RISK- AND RELIABILITY ANALYSIS WITH APPLICATIONS

by

ARNE BANG HUSEBY and KRISTINA ROGNLIEN DAHL

Department of Mathematics
Faculty of Mathematics and Natural Sciences
University of Oslo

Abstract

This is a compendium intended for use in the course STK3405 and STK4405 at the Department of Mathematics, University of Oslo, from fall 2022. The compendium provides the foundation of probability theory needed to compute the reliability of a system, i.e., the probability that a system is functioning, when the reliabilities of the components which the system is composed of are known. Examples of systems are energy systems and networks. In addition, various examples of the use of risk analysis for industrial applications are studied. The material is illustrated by different simulation techniques. This material can serve as an introduction to more advanced studies, but is also suitable as support literature to studies in other fields and as further education for realists and engineers.

Contents

List of Figures

Chapter 1

Introduction

The purpose of these notes is to provide the reader with a basic knowledge on the theoretical foundations of risk- and reliability analysis. In addition, the compendium covers a wide array of current applications of these fields as well as simulation techniques and an introduction to software suitable for applying the theory to practical problems.

Today, various aspects of risk- and reliability theory are used in many different sectors. Examples include transport, project planning, time scheduling, energy networks and aerospace. At the core of all these applications is a need to quantify and control risk, determine how reliable a system is and how to monitor and maintain systems is the best way possible. Clever use of the risk- and reliability theory can provide safer and more efficient systems which are supervised in an optimal way. This leads to financial savings, well-functioning systems and prevention of accidents.

These notes are suitable for a bachelor or master level course in modern risk- and reliability theory, but also as further education for practitioners (engineers, risk analysts etc.).

The structure of the notes is as follows: In Chapter 2 we introduce the theoretical foundation of system analysis. We define a system of components and study various kinds of systems. The structure/reliability function of the system is defined based on component reliabilities. The chapter also includes important defining properties of systems such as monotonicity, coherency and duality. In Chapter 3 we introduce the basic tools for calculating reliability, including pivotal decompositions, path and cut sets as well as modular decompositions. Then, in Chapter 4, we consider different methods for exact

computation of the reliability of systems. In Chapter 5 we introduce various measures for computing the structural and the reliability importance of the system components.

In Chapter 7 we show how to analyse systems where the states of the components, and hence also the state of the system, evolve over time. We also introduce discrete event simulation and show how this can be used to estimate availability and importance in systems with repairable components.

In some cases, computing the exact reliability is too time-consuming, so in Chapter 8 we give upper and lower bounds for the system reliabilities. In this chapter, we also define a specific kind of dependence between the components of a system called association, and study how this property can be used to derive bounds for the system reliability without requiring independence of components.

In Chapter 9 we present various applications and examples of practical use of risk- and reliability theory. Some examples include project management and time scheduling, analyzing a network for the transmission of electronic pulses and a conditional Monte Carlo approach to system reliability evaluation.

Throughout the notes, there are various exercises intended to provide the reader with a deeper understanding of the material. Some of the exercises are more theoretical, some are oriented towards analyzing smaller systems by hand and other rely on simulation techniques and use of Riscue.

The notation used in the compendium is summarized in Chapter A.

Parts of these notes are based on the compendium by Natvig [50] (in Norwegian). In particular, this is the case for Chapter 2. However, compared to Natvig [50] there are several new examples, exercises, some new topics are included and some others are excluded. The presentation has also been reorganized and reworked.

Chapter 2

System analysis

In this chapter, we will introduce the basic concepts of system analysis and reliability theory. In contrast to several other books on this topic, for instance Barlow and Proschan [7] and Natvig [50], we will begin by introducing *stochastic systems* (i.e., when the component states are random). The more classical approach is to first consider *deterministic system* analysis (i.e., where the component states are deterministic), and then move on to the stochastic case. However, as the two are very similar, we go straight to the general stochastic case, and highlight important aspects of the deterministic case whenever necessary. This hopefully saves the reader some work and also makes the presentation more unified. At first, we analyse systems at a particular time. We then move on to *dynamic system analysis* where we consider the development of the systems over time.

2.1 Binary monotone systems

When we use the term *system*, we think of some technological unit consisting of a finite set $C = \{1, \ldots, n\}$ of *components* which are operating together. A *binary* system has only two possible states: *functioning* or *failed*. Moreover, each component is either functioning or failed as well[1]. For each component

[1]More generally, *multistate reliability theory*, handles cases where the system and its components have more than two potential states are considered. Much of the theory for binary systems can be extended to multistate systems. Still such systems typically are more computationally challenging. See Natvig [51].

$i \in C$ we introduce the component state variable x_i denoting the *state* of component i and defined as:

$$x_i = \begin{cases} 1 & \text{if component } i \text{ is functioning} \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

The vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ is referred to as the component state vector. Note that we sometimes use upper case letters for the component state variables and the component state vector when these are considered to be stochastic variables. In such cases specific values of the stochastic component state variables will be denoted by lower case letters, $x_1, \ldots, x_n$. Similarly, $\boldsymbol{x}$ denotes a specific value of the stochastic vector $\boldsymbol{X}$.

The *state* of the system is denoted by ϕ defined as:

$$\phi = \begin{cases} 1 & \text{if the system is functioning} \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

The variables x_i, $i = 1, \ldots, n$ and ϕ are said to be *binary*, since they only have two possible values, 0 and 1.

The state of the system is assumed to be uniquely determined by the component state vector, $\boldsymbol{x}$. That is, ϕ is a stochastic variable which can be expressed as a function of $\boldsymbol{x}$:

$$\phi = \phi(\boldsymbol{x}).$$

The function $\phi : \{0,1\}^n \to \{0,1\}$ is called the *structure function* of the system. For a specific value of the component state vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, the resulting system state is given by $\phi = \phi(\boldsymbol{x})$.

Intuitively, repairing a component should not make the system worse, and breaking a component should not make system better. Mathematically this property implies that the structure function should be non-decreasing in each argument. This property is an essential part of the definition of a *binary monotone system* which can be stated as follows:

Definition 2.1.1 *A binary monotone system is an ordered pair (C, ϕ), where $C = \{1, \ldots, n\}$ is the component set and ϕ is the structure function. The function ϕ is assumed to be non-decreasing in each argument. The number of components, n, is referred to as the* order *of the system.*

Note that according to this definition the class of binary monotone systems includes systems where the structure function is constant, i.e., systems where either $\phi(\boldsymbol{x}) = 1$ for all $\boldsymbol{x} \in \{0,1\}^n$ or $\phi(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in \{0,1\}^n$. We will refer to such systems as *trivial* systems[2].

We let $\mathbf{0}$ denote the vector where all entries are 0, and similarly, let $\mathbf{1}$ be the vector where all entries are 1. The following result provides a useful property of non-trivial systems.

Theorem 2.1.2 *Let (C, ϕ) be a non-trivial binary monotone system. Then $\phi(\mathbf{0}) = 0$ and $\phi(\mathbf{1}) = 1$.*

Proof: Since (C, ϕ) is assumed to be non-trivial, there exists at least one vector $\boldsymbol{x}_0$ such that $\phi(\boldsymbol{x}_0) = 0$ and another vector $\boldsymbol{x}_1$ such that $\phi(\boldsymbol{x}_1) = 1$. Since (C, ϕ) is assumed to be monotone, it follows that $0 \leq \phi(\mathbf{0}) \leq \phi(\boldsymbol{x}_0) = 0$ and $1 = \phi(\boldsymbol{x}_1) \leq \phi(\mathbf{1}) \leq 1$. Hence, it follows that $\phi(\mathbf{0}) = 0$ and $\phi(\mathbf{1}) = 1$ $\square$

In order to describe the logical relation between the components and the system, we often use a *reliability block diagram*. In such diagrams components are represented as circles and connected by lines. The system is functioning if and only it is possible to find a way from the left-hand side to the right-hand side of the diagram passing only functioning components. Note that in order to represent arbitrarily complex binary monotone systems, it is sometimes necessary to allow components to occur multiple places in the block diagram. In such cases the reliability block diagram should not be interpreted as a picture of a physical system.

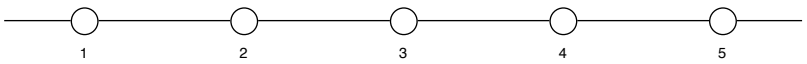

Figure 2.1: A reliability block diagram of a series system of order 5 with component set $C = \{1, \ldots, 5\}$

[2]Some textbooks exclude trivial systems from the class of monotone systems. We have chosen not to do so in order to have a class which is closed with respect to conditioning.

Example 2.1.3 *Figure 2.1 shows a reliability block diagram of a* series system *of order* 5. *From the diagram we see that the series system is functioning if and only if all of its components are functioning, i.e., if and only if* $x_1 = \cdots = x_5 = 1$. *Hence, the state of the system can be expressed as a function of the component state variables as follows:*

$$\phi = x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 = \prod_{i=1}^{5} x_i.$$

Alternatively, the structure function of the series system can be written as:

$$\phi(\boldsymbol{x}) = \min\{x_1, \ldots, x_5\}$$

since the minimum of $x_1, \ldots, x_5$ *is* 1 *if and only if* $x_1 = \cdots = x_5 = 1$.

It may be useful to think of the product operator $(\cdot)$ of binary variables as representing the logical *and*-operation. Thus, for the series system to function, components 1 *and* 2 *and* $\cdots$ *and* 5 all need to function.

Both the *product*-expression and the *minimum*-expression can be extended to series systems of arbitrary orders. Thus, the structure function of a series system of order n can be written as:

$$\phi(\boldsymbol{x}) = \prod_{i=1}^{n} x_i = \min_{1 \le i \le n} x_i. \tag{2.3}$$

In order to derive convenient representations of structure functions of other types of binary monotone system, we also need another operator, called the *coproduct operator*. This operator is denoted by the symbol II and defined as follows:

Notation 2.1.4 *For any* $a_1, a_2, \ldots, a_n$ *where* $a_i \in [0, 1]$, $i = 1, 2, \ldots, n$, *we define:*

$$a_1 \amalg a_2 = 1 - (1 - a_1)(1 - a_2),$$
$$\coprod_{i=1}^{n} a_i = 1 - \prod_{i=1}^{n} (1 - a_i).$$

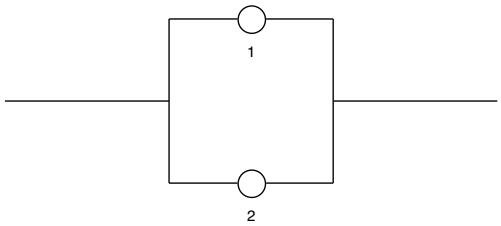

Figure 2.2: A parallel system of order 2

It is easy to verify that $1 \amalg 1 = 1 \amalg 0 = 0 \amalg 1 = 1$, while $0 \amalg 0 = 0$. More generally, if $a_1, \ldots, a_n$ are *binary numbers* (i.e., either 0 or 1), their coproduct is 1 if $a_i = 1$ for at least one i, and 0 if $a_i = 0$ for all i.

Example 2.1.5 *Figure 2.2 shows a reliability block diagram of a* parallel system *of order* 2. *We see that the parallel system functions if and only if at least one of its components is functioning. The structure function of this system can be written as:*

$$\phi(\boldsymbol{x}) = x_1 \amalg x_2 = 1 - (1 - x_1) \cdot (1 - x_2). \tag{2.4}$$

To see this, note that the only way the system is not functioning is that x_1 *and* x_2 *are both zero. In this case, the right hand side of* (2.4) *is clearly* 0. *For any other combination of values for* x_1 *and* x_2, *the right hand side of* (2.4) *is* 1.

Alternatively, the structure function of the parallel system can be written as:

$$\phi(\boldsymbol{x}) = \max\{x_1, x_2\}$$

since the maximum of x_1 *and* x_2 *is* 0 *if and only if* $x_1 = x_2 = 0$.

