrix with ones on its main diagonal. The entries $\rho_{1,2} = \rho_{2,1}, \rho_{1,3} = \rho_{3,1}, \rho_{2,3} = \rho_{3,2}, \rho_{5,7} = \rho_{7,5}$ take different values $\alpha \in \mathbb{R}$, with $0 < \alpha < 1$. The rest of the matrix is filled with zeros.
*W.l.o.g.* the edges are labeled by natural numbers representing Boolean variables or components. There is a 2-out-of-3 edge modeled by multiple edges assigned to components 1, 2 and 3. The two terminal nodes $s$ and $t$ are marked in black. Because there are no possible reductions, we start with looking for a shortest path. The algorithm delivers for example path $1 \wedge 2$ (Figure 9.2 (a)). In the next step we would obtain two subgraphs: $G1$ by deleting edges labeled with component 1. $G2$ by contracting edges assigned to component 1 and deleting edges assigned to component 2. Again, for each of those subgraphs we try to reduce. We notice that a series reduction can be made between the edges labeled with 2 and 3, because 2 and 3 are in the same SIC. One of the edges will be labeled as $R_1$ and the other deleted. We store the reduction made in our edge-probability-map for graph $G1'$ which is a complete graph containing four nodes[1]. Now we continue to search for the shortest path which is obviously $R_1$. After deleting $R_1$ (Figure 9.2 b), we would obtain $G3$. Normally the algorithm would proceed with subgraph $G2$ which has the same structure as $G1'$. Hence we omit the sketch of processing $G2$, nevertheless it can be reconstructed by the help of Figure 9.3. Continuing with $G3$, we look for a shortest path since no parallel or series reductions are possible. Partitioning the graph on the base of shortest path $4 \wedge 8$ we arrive at $G5$ and $G6$. Though $G5$ is a series structure, we are not allowed to reduce because components 7 and 6 are not from the same SIC. Proceeding on the basis of the shortest path $7 \wedge 6$

[1]In the literature this type of graph is also known as "K4 graph".

we obtain two cuts since the terminal nodes are disconnected. The cuts are highlighted in the dotted boxes of Figure 9.3. We climb up the recursion tree to go on with G6. There we can do a dependent parallel reduction, since components 5 and 7 are from the same SIC. The reduction is captured in a separate map as $R_2 = 5 \vee 7$. Again, we obtain two cuts based on the shortest path $R_2 \wedge 6$. The recursion tree in Figure 9.3 illustrates all possible paths and cuts obtained in each depth/level of the recursion. As mentioned before, the Boolean expressions within cuts and paths are rearranged and grouped according to their SIC filiation. The grouped terms are stored separately in the maps *pathDependent* or *cutDependent* to be handed over to the Gaussian Copula. The probabilities of the independent expressions are simply multiplied and then added to maps *pathIndependent* or *cutIndependent*. After the probabilities of the dependent terms were returned from the Gaussian Copula, the whole probability for any path/cut term can be reassembled by multiplying, since the SICs are independent among each other. For instance, the relevant probabilities of the first cut term $!1 \wedge R_1 \wedge !4 \wedge !7$ would be classified as follows: The value of $p_{!4}$ would be added to map *cutIndependent* at the first index whereas $!1 \wedge R_1$, belonging to $C_1$, and $!7$, belonging to $C_2$, would be added to the first position of map *cutDependent*. Analogously the second cut would be stored at the second position of the respective map. When all values for the dependent basic events are returned from the Gaussian Copula, the probability for the first cut is computed as $p_{!4} \cdot p_{!1 \wedge R_1} \cdot p_{!7}$. For implementational details, we refer to [40].

### 9.2.2. Complexity considerations

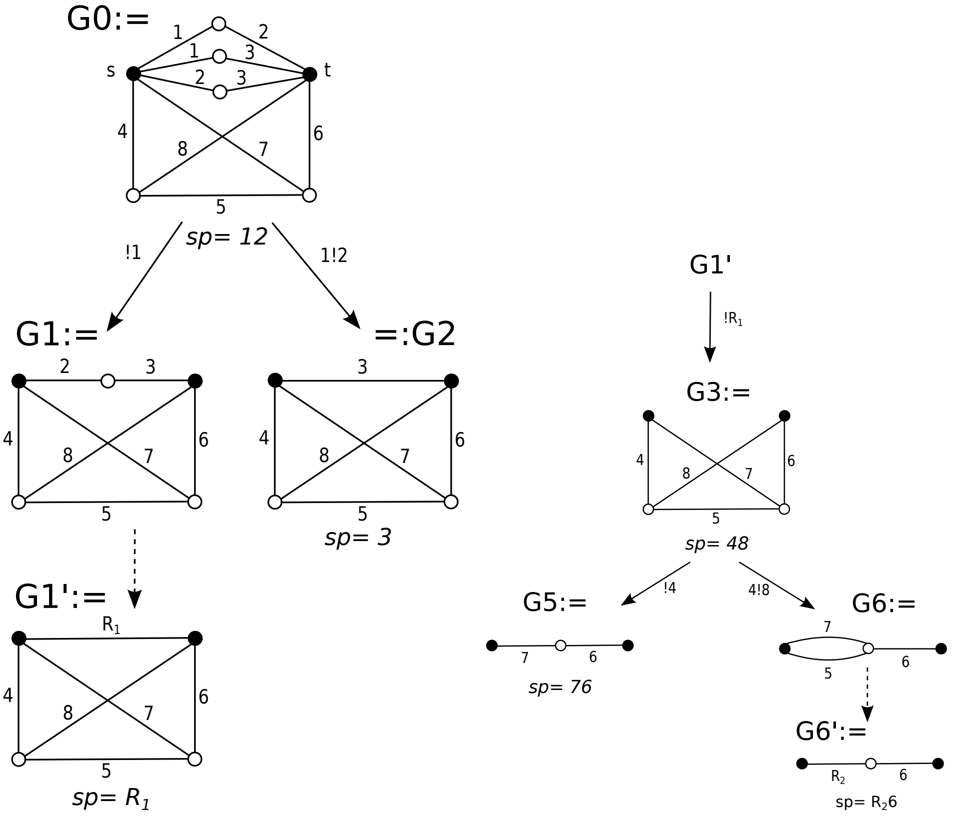
It is clear that the runtime and memory demand for the dependent case is at least as much as the independent case. The memory demand grows with the storage of dependent conjunction terms in the dependent maps. Furthermore, the performed dependent reductions must be captured in separate maps. The runtime increases with the rearrangement and classification of the Boolean terms (paths and cuts), the decomposition of the DNF terms, the simplification of conjunction terms with negated variables[2] and the computation of the multiple integral. However, in practical applications we hope to expect only a small number of dependent subsets so that the computation of the extended problem does not take much more time than the original problem.