From Example 2.1.5 we see that the coproduct-operator, $\amalg$, may be though of as representing the logical *or*-operation. Thus, for the above parallel system to function components 1 *or* 2 must function.

Both the *coproduct*-expression and the *maximum*-expression can be extended to parallel systems of arbitrary orders. Thus, the structure function

of a parallel system of order n can be written as:

$$\phi(\boldsymbol{x}) = \coprod_{i=1}^{n} x_i = \max_{1 \le i \le n} x_i. \tag{2.5}$$

2.2 Coherent systems

Every component of a binary monotone system should have some impact on the system state. More precisely, if i is a component in a system, there should ideally exist at least some state of the rest of the system where the state of component i is crucial for the system state. Before we formalise this concept further, we introduce some notation:

$$\begin{aligned}(1_i, \boldsymbol{x}) &= (x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n),\\ (0_i, \boldsymbol{x}) &= (x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n),\\ (\cdot_i, \boldsymbol{x}) &= (x_1, \ldots, x_{i-1}, \cdot, x_{i+1}, \ldots, x_n).\end{aligned}$$

Definition 2.2.1 *Let (C, ϕ) be a binary monotone system, and let $i \in C$. The component i is said to be* relevant *for the system (C, ϕ) if:*

$$0 = \phi(0_i, \boldsymbol{x}) < \phi(1_i, \boldsymbol{x}) = 1 \text{ for some } (\cdot_i, \boldsymbol{x}).$$

If this is not the case, component i is said to be irrelevant *for the system.*

Relevance of components plays a very important role in reliability theory. Based on this concept we also introduce the following crucial concept.

Definition 2.2.2 *A binary monotone system (C, ϕ) is* coherent *if all its components are relevant.*

Note that a coherent system is obviously non-trivial as well, since in a trivial system *all* components are irrelevant.

One might think that all binary monotone systems we encounter should be coherent. However, coherency should be viewed a dynamic property. As parts of a system breaks, some of the remaining components may become irrelevant. Similarly, if we, e.g., as part of a reliability calculation, condition on the state of a component, the resulting system may not be coherent.

Finally, depending on how we model a system, we may encounter components which improve the performance of the system even though they are not relevant according to our definition.

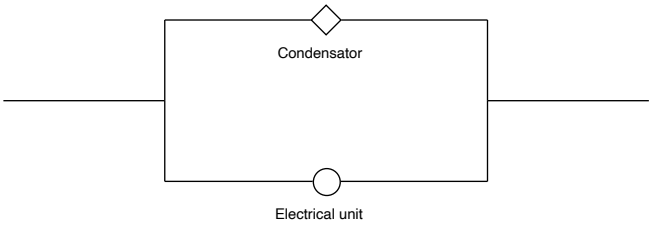

Figure 2.3: A system with an irrelevant component.

Example 2.2.3 *Figure 2.3 shows a part of a large machine consisting of an electrical unit and a condensator. The condensator's job is to prevent the electrical unit from being broken due to high voltage. Viewed as a component, the condensator is irrelevant. However, this does not mean that it is not important! The condensator can prolong the lifetime of the electrical unit, and hence also the lifetime of the whole machine.*

Note that for this example, the problem can be avoided by considering the electrical unit and the condensator as one component instead of two.

We now prove the intuitively reasonable result that within the class of non-trivial monotone systems, the parallel structure is "the best" while the series system is "the worst".

Theorem 2.2.4 *Let (C, ϕ) be a non-trivial binary monotone system of order n. Then for all $\boldsymbol{x} \in \{0,1\}^n$ we have:*

$$\prod_{i=1}^{n} x_i \leq \phi(\boldsymbol{x}) \leq \coprod_{i=1}^{n} x_i. \tag{2.6}$$

Proof: We focus on the left-hand inequality. If $\prod_{i=1}^{n} x_i = 0$, this inequality is trivial since $\phi(\boldsymbol{x}) \in \{0,1\}$ for all $\boldsymbol{x} \in \{0,1\}^n$. If on the other hand $\prod_{i=1}^{n} x_i = 1$, we must have $\boldsymbol{x} = \mathbf{1}$. Since (C, ϕ) is assumed to be non-trivial, it follows by Theorem 2.1.2 that $\phi(\mathbf{1}) = 1$. Thus, the inequality is valid in this case as well. This completes the proof of the left-hand inequality. The right-hand inequality is proved in a similar fashion and is left as an exercise. □

By introducing redundancy into a system one could improve the reliability. However, there are many different ways of doing this. In particular, we may introduce redundancy at the component level or at the system level. The next result shows that redundancy at the component level is better than redundancy at the system level. Moreover, the opposite is true for series connections. In order to prove this result, we introduce the product and coproduct operators for vectors:

$$\begin{aligned}\boldsymbol{x} \cdot \boldsymbol{y} &= (x_1 \cdot y_1, x_2 \cdot y_2, \ldots, x_n \cdot y_n),\\ \boldsymbol{x} \amalg \boldsymbol{y} &= (x_1 \amalg y_1, x_2 \amalg y_2, \ldots, x_n \amalg y_n).\end{aligned}$$

Theorem 2.2.5 *Let (C, ϕ) be a binary monotone system of order n. Then for all $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^n$ we have:*

(i) $\phi(\boldsymbol{x} \amalg \boldsymbol{y}) \geq \phi(\boldsymbol{x}) \amalg \phi(\boldsymbol{y})$,

(ii) $\phi(\boldsymbol{x} \cdot \boldsymbol{y}) \leq \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{y})$.

Moreover, assume that (C, ϕ) is coherent. Then equality holds in (i) for all $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^n$ if and only if (C, ϕ) is a parallel system. Similarly, equality holds in (ii) for all $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^n$ if and only if (C, ϕ) is a series system.

Proof: Since ϕ is non-decreasing and $x_i \amalg y_i \geq x_i$ for $i = 1, \ldots, n$, it follows that:

$$\phi(\boldsymbol{x} \amalg \boldsymbol{y}) \geq \phi(\boldsymbol{x}).$$

Similarly, we see that

$$\phi(\boldsymbol{x} \amalg \boldsymbol{y}) \geq \phi(\boldsymbol{y}).$$

Hence,

$$\phi(\boldsymbol{x} \amalg \boldsymbol{y}) \geq \max\{\phi(\boldsymbol{x}), \phi(\boldsymbol{y})\} = \phi(\boldsymbol{x}) \amalg \phi(\boldsymbol{y}).$$

This proves (i). The proof of (ii) is similar and is left as an exercise.

We then assume that (C, ϕ) is coherent. We shall prove the first equivalence, i.e., that equality holds in (i) for all $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^n$ if and only if (C, ϕ) is a parallel system.

If (C, ϕ) is a parallel system, it follows that:

$$\begin{aligned}\phi(\boldsymbol{x} \amalg \boldsymbol{y}) &= \coprod_{i=1}^{n}(x_i \amalg y_i) = \max_{1\le i\le n}\big\{\max\{x_i, y_i\}\big\} \\ &= \max\big\{\max_{1\le i\le n} x_i, \max_{1\le i\le n} y_i\big\} = \phi(\boldsymbol{x}) \amalg \phi(\boldsymbol{y}),\end{aligned}$$

which proves the "if"-part of the equivalence.

Assume conversely that $\phi(\boldsymbol{x} \amalg \boldsymbol{y}) = \phi(\boldsymbol{x}) \amalg \phi(\boldsymbol{y})$ for all $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^n$. Since (C, ϕ) is coherent it follows that for any $i \in C$ there exists a vector $(\cdot_i, \boldsymbol{x})$ such that:

$$\phi(1_i, \boldsymbol{x}) = 1 \text{ and } \phi(0_i, \boldsymbol{x}) = 0.$$

For this particular vector $(\cdot_i, \boldsymbol{x})$ we have:

$$\begin{aligned}1 = \phi(1_i, \boldsymbol{x}) &= \phi\big((1_i, \mathbf{0}) \amalg (0_i, \boldsymbol{x})\big) \\ &= \phi(1_i, \mathbf{0}) \amalg \phi(0_i, \boldsymbol{x}) = \phi(1_i, \mathbf{0}) \amalg 0 \\ &= \phi(1_i, \mathbf{0}).\end{aligned}$$

Since $\phi(\mathbf{0}) = 0$, it follows that:

$$\phi((x_i)_i, \mathbf{0}) = x_i \qquad x_i = 0, 1; \quad i = 1, \ldots, n.$$

Hence, for any $\boldsymbol{x}$, we have:

$$\begin{aligned}\phi(\boldsymbol{x}) &= \phi\big(((x_1)_1, \mathbf{0}) \amalg ((x_2)_2, \mathbf{0}) \amalg \ldots \amalg ((x_n)_n, \mathbf{0})\big) \\ &= \phi((x_1)_1, \mathbf{0}) \amalg \phi((x_2)_2, \mathbf{0}) \amalg \ldots \amalg \phi((x_n)_n, \mathbf{0}) \\ &= x_1 \amalg x_2 \amalg \ldots \amalg x_n = \coprod_{i=1}^{n} x_i.\end{aligned}$$

Thus, we have shown that (C, ϕ) is a parallel system which proves the "only if"-part of the equivalence. The other equivalence is proved similarly and is left as an exercise. □

Example 2.2.6 *To illustrate Theorem 2.2.5, let ϕ be a series system of order 2. Then the statement of the theorem is that componentwise connection of the nodes provides a better result than systemwise connection, i.e.,*

$$(x_1 \amalg y_1) \cdot (x_2 \amalg y_2) \ge (x_1 \cdot x_2) \amalg (y_1 \cdot y_2).$$

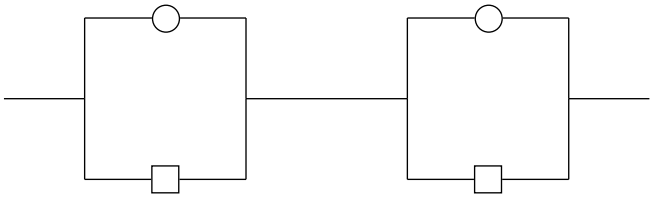

Figure 2.4: A componentwise parallel structure: "better" than systemwise parallel.

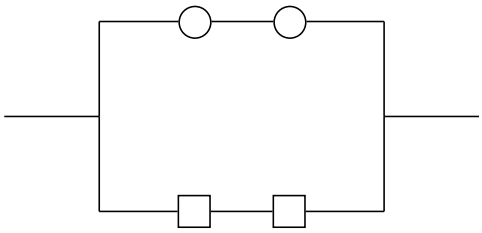

Figure 2.5: A systemwise parallel structure: "worse" than componentwise parallel.

See Figures 2.4 and 2.5 for an illustration of this result. Here x_1 *and* x_2 *represent the states of the circular nodes, while* y_1 *and* y_2 *represent the states of the square nodes.*

Assume e.g., that $x_1 = y_2 = 1$ *while* $x_2 = y_1 = 0$*. Then the componentwise parallel structure is functioning, while the systemwise parallel structure is failed. By Theorem 2.2.5 the converse will never hold true. If the systemwise parallel structure is functioning, the componentwise parallel structure must be functioning as well. Similarly, if the componentwise parallel structure is failed, then the systemwise parallel structure must be failed as well.*

2.3 Dual systems

In this section we will introduce a concept of *duality*:

Definition 2.3.1 *Let ϕ be a structure function of a binary monotone system of order n. We then define the* dual structure function*, ϕ^D for all $\boldsymbol{y} \in \{0,1\}^n$ as*[3]*:*

$$\phi^D(\boldsymbol{y}) = 1 - \phi(\mathbf{1} - \boldsymbol{y}).$$

Furthermore, if $\boldsymbol{X}$ is the component state vector of a binary monotone system, we define the dual component state vector $\boldsymbol{X}^D$ as:

$$\boldsymbol{X}^D = (X_1^D, \ldots, X_n^D) = (1 - X_1, \ldots, 1 - X_n) = \mathbf{1} - \boldsymbol{X}$$

Note that the relation between ϕ and ϕ^D should be interpreted as a relation between two *functions*, while the relation between $\boldsymbol{X}$ and $\boldsymbol{X}^D$ is a relation between two stochastic vectors. Corresponding to the dual component state vector $\boldsymbol{X}^D$ we introduce the dual component set $C^D = \{1^D, \ldots, n^D\}$, where it is understood that the dual component i^D is functioning if the component i is failed, while i^D is failed if the component i is functioning.

By combining duality on the component level and on the system level, we see that we have the following relation between the two stochastic variables $\phi(\boldsymbol{X})$ and $\phi^D(\boldsymbol{X}^D)$:

$$\phi^D(\boldsymbol{X}^D) = 1 - \phi(\mathbf{1} - \boldsymbol{X}^D) = 1 - \phi(\boldsymbol{X}).$$

Hence, the dual system is functioning if and only if the original system is failed and vice versa.

Example 2.3.2 *Let ϕ be the structure function of a system of order 3 where:*

$$\phi(\boldsymbol{y}) = y_1 \amalg (y_2 \cdot y_3),$$

The dual structure function is then given by:

$$\begin{aligned}
\phi^D(\boldsymbol{y}) &= 1 - \phi(\mathbf{1} - \boldsymbol{y}) \\
&= 1 - (1 - y_1) \amalg ((1 - y_2) \cdot (1 - y_3)) \\
&= 1 - [1 - (1 - (1 - y_1))(1 - (1 - y_2) \cdot (1 - y_3))] \\
&= 1 - [1 - y_1 \cdot (1 - (1 - y_2) \cdot (1 - y_3))] \\
&= y_1 \cdot (y_2 \amalg y_3)
\end{aligned}$$

[3]Note that we use $\boldsymbol{y}$ as argument instead of $\boldsymbol{X}$ here to emphasize that we are describing a relation between the two binary functions ϕ and ϕ^D. This should not be confused with the relation between the two random variables $\phi(\boldsymbol{X})$ and $\phi^D(\boldsymbol{X}^D)$.

Example 2.3.3 *Let (C, ϕ) be a series system of order n. That is, ϕ is given by:*

$$\phi(\boldsymbol{y}) = \prod_{i=1}^{n} y_i.$$

The dual structure function is then given by:

$$\begin{aligned}\phi^D(\boldsymbol{y}) &= 1 - \phi(\mathbf{1} - \boldsymbol{y}) \\ &= 1 - \prod_{i=1}^{n}(1 - y_i) \\ &= \coprod_{i=1}^{n} y_i.\end{aligned}$$

Thus, (C^D, ϕ^D) is a parallel system of order n.

Example 2.3.4 *Let (C, ϕ) be a parallel system of order n. That is, ϕ is given by:*

$$\phi(\boldsymbol{y}) = \coprod_{i=1}^{n} y_i.$$

The dual structure function is then given by:

$$\begin{aligned}\phi^D(\boldsymbol{y}) &= 1 - \phi(\mathbf{1} - \boldsymbol{y}) \\ &= 1 - \coprod_{i=1}^{n}(1 - y_i) \\ &= 1 - (1 - \prod_{i=1}^{n}(1 - (1 - y_i)) \\ &= \prod_{i=1}^{n} y_i.\end{aligned}$$

Thus, (C^D, ϕ^D) is a series system of order n.

We close this section by a result showing that if we take the dual of the dual system, we get the original system back.

Theorem 2.3.5 *Let ϕ be the structure function of a binary monotone system, and let ϕ^D be the corresponding dual structure function. Then we have:*

$$(\phi^D)^D = \phi.$$

That is, the dual of the dual system is equal to the original system.

Proof: For all $\boldsymbol{y} \in \{0,1\}^n$ we have:

$$\begin{aligned}(\phi^D)^D(\boldsymbol{y}) &= 1 - \phi^D(\mathbf{1} - \boldsymbol{y}) \\ &= 1 - [1 - \phi(\mathbf{1} - (\mathbf{1} - \boldsymbol{y}))] \\ &= \phi(\boldsymbol{y}).\end{aligned}$$

□

2.4 Reliability of binary monotone systems

If (C, ϕ) is a binary monotone system, the *reliability* of a component $i \in C$, denoted by p_i, is defined as the probability that component i is functioning. That is, $p_i = P(X_i = 1)$, for all $i \in C$. Obviously, a component with high reliability is likely to be functioning, while a component with low reliability is more likely to fail. Note that since X_i is binary, we have:

$$E[X_i] = 0 \cdot P(X_i = 0) + 1 \cdot P(X_i = 1) = P(X_i = 1) = p_i, \text{ for all } i \in C. \quad (2.7)$$

Thus, the reliability of component i is equal to the expected value of its component state variable, X_i.

Similarly, the *reliability* of the system, denoted by h, is defined as the probability that the system is functioning. That is, $h = P(\phi = 1)$. Again, since ϕ is binary, we have:

$$E[\phi(\boldsymbol{X})] = 0 \cdot P(\phi(\boldsymbol{X}) = 0) + 1 \cdot P(\phi(\boldsymbol{X}) = 1) = P(\phi(\boldsymbol{X}) = 1) = h. \quad (2.8)$$

Thus, the reliability of the system is equal to the expected value of the structure function, $\phi(\boldsymbol{X})$. From this it immediately follows that the reliability of a system, at least in principle, can be calculated as:

$$h = E[\phi(\boldsymbol{X})] = \sum_{\boldsymbol{x} \in \{0,1\}^n} \phi(\boldsymbol{x}) P(\boldsymbol{X} = \boldsymbol{x}) \quad (2.9)$$

In the case where the component state variables are dependent, the system reliability h will depend on the full joint distribution of the component state vector. Hence, if we only know the component reliabilities, the best we can do is to establish upper and lower bounds for h. See Section 8.2 for more on this. In the remaining part of this section we instead consider the case where the component state variables can be assumed to be independent. In order to do so we introduce the vector of component reliabilities, $\boldsymbol{p} = (p_1, p_2, \ldots, p_n)$, and note that:

$$P(X_i = x_i) = \begin{cases} p_i & \text{if } x_i = 1, \\ 1 - p_i & \text{if } x_i = 0. \end{cases}$$

Since x_i is either 0 or 1, this probability can be written in the following more compact form:

$$P(X_i = x_i) = p_i^{x_i}(1 - p_i)^{1-x_i}.$$

If the component state variables $X_1, X_2, \ldots, X_n$ are independent, we can use this in order to write the joint distribution of $\boldsymbol{X}$ in terms of $\boldsymbol{p}$ as follows:

$$P(\boldsymbol{X} = \boldsymbol{x}) = \prod_{i=1}^{n} P(X_i = x_i) = \prod_{i=1}^{n} p_i^{x_i}(1 - p_i)^{1-x_i}. \tag{2.10}$$

Thus, when the component state variables are independent, we may insert (2.10) into (2.9) and get the following expression for the system reliability:

$$h = E[\phi(\boldsymbol{X})] = \sum_{\boldsymbol{x} \in \{0,1\}^n} \phi(\boldsymbol{x}) \prod_{i=1}^{n} p_i^{x_i}(1 - p_i)^{1-x_i} \tag{2.11}$$

Note that in this case the reliability of the system is a function of the component reliabilities, so we may write

$$h = h(\boldsymbol{p}). \tag{2.12}$$

The function $h(\boldsymbol{p})$ is called *the reliability function* of the system.

Example 2.4.1 *Consider a series system of order n. Assuming that the component state variables are independent, the reliability of this system is given by:*

$$h(\boldsymbol{p}) = E[\phi(\boldsymbol{X})] = E[\prod_{i=1}^{n} X_i] = \prod_{i=1}^{n} E[X_i] = \prod_{i=1}^{n} p_i,$$

where the third equality follows by using that $X_1, X_2, \ldots, X_n$ are assumed to be independent.

Example 2.4.2 *Consider a parallel system of order n. Assuming that the component state variables are independent, the reliability of this system is given by:*

$$h(\boldsymbol{p}) = E[\phi(\boldsymbol{X})] = E[\coprod_{i=1}^{n} X_i] = E[1 - \prod_{i=1}^{n}(1 - X_i)]$$
$$= 1 - \prod_{i=1}^{n}(1 - E[X_i]) = \coprod_{i=1}^{n} E[X_i] = \coprod_{i=1}^{n} p_i,$$

where the fourth equality follows by using that $X_1, X_2, \ldots, X_n$ are assumed to be independent.

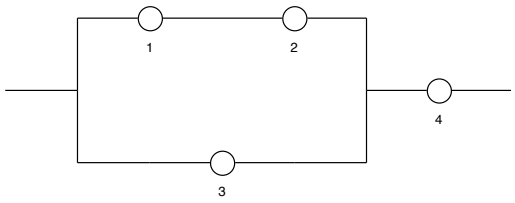

Figure 2.6: A mixed parallel and series system

Example 2.4.3 *The structure function for the system in Figure 2.6 is*

$$\phi(\boldsymbol{X}) = \Big((X_1 \cdot X_2) \amalg X_3\Big) \cdot X_4, \tag{2.13}$$

because for the system to function ((components 1 and 2 must function) or (component 3 must function)) and component 4 must function.

Alternatively, we see that components 1 and 2 are in series implying that the structure function depends on components 1 and 2 through $X_1 \cdot X_2$. Furthermore, the part of the system containing components 1, 2 is in parallel with component 3 implying that the structure function depends on components 1, 2 and 3 through $(X_1 \cdot X_2) \amalg X_3$. Finally, the part of the system containing components 1, 2 and 3 is in series with component 4, implying that the structure function is as in equation (2.13).

Assuming that the component state variables are independent it is easy to verify, using a similar argument as we used for the structure function, that the reliability of the system is given by:

$$h(\boldsymbol{p}) = \Big((p_1 \cdot p_2) \amalg p_3\Big) \cdot p_4.$$

Theorem 2.2.5 can be extended to reliability functions as well. That is, we have the following result:

Theorem 2.4.4 *Let $h(\boldsymbol{p})$ be the reliability function of a monotone system (C, ϕ) of order n. Then for all $\boldsymbol{p}, \boldsymbol{p}' \in [0,1]^n$ we have:*

(i) $h(\boldsymbol{p} \amalg \boldsymbol{p}') \geq h(\boldsymbol{p}) \amalg h(\boldsymbol{p}')$,

(ii) $h(\boldsymbol{p} \cdot \boldsymbol{p}') \leq h(\boldsymbol{p}) \cdot h(\boldsymbol{p}')$

If in addition (C, ϕ) is coherent, equality holds in (i) for all $\boldsymbol{p}, \boldsymbol{p}' \in [0,1]^n$ if and only if (C, ϕ) is a parallel system. Similarly, equality holds in (ii) for all $\boldsymbol{p}, \boldsymbol{p}' \in [0,1]^n$ if and only if (C, ϕ) is a series system.

Proof: Note that the theorem implicitly assumes independence, since we can write $h(\boldsymbol{p})$. Then, for independent $\boldsymbol{X}, \boldsymbol{Y}$ (corresponding to $\boldsymbol{p}, \boldsymbol{p}'$), we have:

$$\begin{aligned} h(\boldsymbol{p} \amalg \boldsymbol{p}') &- h(\boldsymbol{p}) \amalg h(\boldsymbol{p}') \\ &= E[\phi(\boldsymbol{X} \amalg \boldsymbol{Y})] - E[\phi(\boldsymbol{X})] \amalg E[\phi(\boldsymbol{Y})] \\ &= E[\phi(\boldsymbol{X} \amalg \boldsymbol{Y}) - \phi(\boldsymbol{X}) \amalg \phi(\boldsymbol{Y})], \end{aligned}$$

where the last expectation must be non-negative since by Theorem 2.2.5 we know that:

$$\phi(\boldsymbol{x} \amalg \boldsymbol{y}) - \phi(\boldsymbol{x}) \amalg \phi(\boldsymbol{y}) \geq 0,$$

for all $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^n$. This completes the proof of (i). The proof of (ii) is similar and is left as an exercise.