## 9.3. Extension for BDD-based approaches

In what follows, we explain how to appropriately extend BDD-based approaches, such as the KLY or decomposition approach, in order to consider the described dependencies. In principle, we do not need to change the algorithms. Instead, the following four major steps need to be done:

1. First, we need to rearrange the preprocessed variable order (*e.g.* bfs-order or new-heuristic order) so that variables from the same SIC occur consecutively in the variable order. This step is realized by Algorithm 13 (*sortVarorder*) which is applied to an existent variable order list, called $L$. As a result, a rearranged list $L'$ is returned. Based on the rearranged variable order in $L'$, the algorithms are performed on the input graph as usual.

[2]See [80] for details.



(a) First level (b) Second and third level

Figure 9.2.: Algorithm

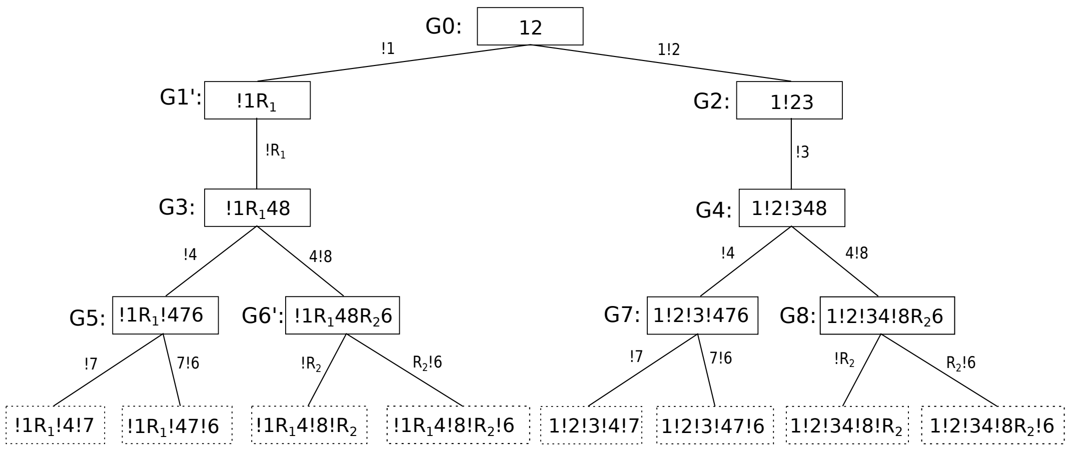


Figure 9.3.: Recursion Tree.

2. The resulting BDD is replicated to a MDD (Multiple Decision Diagram). This data structure contains nodes (MDD nodes labeled with "M") having more than two outgoing directed edges that are labeled with the respective conjunction terms. Those edges are created by traversing SIC layers (sequence of consecutive BDD nodes representing dependent variables). The nodes representing independent variables (with SIC size one) and their two outgoing edges are merely copied. We start the replication with Algorithm 15 which initially takes the result BDD root-node, a newly created MDD root-node and the true-value as input. It iteratively creates the MDD by traversing the result BDD in dfs-order. The $addNode$ function is responsible for MDD node creation. Its second argument is the label (Boolean term) of the MDD edge pointing to the first argument, the MDD child node. To avoid redundant MDD node creation, $addNode$ keeps track of already created nodes by the help of a hashmap. If a node was already created it is returned by $addNode$.

3. After the creation of the MDD, the missing probability values for the labeled MDD edges are queried from the Gaussian Copula solver so that the complete information can then be provided for obtaining the seeked reliability value. This is done by passing on the $ToSolver$ list to the Gaussian Copula solver.