We now show that equality in (i) holds for all $\boldsymbol{p}, \boldsymbol{p}' \in [0,1]^n$ if and only if (C, ϕ) is a parallel system. In order to do so, we may choose $\boldsymbol{p}$ and $\boldsymbol{p}'$ arbitrary such that $0 < p_i < 1, 0 < p_i' < 1$ for $i = 1, \ldots, n$. This implies that:

$$P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y}) > 0, \text{ for all } \boldsymbol{x} \in \{0,1\}^n \text{ and } \boldsymbol{y} \in \{0,1\}^n.$$

Hence, $E[\phi(\boldsymbol{X} \amalg \boldsymbol{Y}) - \phi(\boldsymbol{X}) \amalg \phi(\boldsymbol{Y})] = 0$ if and only if $\phi(\boldsymbol{x} \amalg \boldsymbol{y}) - \phi(\boldsymbol{x}) \amalg \phi(\boldsymbol{y}) = 0$ for all $\boldsymbol{x} \in \{0,1\}^n$ and $\boldsymbol{y} \in \{0,1\}^n$. By Theorem 2.2.5 this holds if and only if (C, ϕ) is a parallel system. The other equivalence is proved similarly, and left as an exercise. $\square$

Example 2.4.5 *Consider the system* (C, ϕ) *of order* 3 *where:*

$$\phi(\boldsymbol{x}) = x_1 \cdot (x_2 \amalg x_3),$$

and where the component state variables are independent and $P(X_i = 1) = p$ *for all* $i \in C$*. Then it is easy to verify that* $h(p) = p \cdot (p \amalg p) = 2p^2 - p^3$*. From Theorem 2.4.4, it follows that for all* $0 \leq p \leq 1$*, we have:*

$$\begin{aligned} h(p \amalg p) &= 2(p \amalg p)^2 - (p \amalg p)^3 \\ &\geq h(p) \amalg h(p) = (2p^2 - p^3) \amalg (2p^2 - p^3) \end{aligned}$$

In Figure 2.7 we have plotted the reliabilities of the two systems as functions of p*. We see that indeed* $h(p \amalg p)$ *is greater than or equal to* $h(p) \amalg h(p)$ *for all* $0 \leq p \leq 1$*. In fact the inequality is strict when* $0 < p < 1$*.*

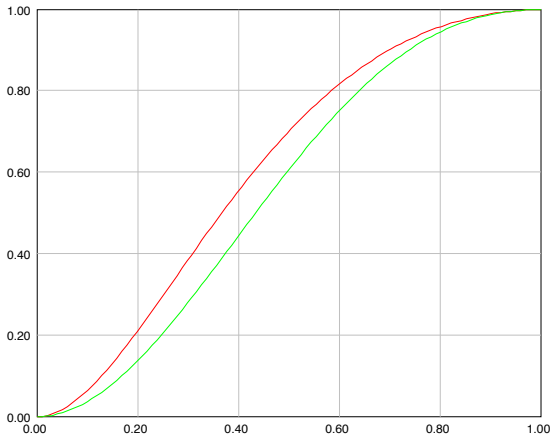

Figure 2.7: $h(p \amalg p)$ (red curve) versus $h(p) \amalg h(p)$ (green curve).

Example 2.4.6 *Consider the system* (C, ϕ) *of order* 3 *where:*

$$\phi(\boldsymbol{x}) = x_1 \amalg (x_2 \cdot x_3),$$

and where the component state variables are independent and $P(X_i = 1) = p$ *for all* $i \in C$*. In this case* $h(p) = p \amalg p^2 = p + p^2 - p^3$*. From Theorem 2.4.4, it follows that for all* $0 \leq p \leq 1$*, we have:*

$$\begin{aligned} h(p \cdot p) &= p^2 + p^4 - p^6 \\ &\leq h(p) \cdot h(p) = (p + p^2 - p^3)^2 \end{aligned}$$

In Figure 2.8 we have plotted the reliabilities of the two systems as functions of p. We see that indeed $h(p \cdot p)$ is less than or equal to $h(p) \cdot h(p)$ for all $0 \leq p \leq 1$, and that the inequality is strict when $0 < p < 1$.

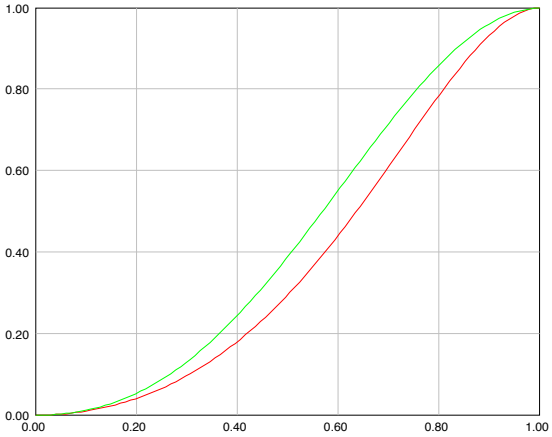

Figure 2.8: $h(p \cdot p)$ (red curve) versus $h(p) \cdot h(p)$ (green curve).

2.5 k-out-of-n systems

A k-out-of-n system is a binary monotone system (C, ϕ) where $C = \{1, \ldots, n\}$ which is functioning if and only if at least k out of the n components are functioning. Note that an n-out-of-n system is a series structure, while a 1-out-of-n system is a parallel structure.

We observe that in order to evaluate the structure function of a k-out-of-n system, we need to count the number of functioning components and check if this number is greater than or equal to k. The number of functioning components is simply the sum of the component state variables. Hence, the structure function can be written as:

$$\phi(\boldsymbol{X}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} X_i \geq k \\ 0 & \text{otherwise.} \end{cases} \tag{2.14}$$

In order to evaluate the reliability of a k-out-of-n system it is convenient to introduce the following random variable:

$$S = \sum_{i=1}^{n} X_i.$$

Thus, S is the number of functioning components. This implies that:

$$h = P(\phi(\boldsymbol{X}) = 1) = P(S \geq k).$$

If the component state variables are independent, it is easy to derive the distribution of S. In particular, if $p_1 = \cdots = p_n = p$, S is a binomially distributed random variable. Thus, we have:

$$P(S = i) = \binom{n}{i} p^i (1-p)^{n-i}.$$

Hence, the reliability of the system is given by:

$$h(\boldsymbol{p}) = h(p) = P(S \geq k) = \sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{n-i} \tag{2.15}$$

In the general case with unequal component reliabilities, an explicit analytical expression for the distribution of S is a bit more complicated. We will return to this problem in Section 4.4.

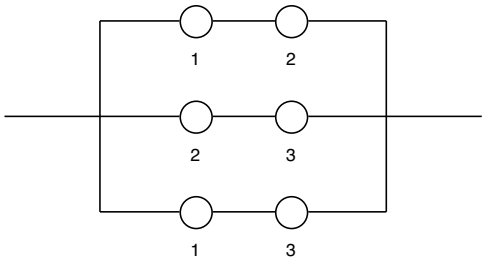

Figure 2.9: A reliability block diagram of a 2-out-of-3 system.

A k-out-of-n system can represented by a reliability block diagram in several ways. The first type of diagram is shown for a 2-out-of-3 system in

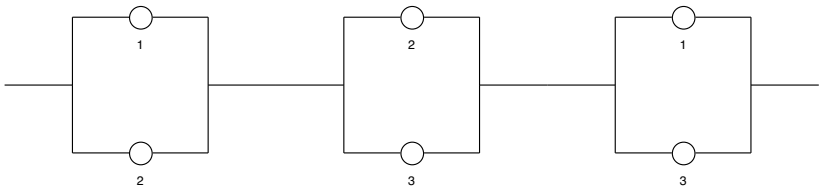

Figure 2.10: An alternative reliability block diagram of a 2-out-of-3 system.

Figure 2.9. Here, we consider a parallel connection of several series structures. Each of the series structures corresponds to a minimal way in which a signal can pass through the system, i.e. any way two out of three components are functioning. The second type of diagram is shown for the same 2-out-of-3 system in Figure 2.10. In this case, we draw a series connection of several parallel structures. Each of the parallel structures corresponds to a minimal way to make the system fail. Again, this happens if any two out of the three components are not working.

2.6 Exercises

Exercise 1. What is the reliability function of a series system of order n where the component states are assumed to be independent?

Exercise 2. Consider the parallel structure of order 2 in Figure 2.2. Assume that the component states are independent.

a) If you know that $p_1 = P(X_1 = 1) = 0.5$ and $p_2 = P(X_2 = 1) = 0.7$, what is the reliability of the parallel system?

b) What is the system reliability if $p_1 = 0.9$ and $p_2 = 0.1$?

c) Can you give an interpretation of these results?

Exercise 3. Consider a binary monotone system (C, ϕ), where the component set is $C = \{1, \ldots, 4\}$ and where ϕ is given by:

$$\phi(\boldsymbol{X}) = X_1 \cdot X_2 \cdot (X_3 \amalg X_4).$$

a) Draw a reliability block diagram of this system.

b) Assume that the components in the system are independent. What is the corresponding reliability function?

Exercise 4. Consider the system shown in Figure 2.11.

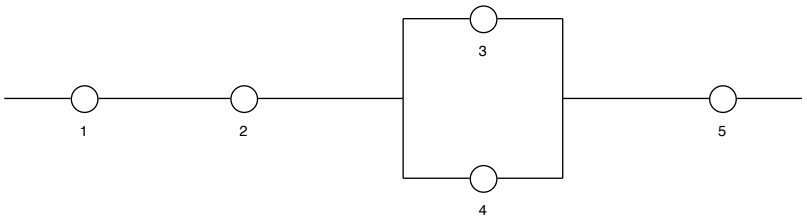

Figure 2.11: A mixed parallel and series system.

a) What is the structure function of this system?

b) Assume that the components in the system are independent. What is the corresponding reliability function?

Exercise 5. There are 8 different coherent systems of order less than or equal to 3 (not counting permutations in the numbering of components). What are they?

(*Hint:* Here are a couple to get you started $\phi(\boldsymbol{X}) = X_1 \cdot X_2 \cdot X_3$, $\phi(\boldsymbol{X}) = X_1 \amalg X_2$.)

Exercise 6. Consider a monotone system (C, ϕ) of order n, and let $A \subset C$ the set of irrelevant components. Furthermore, let $(C \setminus A, \phi')$, be a binary monotone system of order $m = n - |A|$, where ϕ' is a binary non-decreasing function defined for all m-dimensional binary vectors $\boldsymbol{x} \in \{0,1\}^m$ such that:

$$\phi'(\boldsymbol{x}) = \phi(\mathbf{1}^A, \boldsymbol{x}^{\bar{A}})$$

Show that $(C \setminus A, \phi')$ is coherent.

Exercise 7. Prove the right-hand inequality in Theorem 2.2.4.

Exercise 8. Prove item (ii) in Theorem 2.2.5 as well as the equivalence which is not proved in the text.

Exercise 9. Prove that the dual structure of a k-out-of-n structure is an $(n-k+1)$-out-of-n structure.

Chapter 3

Basic reliability calculation methods

In this chapter we will present various methods for simplifying reliability calculations by structuring the problems in certain ways. The first method involves conditioning on the state of a given component. This enables us to decompose the system into simpler building blocks. Furthermore, we will see that the structure function of monotone systems can be represented via its so-called paths and cuts; roughly, this is the components which ensure that the system functions or fails. Another useful way to represent monotone systems is via modules. The idea here is to divide a complicated system into parts which may then be analysed separately. Finally, we will comment briefly on how to include time into this framework by looking at dynamic system analysis.