4. Finally, we can compute the numeric reliability value by applying Algorithm 14 (*Prob2*) to the MDD root-node.

### 9.3.1. Case study

To get a better idea of the described extensions, we perform the KLY approach using bfs variable order on the example graph from Figure 9.1. A possible bfs variable order would be $1, 2, 4, 7, 3, 5, 6, 8$. Then, by applying Algorithm 13 we obtain the rearranged version $[1, 2, 3], 4, [7, 5], 6, 8$. As we can see, the SIC variables are now placed in a consecutive order. The two SIC layers are highlighted in square brackets. The KLY approach creates the ROBDD (see left half of Figure 9.4) representing the system structure function:

$$X_G = (1 \wedge (2 \vee 3)) \vee (2 \wedge 3) \vee (7 \wedge (6 \vee 5 \wedge 8)) \vee (4 \wedge (8 \vee 5 \wedge 6)).$$

Note that we could also have used the decomposition approach or other BDD-based approaches instead. In any case, the resulting BDD would be the same.
Now the result ROBDD is replicated by Algorithm 15 to ROMDD (left side of Figure 9.4). The conjunction terms labeled at the "M"-edges are collected in the $ToSolver$ list and their probabilities are resolved by the Gaussian Copula. Afterwards, the seeked reliability value can be determined by applying Algorithm 14 on ROMDD root-node M1.

### 9.3.2. Complexity discussion

Similar to the previous complexity discussion for the dependent DG approach, the complexity of a BDD-based approach is increased by the following overheads: the overhead for resorting the variable order is negligible. However, resorting the variable order may leave a significant impact on the performance of the BDD-based approach. As shown by the measurements in Chapter 8, the impact can be drastic - both negative and positive in terms of runtime and memory demand - indicated by the resulting BDD-size. Furthermore, the additional overhead for replicating the result BDD is, in the worst case, linear in the size of the result BDD: if all components are dependent, *i.e.* they belong to one and the same SIC, then the resulting MDD would consist of one root-node, two leafs (1 and 0) and a set of labeled edges equal to the number of all possible directed root-to-leaf paths of the result BDD. This number can be determined in time linear to the number of BDD edges [9]. The traversal of the result MDD with function $Prob2$ also claims time linear in the number of MDD edges. Finally, we have to compute the joint probability for each labeled MDD-edge: in contrast to the DG method with series-parallel reductions, no decomposition of DNF terms needs to be conducted. This means that another considerable effort, which is exponential in the worst case is spared. The time complexity for simplifying each conjunction term that contains negated variables, is linear in the number of negated variables. Lastly, the total overhead for determining the probabilities of the resolved conjunction terms depends on the used Monte-Carlo integrator. For example, one can use the method in [43]: it is efficient even in high dimensions[3], since the method's overhead and memory requirements grow only linearly with dimension.

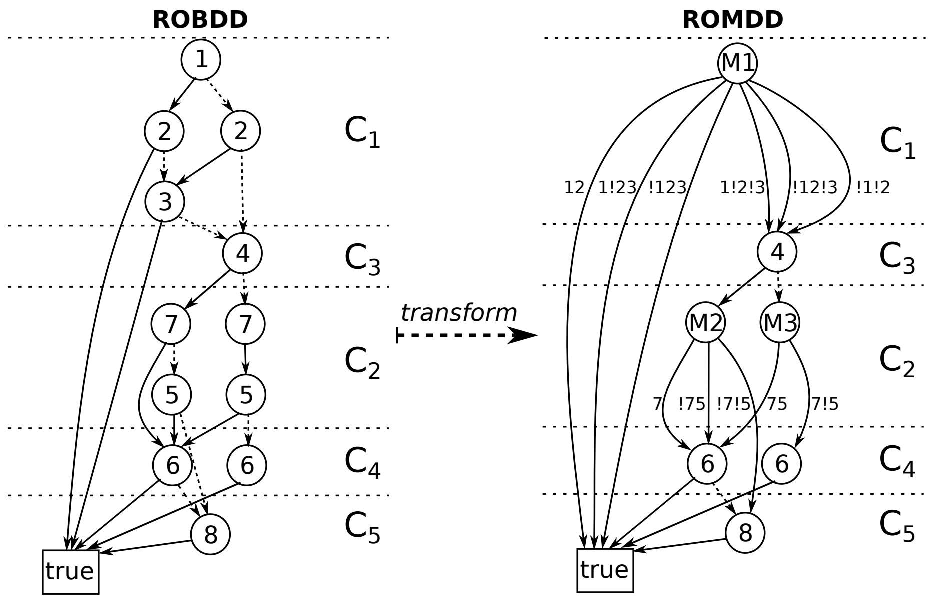


Figure 9.4.: Referring to the example from Figure 9.1, ROBDD (Left), ROMDD (Right).

[3]The dimension of the multiple integral equals the number of different variables in the resolved conjunction term.

Algorithm 13: sortVarorder

```
Require: List $L$
1: $shift = 1,\ C' = \emptyset$
2: for $i = 1 \to |L|$ do
3:    if $L(i) \in C_k$ and $|C_k| > 1$ then
4:       if $L(i) \in C'$ then
5:          continue for-loop
6:       end if
7:       for $j = 0 \to |C_k| - 1$ do
8:          $L'(i + j) = C_k(j)$
9:       end for
10:      $C' = C' \cup C_k,\ shift = shift + |C_k|$
11:   else
12:      $//L(i) \in C_k$ and $|C_k| = 1$
13:      $L'(shift) = L(i),\ shift = shift + 1$
14:   end if
15: end for
16: return $L'$
```

Algorithm 14: Prob2

```
Require: MDDNode $m$
1: if $m == true$ then
2:    return 1
3: end if
4: if $m == false$ then
5:    return 0
6: end if
7: if $result = hm.find(m) \neq NULL$ then
8:    return $result$
9: else
10:   if $m$ has label "M" then
11:      for all $m$-adjacent edges $e := (m, m')$ do
12:         $result = result + p_e \cdot Prob2(m')$
13:      end for
14:   else
15:      $//m$ is a binary node, $p$ is the reliability of the component represented by $m$
16:      $result = p \cdot Prob2(m.high) + (1 - p) \cdot Prob2(m.low)$
17:   end if
18:   $hm.put(m, result)$
19: end if
```