3.1 Pivotal decompositions

Sometimes reliability calculations can be simplified by dividing the problem into two simpler problems. By considering these simpler problems and combining the results afterwards, even very complex systems can be analysed successfully. The following result shows how a structure and reliability function of order n can be expressed via two corresponding functions of order $n-1$. This is called *pivotal decomposition*, and allows us to reduce the order of the system at hand. This is particularly useful when computing the exact system reliability, see Chapter 4:

Theorem 3.1.1 *Let (C, ϕ) be a binary monotone system of order n. Then we have:*

$$\phi(\boldsymbol{x}) = x_i\phi(1_i, \boldsymbol{x}) + (1 - x_i)\phi(0_i, \boldsymbol{x}), \quad i = 1, 2, \ldots, n. \tag{3.1}$$

Similarly, for the reliability function of a monotone system where the component state variables are independent, we have

$$h(\boldsymbol{p}) = p_i h(1_i, \boldsymbol{p}) + (1 - p_i)h(0_i, \boldsymbol{p}), \quad i = 1, 2, \ldots, n. \tag{3.2}$$

Proof: Let $i \in C$. Since x_i is binary, we consider two cases: $x_i = 1$ and $x_i = 0$. If $x_i = 1$, then the right-hand side of (3.1) becomes $\phi(1_i, \boldsymbol{x})$. Hence (3.1) holds in this case. On the other hand, if $x_i = 0$, the right-hand side of (3.1) becomes $\phi(0_i, \boldsymbol{x})$, so (3.1) holds in this case as well. Equation (3.2) is proved by replacing the vector $\boldsymbol{x}$ by the stochastic vector $\boldsymbol{X}$ in (3.1), and taking the expected value on both sides of this equation. □

Note that (3.1) and (3.2) can alternatively be written respectively as:

$$\phi(\boldsymbol{x}) = \phi(0_i, \boldsymbol{x}) + [\phi(1_i, \boldsymbol{x}) - \phi(0_i, \boldsymbol{x})]x_i, \quad i = 1, \ldots n, \tag{3.3}$$

$$h(\boldsymbol{p}) = h(0_i, \boldsymbol{p}) + [h(1_i, \boldsymbol{p}) - h(0_i, \boldsymbol{p})]p_i \quad i = 1, \ldots n. \tag{3.4}$$

From these expressions it follows that ϕ is a linear function of x_i, while h is a linear function of p_i, $i = 1, \ldots, n$. A function which is linear in each argument is said to be *multilinear*.

When we do a pivotal decomposition of a binary monotone system (C, ϕ) of order n with respect to a component $i \in C$, we express the structure function as a linear combination of two simpler functions, $\phi(0_i, \boldsymbol{x})$ and $\phi(1_i, \boldsymbol{x})$. These functions can be interpreted as structure functions of the binary monotone systems $(C \backslash i, \phi(0_i, \cdot)$ and $(C \backslash i, \phi(1_i, \cdot))$ respectively, both of order $n-1$. The system $(C \setminus i, \phi(0_i, \cdot))$ is called the *restriction* of (C, ϕ) with respect to component i, while $(C \setminus i, \phi(1_i, \cdot)$ is called the *contraction* of (C, ϕ) with respect to component i. Restriction and contraction are also referred to as *minor operations*, and we say that a system (D, ψ) is a *minor* of (C, ϕ) if (D, ψ) can be obtained from (C, ϕ) by a sequence of minor operations.

If $\mathcal{C}$ is a class of binary monotone systems such that if $(C, \phi) \in \mathcal{C}$ implies that $(D, \psi) \in \mathcal{C}$ for all minors (D, ψ) of (C, ϕ), then we say that the class $\mathcal{C}$ is *closed under minor operations*. Obviously, the class of *all* binary monotone systems is closed under minor operations since any minor of a binary

monotone system is itself a binary monotone system. Interestingly, the same does *not* hold for the class of non-trivial binary monotone systems. It is easy to find examples where (D, ψ) is a minor of a non-trivial binary monotone system (C, ϕ), but where (D, ψ) is trivial. Assume e.g., that (C, ϕ) is the binary system where $C = \{1, 2\}$ and $\phi(\boldsymbol{x}) = x_1 \amalg x_2$, and let (D, ψ) be the contraction of (C, ϕ) with respect to component 1. Then $D = \{2\}$ and $\psi(x_2) \equiv 1$. Thus, (D, ψ) is trivial. This example also shows that the class of coherent systems is not closed under minor operations.

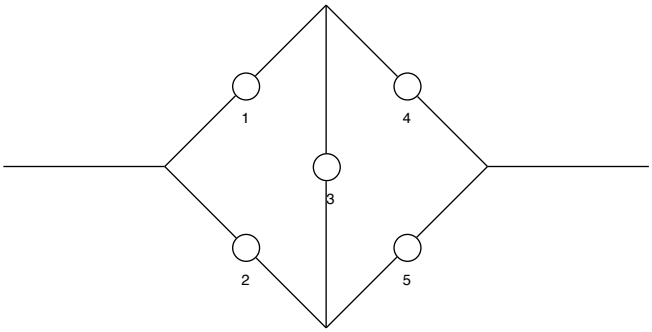

Figure 3.1: A bridge structure.

Example 3.1.2 *Let* (C, ϕ) *be the* bridge structure *shown in Figure 3.1, where* $C = \{1, \ldots, 5\}$. *In order to derive the structure function of this system, we note that:*

$\phi(1_3, \boldsymbol{X}) =$ *The state of the system given that component* 3 *is functioning,*
$\phi(0_3, \boldsymbol{X}) =$ *The state of the system given that component* 3 *is failed.*

Given that component 3 *is functioning, the system becomes a series connection of two parallel systems, the first one is the parallel system of component* 1 *and* 2*, while the second one is the parallel system of component* 4 *and* 5*. Hence, we get that:*

$$\phi(1_3, \boldsymbol{X}) = (X_1 \amalg X_2) \cdot (X_4 \amalg X_5).$$

If on the other hand, component 3 *is failed, the system becomes a parallel connection of two series systems, the first one is the series system of component* 1 *and* 4*, while the second one is the parallel system of component* 2 *and* 5*. From this it follows that:*

$$\phi(0_3, \boldsymbol{X}) = (X_1 \cdot X_4) \amalg (X_2 \cdot X_5).$$

By Theorem 3.1.1 it follows that ϕ *can be written as:*

$$\phi(\boldsymbol{X}) = X_3 \cdot \phi(1_3, \boldsymbol{X}) + (1 - X_3) \cdot \phi(0_3, \boldsymbol{X}).$$

Combining all this we get that ϕ *is given by:*

$$\phi(\boldsymbol{X}) = X_3 \cdot (X_1 \amalg X_2)(X_4 \amalg X_5) + (1 - X_3) \cdot (X_1 \cdot X_4 \amalg X_2 \cdot X_5). \quad (3.5)$$

Assuming that the component state variables are independent, we get that:

$$\begin{aligned} h(\boldsymbol{p}) &= E[\phi(\boldsymbol{X})] \\ &= E[X_3(X_1 \amalg X_2)(X_4 \amalg X_5) + (1 - X_3)(X_1X_4 \amalg X_2X_5)] \\ &= p_3(p_1 \amalg p_2)(p_4 \amalg p_5) + (1 - p_3)(p_1p_4 \amalg p_2p_5) \end{aligned}$$

A consequence of Theorem 3.1.1 is that for coherent systems, the reliability function is strictly increasing if the reliabilities are strictly between 0 and 1:

Theorem 3.1.3 *Let* $h(\boldsymbol{p})$ *be the reliability function of a binary monotone system* (C, ϕ) *of order* n*, and assume that* $0 < p_j < 1$ *for* $j \in C$*. If component* i *is relevant, then* $h(\boldsymbol{p})$ *is strictly increasing in* p_i*.*

Proof: From equation (3.2), it follows that

$$\begin{aligned} \frac{\partial h(\boldsymbol{p})}{\partial p_i} &= h(1_i, \boldsymbol{p}) - h(0_i, \boldsymbol{p}) \quad (3.6) \\ &= E[\phi(1_i, \boldsymbol{X})] - E[\phi(0_i, \boldsymbol{X})] = E[\phi(1_i, \boldsymbol{X}) - \phi(0_i, \boldsymbol{X})] \\ &= \sum_{(\cdot_i, \boldsymbol{x}) \in \{0,1\}^{n-1}} \big[\phi(1_i, \boldsymbol{x}) - \phi(0_i, \boldsymbol{x})\big] P\big((\cdot_i, \boldsymbol{X}) = (\cdot_i, \boldsymbol{x})\big) \end{aligned}$$

Since (C, ϕ) is a monotone system, ϕ is non-decreasing in each argument, and hence all the terms in this sum are non-negative. We then assume that component i is relevant. That is, there exists $(\cdot_i, \mathbf{y})$ such that $\phi(1_i, \boldsymbol{y}) - \phi(0_i, \boldsymbol{y}) = 1$. Moreover, since we have assumed that $0 < p_j < 1$ for $j \in C$ we have:

$$P\big((\cdot_i, \boldsymbol{X}) = (\cdot_i, \boldsymbol{y})\big) = \prod_{j=1, j\neq i}^{n} p_j^{y_j}(1-p_j)^{1-y_j} > 0$$

Thus, at least one of the terms in the last sum in (3.6) is positive which implies that

$$\frac{\partial h(\boldsymbol{p})}{\partial p_i} > 0.$$

Hence, we conclude that $h(\boldsymbol{p})$ is strictly increasing in each p_i. □

Note that if (C, ϕ) is *coherent* and $0 < p_j < 1$ for $j \in C$, it follows from Theorem 3.1.3 that h is strictly increasing in *each argument*, since in this case, all components are relevant.

3.2 Representation of binary monotone systems by paths and cuts

Now, we will look at a way to represent monotone systems via certain subsets of the component set, called path and cut sets. This turns out to be very useful in order to compute the exact system reliability, which will be done in Chapter 4.

In the following, we consider deterministic vectors $\boldsymbol{x}, \boldsymbol{y}$. We say that a vector $\boldsymbol{y}$ is smaller than another vector $\boldsymbol{x}$, that is $\boldsymbol{y} < \boldsymbol{x}$ if $y_i \leq x_i$ for $i = 1, \ldots, n$ and $y_i < x_i$ for at least one i.

Notation 3.2.1 *For any binary monotone system (C, ϕ) of order n, and vector $\boldsymbol{x} \in \{0,1\}^n$ the component set C can be divided into two subsets*

$$C_0(\boldsymbol{x}) = \{i : x_i = 0\}, \qquad C_1(\boldsymbol{x}) = \{i : x_i = 1\}.$$

Thus, if $\boldsymbol{X} = \boldsymbol{x}$, the subset $C_1(\boldsymbol{x})$ denotes the set of components which are functioning, while $C_0(\boldsymbol{x})$ denotes the set of components which are failed.

Definition 3.2.2 *Let (C, ϕ) be a binary monotone system.*

- *A vector $\boldsymbol{x}$ is a* path vector *if and only if $\phi(\boldsymbol{x}) = 1$. The corresponding* path set *is $C_1(\boldsymbol{x})$.*
- *A* minimal path vector *is a path vector, $\boldsymbol{x}$, such that $\boldsymbol{y} < \boldsymbol{x}$ implies that $\phi(\boldsymbol{y}) = 0$. The corresponding* minimal path set *is $C_1(\boldsymbol{x})$.*
- *A vector $\boldsymbol{x}$ is a* cut vector *if and only if $\phi(\boldsymbol{x}) = 0$. The corresponding* cut set *is $C_0(\boldsymbol{x})$.*
- *A* minimal cut vector *is a cut vector, $\boldsymbol{x}$, such that $\boldsymbol{x} < \boldsymbol{y}$ implies that $\phi(\boldsymbol{y}) = 1$. The corresponding* minimal cut set *is $C_0(\boldsymbol{x})$.*

Thus, a path vector is a vector $\boldsymbol{x}_1$ such that the system is functioning if $\boldsymbol{X} = \boldsymbol{x}_1$. A cut vector is a vector $\boldsymbol{x}_0$ such that the system is failed if $\boldsymbol{X} = \boldsymbol{x}_0$. A minimal path vector is a path vector which cannot be decreased in any way and still be a path vector. Similarly, a minimal cut vector is a cut vector which cannot be increased in any way and still be a cut vector. A minimal path set is a minimal set of components which ensures that the system is working if these components are working. A minimal cut set is a minimal set of components such that if these are sabotaged, the system will be sabotaged.

Example 3.2.3 *Consider the bridge structure in Figure 3.2. To find the minimal path sets, look at Figure 3.2 and try to locate paths where it is possible to send a signal through the system. For instance, one such path is through components* 1 *and* 4*. Hence,* $\{1, 4\}$ *is a path set. We then see that this set is in fact minimal, because if only components* 1 *and* 4 *are functioning, and one of them fails, the system will fail as well. Similarly, we can argue for e.g. the path through components* 2 *and* 5*. Continuing like this, we find that the minimal path sets are:*

$$P_1 = \{1, 4\}, \quad P_2 = \{2, 5\}, \quad P_3 = \{1, 3, 5\} \text{ and } P_4 = \{2, 3, 4\}.$$

(Verify that these are in fact minimal!)

To find the minimal cut sets, look at Figure 3.2 and see what combination of components not functioning will sabotage the whole system. For example, if components 1 *and* 2 *are not functioning, the system is sabotaged because there is no way for a signal to make it past the initial point. The set* $\{1, 2\}$

is in fact a minimal cut set, because if all other components are functioning except 1 *and* 2*, and one of them is fixed, the system will start to function. A similar argument shows that* $\{4, 5\}$ *is a minimal cut set. Continuing in the same way, we find that the minimal cut sets are:*

$$K_1 = \{1, 2\}, \quad K_2 = \{4, 5\}, \quad K_3 = \{1, 3, 5\} \text{ and } K_4 = \{2, 3, 4\}.$$

(Make sure you can explain why all of these are minimal cut sets!)

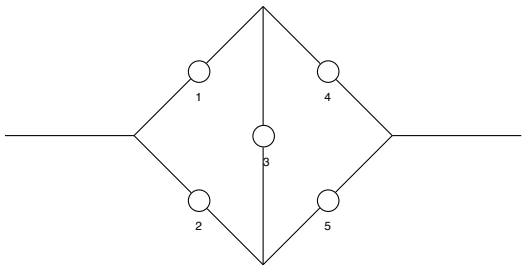

Figure 3.2: A bridge structure.

Before we continue with how to represent structures via their minimal paths and cuts, we need some notation:

Notation 3.2.4 *Let* $A \subset C$*,* $A = \{a_1, a_2, \ldots, a_k\}$*, where* k *is the number of elements in* A*. Also, let* $\boldsymbol{x}$ *be some component state vector. Then, we let* $\boldsymbol{x}^A$ *be the vector where we select the* k *components from* $\boldsymbol{x}$ *corresponding to the components in* A*:*

$$\boldsymbol{x}^A = (x_{a_1}, x_{a_2}, \ldots, x_{a_k}).$$

Now, we are ready to explain how structures can be represented via their minimal path and cut sets. In order to do so, we consider a binary monotone system (C, ϕ), and let $P_j \subseteq C$ be the jth minimal path set of this system. Then we can define the following binary monotone system (P_j, ρ_j), where $\rho_j = \rho_j(\boldsymbol{x}^{P_j})$ is given by:

$$\rho_j(\boldsymbol{x}^{P_j}) = \prod_{i \in P_j} x_i. \tag{3.7}$$

(P_j, ρ_j) is referred to as the jth *minimal path series structure* of the system, $j = 1, \ldots, p$. From this, we find that:

$$\phi(\boldsymbol{x}) = \coprod_{j=1}^{p} \rho_j(\boldsymbol{x}^{P_j}) = \coprod_{j=1}^{p} \prod_{i \in P_j} x_i. \tag{3.8}$$

That is, the system (C, ϕ) can be represented as a parallel structure of the minimal path series structures. See Figure 3.2 for an illustration of how the bridge structure in Example 3.2.3 can be represented as a parallel structure of its minimal path series structures. The reason for this is that the system functions if and only if at least one of the minimal path series structures are functioning.

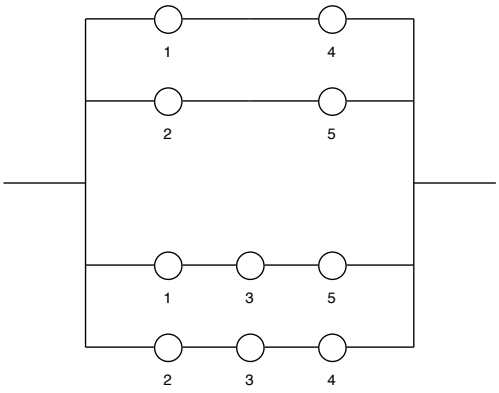

Figure 3.3: The reliability block diagram of a bridge structure in Example 3.2.3 represented as a parallel structure of its minimal path series structures.

Similarly, let $K_j \subseteq C$ be the jth minimal cut set of this system. Then we can define the following binary monotone system (K_j, κ_j), where $\kappa_j = \kappa_j(\boldsymbol{x}^{K_j})$ is given by:

$$\kappa_j(\boldsymbol{x}^{K_j}) = \coprod_{i \in K_j} x_i. \tag{3.9}$$

(K_j, κ_j) is referred to as the jth *minimal cut parallel structure* of the system,

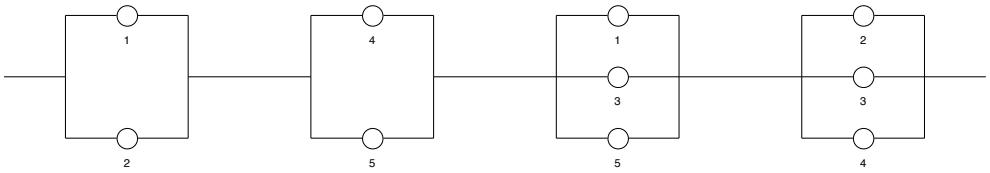

Figure 3.4: The reliability block diagram of a bridge structure in Example 3.2.3 represented as a series structure of its minimal cut parallel structures.

$j = 1, \ldots, k$. From this, we find that:

$$\phi(\boldsymbol{x}) = \prod_{j=1}^{k} \kappa_j(\boldsymbol{x}^{K_j}) = \prod_{j=1}^{k} \coprod_{i \in K_j} x_i. \tag{3.10}$$

That is, the system (C, ϕ) can be represented as a series structure of the minimal cut parallel structures. See Figure 3.4. The reason for this is that for the system to be failed, it suffices for one of the minimal cut parallel structures is failed.

These representations of the structure function are very useful, and will be applied in Chapter 4 to compute the exact system reliability. However, a major challenge is that computing the minimal path and cut sets for a large system is computationally very time consuming. Therefore, it is important to derive fast algorithms for finding the minimal path and cut sets. This is an important challenge in applied reliability theory, but beyond the scope of this book.

The proof of the following theorem is left as an exercise:

Theorem 3.2.5 *Let (C, ϕ) be a binary monotone system, and let (C^D, ϕ^D) be its dual. Then the following statements hold:*

(i) $\boldsymbol{x}$ is a path vector (alternatively, cut vector) for (C, ϕ) if and only if $\boldsymbol{x}^D$ is a cut vector (path vector) for (C^D, ϕ^D).

(ii) A minimal path set (alternatively, cut set) for (C, ϕ) is a minimal cut set (path set) for (C^D, ϕ^D).

We close this section by noting that by using the relation (2.3) between the product and minimum operators, and the relation (2.5) between the coproduct and maximum operators, the equations (3.8) and (3.10) can be exprressed as follows:

$$\phi(\boldsymbol{X}) = \min_{1\le j\le k} \max_{i\in K_j} X_i = \max_{1\le j\le p} \min_{i\in P_j} X_i. \tag{3.11}$$

3.3 Modules of monotone systems

Sometimes, it is possible to divide a monotone system into subsystems which are analysed separately before finally analysing how these subsystems are connected.

Let $A \subseteq C$. Then, the complement set of A, i.e., $C \setminus A$, is denoted by $\bar{A}$. We have the following formal definition of a module:

Definition 3.3.1 *Let (C, ϕ) be a binary monotone system, and $A \subseteq C$. The monotone system (A, χ) is a* module *of (C, ϕ) if and only if the structure function ϕ can be written as:*

$$\phi(\boldsymbol{x}) = \psi(\chi(\boldsymbol{x}^A), \boldsymbol{x}^{\bar{A}}), \qquad \textit{for all } \boldsymbol{x} \in \{0,1\}^n,$$

where ψ is a monotone structure function. The set A is called a modular set *of (C, ϕ).*

The structure function χ expresses the state of the module as a function of the states of the components in the modular set, while ψ expresses how the state of the system depends on the state of the module as well as on the states of components outside of the module. You can think of a module as a "large component" which can be circled in and separated out from the rest of the system. More precisely, a module consists of a subset of the component set which is such that the structure function of the system depends on the states of the components within this set only through the structure function of the module. Thus, in order to determine the state of the system, we only need to know the state of the module, not the states of the individual components within the module.

Definition 3.3.2 *A* modular decomposition *of a monotone system (C, ϕ) is a set of modules $\{(A_j, \chi_j)\}_{j=1}^r$ connected by a binary monotone organisation structure function ψ. The following conditions must be satisfied:*

(i) $C = \bigcup_{j=1}^{r} A_j$, *where* $A_j \cap A_k = \emptyset$ *for* $j \neq k$.

(ii) $\phi(\boldsymbol{x}) = \psi[\chi_1(\boldsymbol{x}^{A_1}), \ldots, \chi_r(\boldsymbol{x}^{A_r})]$.

We observe that a modular decomposition is a disjoint partition of the component set into modules such that the structure function of the whole system is a function of the structure functions of these modules.

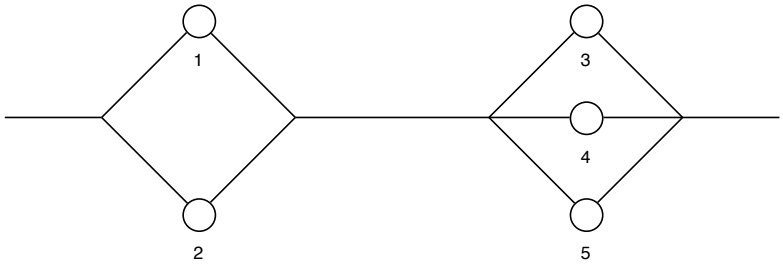

Figure 3.5: A monotone system which can be modularly decomposed.

Example 3.3.3 *In order to find a modular decomposition for the system in Figure 3.5, we can let* $A_1 = \{1, 2\}$, $A_2 = \{3, 4, 5\}$,

$$\chi_1(\boldsymbol{X}^{A_1}) = X_1 \amalg X_2, \ \chi_2(\boldsymbol{X}^{A_2}) = X_3 \amalg X_4 \amalg X_5$$

and

$$\psi(\chi_1(\boldsymbol{X}^{A_1}), \chi_2(\boldsymbol{X}^{A_2})) = \chi_1(\boldsymbol{X}^{A_1}) \cdot \chi_2(\boldsymbol{X}^{A_2}).$$

If the component state variables are independent, we may calculate the reliability of the system in a two-step process. In the first step we compute the reliability of the modules. Then in the second step we treat the modules as components and compute the system reliability by using the reliabilities of the modules as input. Thus, whenever one can identify modules in a system, this can be utilised in order to simplify and speed up the reliability calculations.