Algorithm 15: replicate

```
Require: BDDNode $b$, MDDNode $m$, BoolExpr $acc$
if $b$ is independent then
   if $b.high == true$ then
      $m.high = 1$
   else
      $m_h = m.addNode(b.high, b)$ //$addNode$ returns MDDNode if already existent
      $replicate(b.high, m_h, 1)$
   end if
   if $b.low == false$ then
      $m.low = 0$
   else
      $m_l = m.addNode(b.low, \overline{b})$
      $replicate(b.low, m_l, 1)$
   end if
else
   for $b' = b.high$ and $b' = b.low$ do
      if $b' == b.high$ then
         $acc = acc \wedge b$
      else
         $acc = acc \wedge \overline{b}$
      end if
      if $b' == true$ or $b' == false$ then
         $m.addLeaf(acc)$ //Adds edge labeled with $acc$ directing to leaf node
         $ToSolver.add(acc)$ //add conjunction term $acc$ to $ToSolver$ list
      else
         //Preview whether creation of MDDNode is possible
         if $SICIndex(b)! = SICIndex(b')$ then
            $m' = m.addNode(b', acc)$ //creation of MDDNode $m'$
            $ToSolver.add(acc)$
            $replicate(b', m', 1)$
         else
            $replicate(b', m, acc)$
         end if
      end if
   end for
end if
```

# Part VI.

# Conclusion and Outlook

# 10. Summary

Before summing up this thesis, we want to refer back to the gas supply network example from Section 1.1. In that what follows, possible ways are discussed to determine the reliability of this gas supply network, both with and without the influence of seismic hazard.
Up to this point, we also have accumulated enough background knowledge in network reliability evaluation in order to create an appropriate abstraction for a possible real-world gas supply network (see Figure 10.1). Here the pipelines are modelled as edges, distributors and supplier are represented by nodes. Under the assumption that only pipelines can fail and that their failures are independent in the absence of seismic hazard, the terminal-pair reliability problem can be efficiently solved by one of the suggested state-of-the art methods. For instance, we can stipulate $s$ and D2 to be the terminals. Since all edges fail independently, there would be seven single-element sets of interdependent components. With this graph model, the k-terminal problem can be answered by using the KLY or decomposition method. Even if we assume that distributors and supplier do not operate perfectly, the k-terminal problem with node failures can be efficiently solved by the help of the KLY and decomposition method.
In case of an earthquake, pipelines P5, P6 and P7 are claimed to be spatially correlated. Hence, the respective edges 5, 6 and 7 are subsumed into one SIC C5. Applying the techniques described in the previous chapter, the two-terminal problem for the dependent case can be efficiently solved. However, the k-terminal problem with dependent basic events can only be solved with the extended KLY and decomposition method.
We take another step of model refinement and assume that the nodes may also fail for the dependent case. Then D2 and D3 would be added to SIC C5. Using the KLY and decomposition method for imperfect nodes, we can compute the BDD for the independent case. Afterwards, the k-terminal reliability for dependent failures can be obtained by applying the extension for BDD-based methods, explained in the previous chapter.

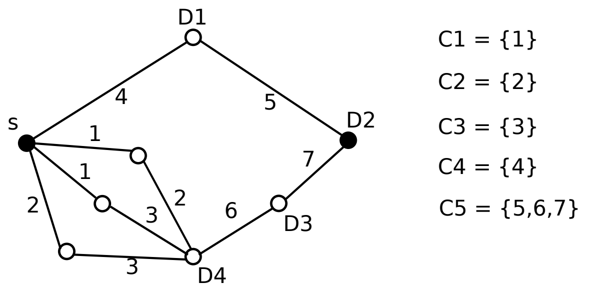


Figure 10.1.: Graph model for the gas supply network under seismic hazard.

## 10.1. Extensions and modifications

Once again, the example raises several important issues that need to be considered in the evaluation of network reliability: imperfect nodes, multiple edges, k-terminal connectedness and intercomponent dependencies. In this thesis, we have first given a chronological overview over the state-of-the-art exact methods that are restricted to independent component failures. The state-of-the-art further provides appropriate extensions to handle node failures with small additional overhead. An extensive survey on bounding algorithms was given in Chapter 4: both for models assuming equal and arbitrary component failure probabilities. It turns out that the two BDD-based methods, the KLY and decomposition method, are currently the most efficient exact methods for two-terminal reliability of undirected networks. Both methods allow arbitrary component failure probabilities, also node failures and k-terminal connectedness can be considered. When restricted to networks of bounded tree-width, the decomposition method is indisputably the most efficient method: its time complexity is then linear in the size of the network. Nevertheless, the decomposition method only works for undirected networks. In contrast to that, the KLY method can also handle directed networks since path information is stored.
With regard to bounds, the DG method with reductions shows to be the most efficient bounding algorithm for two-terminal network reliability with arbitrary component failure probabilities: tight bounds can be rapidly obtained in only a fraction of the total computation time. If the computation can be completed the exact result is obtained. The DG method can also handle node failures and directed networks. However, only the two-terminal case can be treated by the DG method. Our probabilistic graph model allows for considering multiple edges in all three mentioned methods: KLY, decomposition and DG. Each method's compatibilities towards all four aspects are again summed up in the following Table 10.1.