Modular decompositions can also be used in cases where it is not possible to compute the system reliability exactly as it is too time consuming. In such cases, we may have to resort to finding upper and lower bounds for the system reliability. It turns out that modules may be used to improve bounds. We will return to this in Section 8.2.

3.4 Exercises

Exercise 1. Prove equation (3.2) in another way than what is done in the proof of Theorem 3.1.1.

(*Hint:* Condition on the state of component i.)

Exercise 2. Prove Theorem 3.2.5.

Exercise 3. Find the minimal path sets and minimal cut sets of the brigde structure in Figure 3.2.

Exercise 4. Find the representations via the minimal path sets and the minimal cut sets of the following systems:

(i) The 2-out-of-3 system.

(ii) The 3-out-of-4 system.

(iii) The series system of 3 components.

(iv) The parallel system of 4 components.

Exercise 5. Consider a coherent system (C, ϕ) with minimal path sets $P_1, \ldots, P_p$ and minimal cut sets $K_1, \ldots, K_k$. Prove that

$$\bigcup_{j=1}^{p} P_j = C = \bigcup_{j=1}^{k} K_j.$$

Exercise 6. Find the minimal path sets and the minimal cut sets of the system in Figure 3.6. Find two different expressions for the structure function of this system.

Exercise 7. Show that if (C, ϕ) is a bridge system (see Figure 3.1), then (C^D, ϕ^D) is a bridge system as well. [*Hint*: Compare the minimal path and cut sets of (C, ϕ).]

Exercise 8. Let (A, χ) be a module of (C, ϕ). Assume that $\boldsymbol{x}_1$ and $\boldsymbol{x}_0$ are such that $\chi(\boldsymbol{x}_1^A) = 1$ and $\chi(\boldsymbol{x}_0^A) = 0$. Prove that for all $(\cdot^A, \boldsymbol{x}^{\bar{A}})$ we have:

$$\phi(\boldsymbol{x}_1^A, \boldsymbol{x}_1^{\bar{A}}) = \phi(\mathbf{1}^A, \boldsymbol{x}_1^{\bar{A}}) \text{ and } \phi(\boldsymbol{x}_0^A, \boldsymbol{x}_0^{\bar{A}}) = \phi(\mathbf{0}^A, \boldsymbol{x}_1^{\bar{A}}).$$

Exercise 9. Find all the modules of the following structure function:

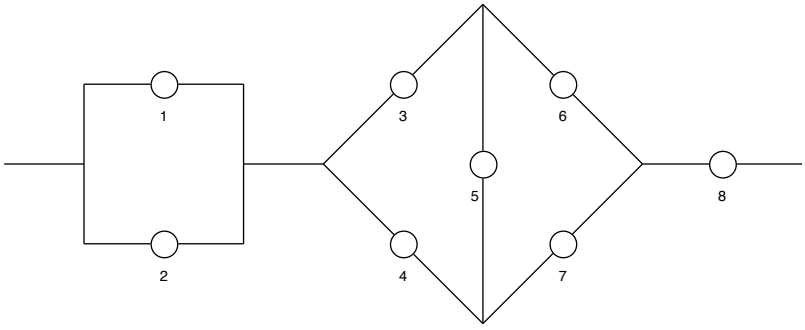

Figure 3.6: A series connection of a parallel system, a bridge system and a single component.

$$\phi(\boldsymbol{x}) = (x_1(x_2 \amalg x_3)) \amalg (x_4 \amalg x_5).$$

Exercise 10. What are the modules of a k-out-of-n system where $1 < k < n$? Give a reason for your answer.

Does this mean that a k-out-of-n system is well suited for modular decomposition?

Exercise 11. Consider the system shown in Figure 3.7.
a) Find the structure function of the system by dividing the system into modules.

b) Find the structure function of the system *without* dividing the system into modules.

c) Compare the computational efforts in a) and b).

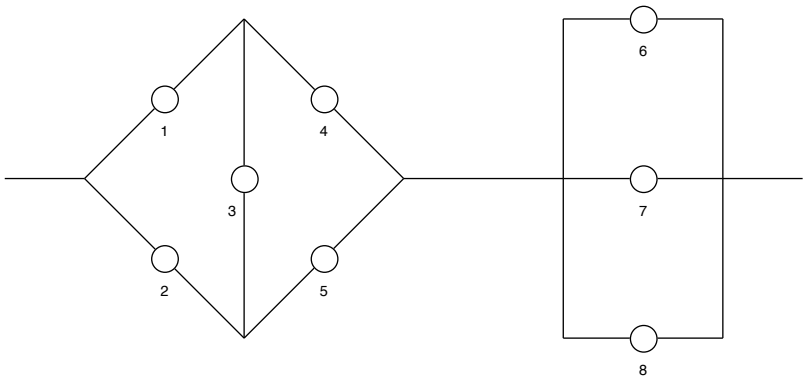

Figure 3.7: A system suited for modular decompostition.

Chapter 4

Exact computation of the reliability of binary monotone systems

In this chapter, we will look at various methods for computing the exact reliability of a binary monotone system. In the general case this is a *hard* problem in the sense that the running time of all known algorithms grows exponentially in the size of the problem, i.e., in the order of the system.

If the running time of an algorithm grows approximately proportionally to a positive function $f(n)$, where n is measuring the size of the problem (i.e, the number of components in a binary monotone system), we say that the algorithm is of *order* $O(f(n))$. More formally, let $t(n)$ denote the worst case running time of the algorithm as a function of the size n. Then the order of the algorithm is $O(f(n))$ if and only if there exists a positive constant M and a positive integer n_0 such that:

$$t(n) \leq Mf(n), \text{ for all } n \geq n_0.$$

If f is a polynomial in n, we say that the algorithm is a *polynomial time* algorithm, while if f is an exponential function of n, we say that the algorithm is an *exponential time* algorithm.

In computational complexity theory, NP (for nondeterministic polynomial time) is a complexity class used to describe certain types of problems. NP contains many important problems, the hardest of which are called NP-complete problems. The most important open question in complexity theory is whether polynomial time algorithms actually exist for solving NP-complete

problems. It is widely believed that this is not the case. The class of *NP-hard* problems is a class of problems that are, informally, *at least as hard as the hardest problems in NP*. A comprehensive treatment of complexity is beyond the scope of this book. See Garey and Johnson [27] for more on this.

Unfortunately, the problem of computing the reliability of a binary monotone system is known to be NP-hard in the general case. See Rosenthal [60] and Ball [3] for details. Thus, all known algorithms for analytical reliability calculations typically has a computational complexity which grows exponentially with the order of the system. Still for moderately sized systems it is possible to calculate the reliability using standard methods. Moreover, for certain classes of systems faster algorithms are available.

In the next sections we present both general algorithms as well as specialized algorithms tailored for certain classes of systems.

4.1 State space enumeration

The idea of this method is simple. We know that the reliability of a binary monotone system (C, ϕ) is simply the expected value of $\phi(\boldsymbol{X})$. Thus, by standard probability theory this can be calculated as:

$$h = E[\phi(\boldsymbol{X})] = \sum_{\boldsymbol{x} \in \{0,1\}^n} \phi(\boldsymbol{x}) P(\boldsymbol{X} = \boldsymbol{x}) \tag{4.1}$$

Note that in order to calculate the reliability using (4.1) we must enumerate all possible states of the component vector (i.e., all $\boldsymbol{x} \in \{0,1\}^n$). Moreover, if the components are dependent, we must know the entire joint distribution of $\boldsymbol{X}$ in order to compute the system reliability using (4.1). As this distribution is usually not known, this method is unsuitable for the case of dependent components. However, if the components are independent, we have:

$$P(\boldsymbol{X} = \boldsymbol{x}) = \prod_{i=1}^{n} p_i^{x_i} (1 - p_i)^{1 - x_i}$$

Inserting this into equation (4.1) we get:

$$h(\boldsymbol{p}) = \sum_{\boldsymbol{x} \in \{0,1\}^n} \phi(\boldsymbol{x}) \prod_{i=1}^{n} p_i^{x_i} (1 - p_i)^{1 - x_i}. \tag{4.2}$$

To compute the system reliability via (4.2), it suffices to know the individual component reliabilities. However, since we need to enumerate all the possible states of the component state vector $\boldsymbol{x}$, the number of terms in the sum (4.2) is 2^n. Thus, this algorithm is at least of order $O(2^n)$. In fact the order may be even greater since we have not included the work of computing $\phi(\cdot)$ for each of the possible states of the component state vector $\boldsymbol{x}$. For some structure functions this may also represent a considerable task.

While the order of the algorithm cannot be changed, there are still ways to improve the method. Sometimes it is possible to run through the states in a clever order. For example, if we come across $\boldsymbol{x}_0$ such that $\phi(\boldsymbol{x}_0) = 0$, we do not have to include any of the states $\boldsymbol{x}$ such that $\boldsymbol{x} < \boldsymbol{x}_0$ since ϕ is non-decreasing (as it is assumed to be monotone), so $\phi(\boldsymbol{x})$ must be 0 as well. Note also that since $\phi(\mathbf{0}) = 0$, the maximum amount of terms we need to compute in (4.2) is $2^n - 1$.

Example 4.1.1 *Consider a binary monotone system (C, ϕ) where $C = \{1, 2, 3\}$, and where the structure function ϕ is given by:*

$$\phi(\boldsymbol{x}) = (x_1 \cdot x_2) \amalg x_3.$$

We assume that the component state variables are independent, and that the component reliabilities are $P(X_i = 1) = p_i$, $i = 1, 2, 3$. By using (4.2) we find that:

$$\begin{aligned} h(\boldsymbol{p}) &= \phi(0,0,0) \cdot (1-p_1)(1-p_2)(1-p_3) \\ &\quad + \phi(1,0,0) \cdot p_1(1-p_2)(1-p_3) + \cdots + \phi(1,1,1) \cdot p_1p_2p_3 \\ &= p_1p_2(1-p_3) \\ &\quad + (1-p_1)(1-p_2)p_3 + p_1(1-p_2)p_3 + (1-p_1)p_2p_3 + p_1p_2p_3. \end{aligned}$$

Note that the last expression contains just 5 terms, not $2^3 = 8$ terms. The reason for this is that there are exactly 5 vectors such that $\phi(\boldsymbol{x}) = 1$. The other 3 terms vanish in the final formula. Moreover, we observe that the last 4 terms have p_3 as a common factor. Thus, the expression for $h(\boldsymbol{p})$ can be simplified to:

$$\begin{aligned} h(\boldsymbol{p}) &= p_1p_2(1-p_3) \\ &\quad + [(1-p_1)(1-p_2) + p_1(1-p_2) + (1-p_1)p_2 + p_1p_2] \cdot p_3 \\ &= p_1p_2(1-p_3) + p_3 \end{aligned}$$

This example shows that the expansion obtained by state space enumeration may not be the most efficient expression for the reliability function.

4.2 The multiplication method

The first step of the multiplication method for computing the exact system reliability, it to find the minimal path sets or alternatively, the minimal cut sets of the system. Denote these by respectively $P_1, \ldots, P_p$ and $K_1, \ldots, K_k$. Now, from the formula (3.8) and (3.10), we know that:

$$\begin{aligned}\phi(\boldsymbol{X}) &= \coprod_{j=1}^{p} \prod_{i \in P_j} X_i = 1 - [\prod_{j=1}^{p} (1 - \prod_{i \in P_j} X_i)] \\ &= \prod_{j=1}^{k} \coprod_{i \in K_j} X_i = \prod_{j=1}^{k} [1 - \prod_{i \in K_j} (1 - X_i)].\end{aligned}$$

By expanding either the formula based on the minimal path sets, or the formula based on the minimal cut sets, and using that $X_i^r = X_i$, $i = 1, \ldots, n$, $r = 1, 2, \ldots$, we eventually get an expression of the form:

$$\phi(\boldsymbol{X}) = \sum_{A \subseteq C} \delta(A) \prod_{i \in A} X_i \tag{4.3}$$

where for all $A \subseteq C$, $\delta(A)$ denotes the coefficient of the term associated with $\prod_{i \in A} X_i$. The δ-function is called the *signed domination function* of the structure. The fact that we end up with an expression of the form (4.3) follows since ϕ is multilinear (see the comments after Theorem 3.1.1), and all multilinear functions can be written in this form.