Table 10.1.: KLY, decomposition and DG method compatibility towards modelling aspects for the independent case.

| **Method** | **k-terminal** | **imperfect nodes** | **multiple edges** | **directed networks** |
|---|---|---|---|---|
| **KLY** | ✓ | ✓ | ✓ | ✓ |
| **Decomposition** | ✓ | ✓ | ✓ | – |
| **DG** | – | ✓ | ✓ | ✓ |

The main reasons that lead to the efficiency of the KLY and decomposition method are the fast recognition of isomorphic subproblems which avoids redundant computations, the locating of mutually disjoint relevant states which oviates another possibly expensive set disjointing process, and the use of BDD for compactly storing the structure function of the network. Nevertheless, we sought for ways to further improve those two methods. In Chapter 5 we have revealed that the KLY method performs redundant computations in the presence of redundant biconnected components. Those structures may occur during the computation process and can be recognized in linear time. On certain network structures, redundant biconnected structures occur more frequently *e.g.* on grid networks or sparse network structures. For significantly improving the KLY in terms of runtime and memory demand, we have shown how to efficiently recognize and remove redundant biconnected structures. This also has direct improvement implications for the k-terminal version of the KLY method along with node failures. To highlight this on the example of a 3x20 grid

network: a speedup of 464 times was reached with the extended KLY and the memory consumption amounts to only 0.3 percent of the memory that was claimed by the original KLY method. In the worst case when no redundant biconnected components occur, the runtime of the new approach is only extended by the recognition overhead and the memory demand remains unchanged.
The time and space complexity of the decomposition method highly depends on the cardinality of the maximal boundary set $F_{max}$, which in turn depends on the chosen variable ordering. Since finding an optimal variable ordering is an NP-hard problem, one has to resort to heuristics for finding a good variable ordering in an acceptable amount of time. In the literature, the bfs heuristic is empirically chosen to be a good heuristic. However, we have developed in Chapter 6 a new heuristic which leads for many examples to much lower $|F_{max}|$. This "greedy" heuristic can also be embedded in other BDD-based methods such as the KLY method. As a result, the size of the result BDD significantly shrinks in comparison to the BDD obtained by the bfs heuristic, and this is immediately noticeable in terms of runtime and memory demand. Hence, the new heuristic leads to better performances for many networks such as the ARPANET from 1979. There a speedup of around 18,000 times is gained and less than 0.1 of the memory is needed with the application of the new heuristic. Nevertheless, we also showed examples where the bfs heuristic is to be preferred. One obtains slightly better performances with the bfs heuristic on complete networks and grid networks, though the same $|F_{max}|$ is generated by both heuristics. This is due to the fact that the distribution of the boundary set sizes differ (see Section 6.2.2).
Due to the exponential nature of the terminal-pair reliability problem, the demand for memory grows unacceptably with the size of the networks to be assessed. To improve the quality of reliability bounds, we must be very economical with memory resources. Therefore we have optimized the GD method in terms of memory consumption in Chapter 7. The requirements for memory consumption could be drastically decreased by an introduced data structure called delta-tree. Instead of storing subgraphs, the delta-tree keeps track of the changes made to the original network graph and the respective subproblems can be reconstructed by the help of the delta-tree. We further suggested to migrate the memory content to hard disk in order to cope with even larger networks and obtain tighter bounds. Moreover, the measurements in Section 8.1.3 proved that the migration of the memory content to hard disk does not constitute a bottleneck.
To cope with the problem of dependent component failures, we have reformulated the two-terminal reliability problem with regard to the introduced SIC (see Chapter 9). Subsequently, the three improved methods were given appropriate modifications in order to allow the consideration of dependent basic events. To put it briefly: for the purpose of considering dependent component failures, we have proposed suitable modifications for the combinatorial methods to efficiently interact with a stochastic based model, in this case the Gaussian Copula, in a hybrid context. Also for the case of dependency, the compatibilities of the three algorithms towards the mentioned modelling aspects are illustrated in the following Table 10.2.

Table 10.2.: KLY, decomposition and DG method compatibility towards modelling aspects for the dependent case

| Method | k-terminal | imperfect nodes | multiple edges | directed networks |
|---|---|---|---|---|
| KLY | ✓ | ✓ | ✓ | ✓ |
| Decomposition | ✓ | ✓ | ✓ | – |
| DG | – | – | ✓ | ✓ |

## 10.2. Choosing the right algorithm

To prove that the discussed extensions and modifications are justified, we have compared the improved versions of the KLY, decomposition and GD method with their original version in a large-scale measurement series comprising typical benchmark networks from the literature, regular networks and four sets of randomized structures of different scale (see Chapter 8). In addition, all three methods were empirically compared against each other in order to conclude their efficiency for arbitrary network structures. The execution of all three implementations on the same architecture allows for a fair comparison. Such a comparison has not yet been found in the literature, when the KLY and decomposition method (in their original version) were compared based on runtimes obtained from different computer architectures [25]. Hence the conclusion that the decomposition method performs better than the KLY method for general networks is not legitimate. Our measurement results reveal that the decomposition method is only superior in regular networks of bounded tree-width. However, the KLY method shows better performances on many unregular structures. There the bfs heuristic often causes high $|F_{max}|$ values rendering the decomposition method inefficient. Only if the new heuristic is applied to both the KLY and decomposition method, we can conclude that with respect to undirected networks, the decomposition method is superior to the original and improved KLY method (see Table 8.7). Based on the following guidelines, a recommendation for the application of a respective method can be inferred: undoubtedly, the decomposition method is to be favored for regular structures with bounded tree-width. Also, with regard to the new heuristic, priority should generally be given to the choice of the decomposition method over the KLY. Unfortunately, the decomposition method only works for undirected networks. Hence, the KLY method turns out to be very useful for regular directed structures and with this regard it is the currently most efficient method. Although the DG method performs quite weakly on regular networks, the measurements justify its indispensability for unregular and randomized network structures of larger scale. While the DG method covers both undirected and directed networks, it unfortunately lacks the ability to determine the k-terminal reliability. Furthermore, imperfect nodes cannot yet be considered in case of dependency (see Table 10.2).

## 10.3. Future research

Finally, we want to point out future research directions that arise from this thesis and the above discussions: as we can see, the technique of dynamic programming avoids redundant computations and is one of the key ideas that lead to significant improvements for BDD-based approaches. Further improvements can be obtained by finding a better variable ordering which leads to an equivalent (in terms of the reliability measure) but smaller result BDD. This means that the number of relevant *full events* (sets of system states) in the smaller result BDD is generally fewer than the number of relevant full events (root-to-leaf-paths) in the equivalent larger BDD (with respect to the worse variable ordering). Hence, the full events in the smaller BDD must be composed of more *elementary events*. In other words, those full events (sets) constitute a larger fraction of the probability space and the number of those respective fractions is fewer. This shows that we have to focus on finding techniques which yield fewer relevant full events, comprising preferably lots of elementary events. For the BDD-based approaches, this can be done by further improving the heuristics for variable ordering.

Finding better variable orderings also means finding tree decompositions of smaller tree-width. However, computation is not practically feasible for graphs with tree-width larger than 16. To obtain at least any valuable results, we have to even more consider how to increase the efficiency of bounding algorithms. Unlike BDD-based methods, the GD method does not make use of dynamic programming. Thus, we have to find ways how to appropriately incorporate dynamic programming into the DG method. This would certainly lead to a significant improvement of bounds. To cover all important modelling aspects, future efforts must be dedicated towards conceiving efficient bounding algorithms for k-terminal reliability for both independent and dependent case. This could be done by using similar strategies which were just mentioned. Furthermore, bounding algorithms should be able to consider imperfect nodes when components may fail dependently. Lastly, it is desirable to extend the generality of the decomposition approach, meaning that it should be also applicable to directed networks.

# Bibliography

[1] J. A. Abraham. An improved algorithm for network reliability. In *IEEE Trans. Reliability, vol.R-28, no.1, 58-61*, 1979.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullmann. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[3] H. R. Andersen. *An Introduction to Binary Decision Diagrams*. Lecture notes for advaced algorithms, 1997.

[4] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. In *SIAM Journal on Algebraic and Discrete Methods, vol.8, no.2, 277-284*, 1987.

[5] S. Arnborg, D. G. Corneil, and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to k-trees. In *Discrete Applied Mathematics vol.23, no.1, 11-24*, 1989.

[6] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press New York, 2009.

[7] M. O. Ball, C. J. Colbourn, and J. S. Provan. *Network reliability*. Handbook of Operations Research: Network Models, Elsevier North-Holland, 673-762, 1995.

[8] M. O. Ball and J. S. Provan. Bounds on the reliability polynomial for shellable independence systems. In *SIAM Journal on Algebraic and Discrete Methods, vol.3, no.2, 166-181*, 1982.

[9] M. O. Ball and J. S. Provan. Calculating bounds on reachability and connectedness in stochastic networks. In *Networks, vol.13, no.2, 253-278*, 1983.

[10] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. In *SIAM Journal on Computing, vol.25, no.6, 1305 - 1317*, 1996.

[11] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Trans. Reliability, vol.35, no.8, 677-691*, 1986.

[12] J. G. Carlier and C. Lucet. A decomposition algorithm for network reliability evaluation. In *Discrete Applied Mathematics, vol.65, no.1, 141-156*, 1996.

[13] Y. Chen, A. Q. Hu, K. W. Yip, X. Hu, and Z. G. Zhong. A modified combined method for computing terminal-pair reliability in networks with unreliable nodes. In *Machine Learning and Cybernetics, vol.4, 2426-2429*, 2003.

[14] C. J. Colbourn. Edge-Packings of Graphs and Network Reliability. In *Discrete Mathematics vol.72, 49-61*, 1988.

[15] C. J. Colbourn and D. D. Harms. Bounding All Terminal Reliability in Computer Networks. In *Networks, vol.18, no.1, 1*, 1988.

[16] C. J. Colbourn, L. D. Nel, T. B. Boffey, and D. F. Yates. Network reliability and the probabilistic estimation of damage from fire spread. In *Annals of Operations Research, vol.50, no.1, 173-185*, 1994.

[17] N. Deo and M. Medidi. Parallel algorithms for terminal pair reliability. In *IEEE Trans. Reliability, vol.41, no.2, 201-209*, 1992.

[18] W. P. Dotson and J. Gobien. A new analysis technique for probabilistic graphs. In *IEEE Trans. Circuit & Systems, vol.26, no.10, 855-865*, 1979.

[19] J. Edmonds. *Optimum branchings*. J. Res. Nat. Bur. Standards, vol.71, 233-240, 1967.

[20] G. S. Fishman. A Comparison of Four Monte Carlo Methods for Estimating the Probability of s-t Connectedness. In *IEEE Trans. Reliability, vol.35, no.2, 145-155*, 1986.

[21] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *SIAM Journal on Computing, vol.39, no.5, 710-713*, 1990.

[22] R. E. Gomory and T. C. Hu. Multi-terminal network flows. In *Journal of the Society for Industrial and Applied Mathematics, vol.9, no.4, 551-570*, 1961.

[23] G. Hardy, C. Lucet, and N. Limnios. K-terminal network reliability measures with Binary Decision Diagrams. In *IEEE Transactions on Reliability, vol.56, no.3, 506-515*, 2007.

[24] D. D. Harms. An investigation into bounds on network reliability. Department of Computational Science, University of Saskatchewan, 1983. [Master's Thesis].

[25] J. U. Hermann. Improving Reliability Calculation with Augmented Binary Decision Diagrams. In *Advanced Information Networking and Applications (AINA), 328-333*, 2010.

[26] J. U. Hermann and S. Soh. Comparison of binary and multi-variate hybrid decision diagram algorithms for k-terminal reliability. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference (ACSC), vol.113, 153-162*, 2011.

[27] J. Hopcroft and R. Tarjan. Dividing a graph into triconnected components. In *SIAM Journal on Computing, vol.2, no.3, 135-158*, 1972.

[28] J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. In *Communications of the ACM vol.16, no.6, 372-378*, 1973.

[29] S. J. Hsu and M. C. Yuang. Efficient computation of terminal-pair reliability using triangle reduction in network management. In *ICC on Communications, vol.1, 281-285*, 1998.

[30] H. Imai, K. Sekine, and K. Imai. Computational investigations of all-terminal network reliability via BDDs. In *IEICE Trans. Fundamentals, vol.82A, no.25, 714-721*, 1999.

[31] I. M. Jacobs. *Connectivity in probabilistic graphs*. M.I.T. Dept. of Electrical Engineering, 1959.

[32] P. Jensen and M. Bellmore. An algorithm to determine the reliability of a complex system. In *IEEE Trans. Reliability, vol.R-18, no.4, 167-174*, 1969.

[33] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations, Plenum Press, 85-103*, 1972.

[34] A. K. Kel'mans. Connectivity of probabilistic networks. In *Automation and remote control, no.3, 98-116*, 1967.

[35] J. Kohlas. *Zuverlässigkeit und Verfügbarkeit: mathematische Modelle, Methoden und Algorithmen*. B.G. Teubner, 1987.

[36] S. Kuo, F. Yeh, and H. Lin. Efficient and Exact Reliability Evaluation for Networks With Imperfect Vertices. In *IEEE Transactions on Reliability, vol.56, no.2, 288-300*, 2007.

[37] K.Wood. A factoring algorithm using polygon-to-chain reductions for computing k-terminal network reliability. In *Networks, vol.15, no.2, 173-190*, 1985.

[38] D. R. Fulkerson L. R. Ford. *Flows in networks*. Princeton University Press, 1962.

[39] M. Lê and M. Walter. Considering dependent components in the terminal pair reliability problem. In *DYADEM-FTS 2011, 415-422*, 2011.

[40] M. Lê and M. Walter. Bounds for Two-Terminal Network Reliability with Dependent Basic Events. In *MMB'12/DFT'12 Proceedings of the 16th international GI/ITG conference on Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance, 31-45*, 2012.

[41] M. Lê, M. Walter, and J. Weidendorfer. Improving the Kuo-Lu-Yeh algorithm for assessing Two-Terminal Reliability. In *European Dependable Computing Conference (EDCC), Newcastle Upon Tyne*, 2014.

[42] M. Lê, J. Weidendorfer, and M. Walter. A Novel Variable Ordering Heuristic for BDD-based k-terminal Reliability. In *Dependable System Networks (DSN), Atlanta*, 2014.

[43] G. P. Lepage. A new algorithm for adaptive multidimensional integration. In *Journal of Computational Physics vol.27, no., 192-203*, 1978.

[44] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy`, 2003. [Online].

[45] M.O. Locks. Inverting and minimizing boolean functions, minimal paths and minimal cuts: Noncoherent system analysis. In *IEEE Trans. Reliability, vol.R-28, no.5, 373-375*, 1979.

[46] M. V. Lomonosov and V. P. Polesskii. Lower bound of network reliability. In *Problems of Information Transmission, vol.8, no.2, 47-53*, 1972.

[47] M. V. Lomonosov and V. P. Polesskii. An upper bound for the reliability of information networks. In *Problems of Information Transmission, vol.7, 337-339*, 1972.

[48] C. Lucet. Méthode de décomposition pour l'évaluation de la fiabilité des reseaux. Université de technologie de Compiègne, 1993. [PhD Thesis].

[49] C. Lucet and J. F. Manouvrier. Exact methods to compute network reliability. In *MMR*, 1997.

[50] D. Moelle, S. Richter, and P. Rossmanith. A faster algorithm for the Steiner tree problem. In *LNCS, vol.3884, 561-750*, 2006.

[51] E. F. Moore. The shortest path through a maze. In *International Symposium on the Theory of Switching & Annals of the Computation Laboratory of Harvard University, 285-292*, 1959.

[52] E. F. Moore and C. E. Shannon. Reliable circuits using less reliable relays. In *Journal of the Franklin Institute, vol.262, no.3*, 1956.

[53] C.St.J.A. Nash-Williams. Edge disjoint spanning trees of finite graphs. In *J. London Math. Soc. 36, 445-450*, 1961.

[54] R. B. Nelsen. *An Introduction to Copulas*. Springer, 1999.

[55] V. P. Polesskii. A lower boundary for the reliability of information networks. In *Problems of Information Transmission, vol.7, 165-171*, 1972.

[56] J. S. Provan and M. O. Ball. The complexity of counting cuts and computing the probability that a graph is connected. In *SIAM Journal on Computing, vol.12, no.4, 777-788*, 1983.

[57] J.S. Provan. The complexity of reliability computations in planar and acyclic graphs. In *SIAM Journal on Computing vol.15, no.3, 694-702*, 1986.

[58] S. Rai and K.K. Aggarwal. An efficient method for reliability evaluation of a general network. In *IEEE Trans. Reliability, vol.R-27, no.3, 206-211*, 1978.

[59] V. Raman. Finding the best edge-packing for two-terminal reliability is NP-hard. In *Journal of Combinatorial Math. and Comb. Computing, vol.9, 91-96*, 1991.

[60] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. In *Journal of Algorithms, vol.7, no.3, 309-322*, 1986.

[61] A. Rosenthal. Computing the reliability of complex networks. In *SIAM Journal on Applied Mathematics, vol.32, no.2, 384-393*, 1994.

[62] R. Sahner, K. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems*. Kluwer Academic Publishers, 1996.

[63] A. Satyanarayana and M. K. Chang. Network reliability and the factoring theorem. In *Networks, vol.13, no.1, 107-120*, 1983.

[64] A. Satyanarayana and K. Wood. A linear time algorithm for computing k-terminal reliability in series parallel networks. In *SIAM Journal on Computing, vol.14, no.4, 818-832*, 1985.

[65] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. In *Information Processing Letters , vol.48, no.3, 139-144*, 1993.

[66] A. Sklar. Fonctions de répartition à n dimensions et leurs marges. In *Publ. Inst. Statist. Univ. Paris vol.8, 229-231*, 1959.

[67] R. V. Slyke and H. Frank. Network reliability analysis: Part I. In *Networks, vol.1, no.3, 279-290*, 1971.

[68] J. Song and S. Y. Ok. Multi-scale system reliability analysis of lifelien networks under earthquake hazards. In *Earthquake Engineering & Structural Dynamics, vol.39, no.3, 259-279*, 2009.

[69] H. Störmer. *Mathematische Theorie der Zuverlässigkeit*. R. Oldenbourg, 1983.

[70] C. J. Colbourn T. B. Brecht. Improving reliability bounds in computer networks. In *Networks, vol.16, no.4, 369-380*, 1986.

[71] C. J. Colbourn T. B. Brecht. Lower bounds on two-terminal network reliability. In *Journal Discrete Applied Mathematics archive, vol.21, no.3, 185-198*, 1988.

[72] C. Tanguy. Asymptotic Mean Time To Failure and Higher Moments for Large, Recursive Networks. In *CoRR*, 2008.

[73] O. R. Theologou and J. G. Carlier. Factoring and reductions for networks with imperfect vertices. In *IEEE Trans. Reliability, vol.40, no.2, 210-217*, 1991.

[74] D. Torrieri. Calculation of node-pair reliability in large networks with unreliable nodes. In *IEEE Trans. Reliability, vol.43, no.3, 375-377*, 1994.

[75] W. T. Tutte. On the Problem of Decomposing a Graph into n Connected Factors. In *J. London Math. Soc. 36, 221-230*, 1961.

[76] L. G. Valiant. The complexity of enumeration and reliability problems. In *SIAM Journal on Computing, vol.8, no.3, 410-421*, 1979.

[77] F. Viger and M. Latapy. Efficient and Simple Generation of Random Simple Connected Graphs with Prescribed Degree Sequence. In *Lecture Notes in Computer Science, vol.3595, 440-449*, 2005.

[78] D. K. Wagner. Disjoint (s, t)-cuts in a network. In *Networks, vol.20, no.4, 361-371*, 1990.

[79] M. Walter. *Application-oriented evaluation of fault-tolerant systems*. SHAKER Verlag, 2009.

[80] M. Walter, S. Esch, and P. Limbourg. A copula-based approach for dependability analyses of fault-tolerant systems with interdependent basic events. In *ESREL, 1705-1714*, 2008.

[81] M. Walter and W. Schneeweiss. *The Modeling World of Reliability/Safety Engineering*. LiLoLe Verlag Hagen, 2005.

[82] K. Wood. Factoring algorithms for computing k-terminal network reliability. In *IEEE Trans. Reliability, vol.3, no.35, 269-278*, 1986.

[83] K. Wood. Triconnected decomposition for computing k-terminal network reliabilty. In *Networks, vol.19, no.2, 203-220*, 1989.

[84] F. Yeh, S. Lu, and S. Kuo. OBDD-based evaluation of k-terminal network reliability. In *IEEE Transactions on Reliability, vol.51, no.4, 443-451*, 2002.

[85] F. M. Yeh, S. K. Lu, and S. Y. Kuo. Determining Terminal-Pair Reliability based on Edge Expansion Diagrams using OBDD. In *IEEE Trans. Reliability, vol.48, no.3, 234-246*, 1999.

[86] Y. B. Yoo and N. Deo. A comparison of algorithms for terminal pair reliability. In *IEEE Trans. Reliability, vol.37, no.2, 210-215*, 1988.