By taking the expectation on both sides of (4.3), and assuming that the component state variables are independent, we obtain the following expression:

$$h(\boldsymbol{p}) = E[\phi(\boldsymbol{X})] = \sum_{A \subseteq C} \delta(A) \prod_{i \in A} E[X_i] = \sum_{A \subseteq C} \delta(A) \prod_{i \in A} p_i \tag{4.4}$$

As for the state space enumeration method, we again end up with a sum containing 2^n terms. Thus, calculating the reliability of the system using (4.4) also has the order $O(2^n)$. In a given case, however, there will typically be far less terms in this expansion since $\delta(A) = 0$ for many of the sets A.

Note, that when using this method, we also need to determine the signed domination function before we can use (4.4). It turns out that identifying all the minimal path or cut sets is another operation which has the order $O(2^n)$. Moreover, depending on the number of minimal path or cut sets, the

task of expanding the structure function into the form (4.3) is yet another complicated task.

The multiplication method can be improved by avoiding identifying the minimal path or cut sets as well as the multiplication step. In order to explain this, we introduce the following alternative way of expressing the structure function of a binary monotone system (C, ϕ):

$$\phi(B) = \phi(\mathbf{1}^B, \mathbf{0}^{\bar{B}}), \text{ for all } B \subseteq C.$$

Thus, for all $B \subseteq C$, $\phi(B)$ denotes the state of the system given that all the components in the set B are functioning, while all the components in the set $\bar{B} = C \setminus B$ are failed. In Huseby [32] the following formula was proved:

$$\delta(A) = \sum_{B \subseteq A} (-1)^{|A|-|B|} \phi(B), \text{ for all } A \subseteq C. \tag{4.5}$$

This formula allows us to compute the signed domination function without using the minimal path and cut sets. In its most general form the sum in (4.5) still contains 2^n terms. Thus, evaluating this formula is yet another complex operation of order 2^n. However, for certain classes of systems (4.5) can be used as basis for deriving simpler formulas allowing much faster calculations. We will return to this in Section 4.6.

Example 4.2.1 *Consider a binary monotone system (C, ϕ) where $C = \{1, 2, 3\}$, and where the minimal path sets of the system are $P_1 = \{1, 2\}$ and $P_2 = \{1, 3\}$. Using the multiplication method we obtain:*

$$\begin{aligned}
\phi(\boldsymbol{X}) &= (X_1 X_2) \amalg (X_1 X_3) = 1 - (1 - X_1 X_2)(1 - X_1 X_3) \\
&= 1 - (1 - X_1 X_2 - X_1 X_3 + X_1^2 X_2 X_3) \\
&= X_1 X_2 + X_1 X_3 - X_1 X_2 X_3,
\end{aligned}$$

where we have used that $X_1^2 = X_1$. Thus, we see that ϕ is of the form (4.3), where $\delta(\{1, 2\}) = \delta(\{1, 3\}) = 1$, $\delta(\{1, 2, 3\}) = -1$, while $\delta(A) = 0$ for all other subsets of C.

Note that these coefficients can be obtained using (4.5) as well since:

$$\begin{aligned}
\delta(\{1,2\}) &= (-1)^{|\{1,2\}|-|\{1,2\}|}\phi(\{1,2\}) = 1, \\
\delta(\{1,3\}) &= (-1)^{|\{1,3\}|-|\{1,3\}|}\phi(\{1,3\}) = 1, \\
\delta(\{1,2,3\}) &= (-1)^{|\{1,2,3\}|-|\{1,2\}|}\phi(\{1,2\}) + (-1)^{|\{1,2,3\}|-|\{1,3\}|}\phi(\{1,2\}) \\
&\quad + (-1)^{|\{1,2,3\}|-|\{1,2,3\}|}\phi(\{1,2,3\}) = -1 - 1 + 1 = -1.
\end{aligned}$$

Having derived the formula for the structure function ϕ, we immediately obtain the reliability function:

$$h(\boldsymbol{p}) = p_1p_2 + p_1p_3 - p_1p_2p_3.$$

4.3 The inclusion-exclusion method

Just like for the multiplication method, the inclusion exclusion method is based on first finding the minimal path sets and the minimal cut sets of the system. Let these sets be denoted respectively by $P_1, \ldots, P_p$ and $K_1, \ldots, K_k$. We then introduce the events

$$E_j = \{\text{All of the components in } P_j \text{ are functioning}\}, \quad j = 1, \ldots, p.$$

Since the system is functioning if and only if at least one of the minimal path sets is functioning, we have:

$$\phi = 1 \quad \text{if and only if} \quad \bigcup_{j=1}^{p} E_j \text{ holds true.} \tag{4.6}$$

Calculating the probability of a union of events can be done by using the well-known *inclusion-exclusion* formula. That is we have:

$$\begin{aligned} h &= P(\bigcup_{j=1}^{p} E_j) \\ &= P(E_1) + P(E_2) + \cdots + P(E_p) \\ &\quad - P(E_1 \cap E_2) - P(E_1 \cap E_3) - \cdots - P(E_{p-1} \cap E_p) \\ &\quad + P(E_1 \cap E_2 \cap E_3) + \cdots + P(E_{p-2} \cap E_{p-1} \cap E_p) \\ &\quad \cdots \\ &\quad + (-1)^{p-1} P(E_1 \cap \cdots \cap E_p). \end{aligned} \tag{4.7}$$

The initial number of terms in (4.7) is $2^p - 1$. However, typically many of the terms can be merged as they correspond to the same component set. Thus, after simplifying the expression we usually end up with much fewer terms.

Note that an event of form $E_{i_1} \cap \cdots \cap E_{i_r}$ occurs if and only if all the components in the set $P_{i_1} \cup \cdots \cup P_{i_r}$ are functioning. Thus, when the component state variables are independent, we get that:

$$P(E_{i_1} \cap \cdots \cap E_{i_r}) = \prod_{i \in P_{i_1} \cup \cdots \cup P_{i_r}} p_i.$$

Example 4.3.1 *Consider a binary monotone system (C, ϕ) where $C = \{1, 2, 3, 4\}$ with minimal path sets $P_1 = \{1, 2\}$, $P_2 = \{1, 3\}$, $P_3 = \{2, 3, 4\}$. We then let E_j denote the event that all components in P_j are functioning, $j = 1, 2, 3$. Assuming that the component state variables are independent, we then get the following probabilities:*

$$\begin{aligned} P(E_1) = p_1p_2, \quad P(E_2) = p_1p_3, \quad P(E_3) = p_2p_3p_4 \\ P(E_1 \cap E_2) = p_1p_2p_3, \quad P(E_1 \cap E_3) = P(E_2 \cap E_3) = p_1p_2p_3p_4 \\ P(E_1 \cap E_2 \cap E_3) = p_1p_2p_3p_4. \end{aligned}$$

Hence, the reliability of the system is:

$$\begin{aligned} h(\boldsymbol{p}) &= p_1p_2 + p_1p_3 + p_2p_3p_4 - p_1p_2p_3 - 2p_1p_2p_3p_4 + p_1p_2p_3p_4 \\ &= p_1p_2 + p_1p_3 + p_2p_3p_4 - p_1p_2p_3 - p_1p_2p_3p_4. \end{aligned}$$

We observe that the final expression for the reliability function is exactly the same as the one we get using the multiplication methods. That is, in both cases we end up with a formula of the form:

$$h(\boldsymbol{p}) = \sum_{A \subseteq C} \delta(A) \prod_{i \in A} p_i,$$

where δ denotes the signed domination function. However, the steps we take in order to get this expression is different.

When using the inclusion-exclusion formula we see that all terms in the expansion correspond to sets $A \subseteq C$ which are unions of minimal path sets. If $P_{i_1}, \ldots, P_{i_r}$ is a collection of minimal path sets such that:

$$\bigcup_{j=1}^{r} P_{i_j} = A,$$

the collection is said to be a *formation* of the set A. The formation is *odd* if r is an odd number, and *even* if r is an even number. We note that odd formations produce terms with a coefficient $+1$ in the inclusion-exclusion formula, while even formations produce terms with a coefficient -1 in the inclusion-exclusion formula. When we simplify the expansion, all terms corresponding to formations of the same set are merged. From this observation it follows that we have the following result:

Theorem 4.3.2 *Let (C, ϕ) be a binary monotone system with minimal path sets $P_1, \ldots, P_p$, and let δ denote the signed domination function of the system. Then for all $A \subseteq C$ we have:*

$$\begin{aligned} \delta(A) = &\text{ The number of odd formations of } A \\ &- \text{ The number of even formations of } A. \end{aligned} \tag{4.8}$$

In particular $\delta(A) = 0$ if A is not a union of minimal path sets.

As a corollary we obtain the following result:

Corollary 4.3.3 *If (C, ϕ) is a binary monotone system which is not coherent, then $\delta(C) = 0$.*

The proof is left as an exercise.

A nice feature of the inclusion-exclusion formula is that it can be used to produce a simple upper bound for the system reliability. Thus, if we skip all higher order terms and only keep the probabilities of the individual events $E_1, \ldots, E_p$, we get:

$$h \leq \sum_{j=1}^{p} P(E_j). \tag{4.9}$$

By including higher order terms, it is possible to obtain both lower bounds and improved upper bounds as well. However, as we introduce the higher order terms, the total number of terms in these expansions grows fast. Thus, only the simple bound (4.9) is considered here. In order to obtain a lower bound we instead use a *dual* approach. That is, we consider the minimal cut sets of the system $K_1, \ldots, K_k$, and introduce the events:

$$F_j = \{\text{All the components in } K_j \text{ are failed}\}, \quad j = 1, \ldots, k.$$

Since the system is failed if and only if all the components in at least one cut set are failed, we have:

$$1 - h = P(\bigcup_{j=1}^{k} F_j). \tag{4.10}$$

The quantity $1 - h$ is referred to as the *unreliability* of the system. By the same argument as we used above, an upper bound on this quantity is given by:

$$1 - h \leq \sum_{j=1}^{k} P(F_j). \tag{4.11}$$

By combining (4.9) and (4.11), we see that

$$1 - \sum_{j=1}^{k} P(F_j) \leq h \leq \sum_{j=1}^{p} P(E_j). \tag{4.12}$$

If the components are independent, (4.12) implies that:

$$1 - \sum_{j=1}^{k} \prod_{i \in K_j} (1 - p_i) \leq h(\boldsymbol{p}) \leq \sum_{j=1}^{p} \prod_{i \in P_j} p_i. \tag{4.13}$$

In many real life applications the component reliabilities are close to 1. In such cases the lower bound given in (4.13) turns out to be very good, while the upper bound is extremely poor. In fact it may easily happen that the upper bound becomes greater than 1. If on the other hand the component reliabilites are close to 0, the lower bound may be less than 0, while the upper bound turns out to be very good. In order to avoid bounds outside of the interval $[0, 1]$, one could replace (4.13) by:

$$\max(1 - \sum_{j=1}^{k} \prod_{i \in K_j} (1 - p_i),\, 0) \leq h(\boldsymbol{p}) \leq \min(\sum_{j=1}^{p} \prod_{i \in P_j} p_i,\, 1). \tag{4.14}$$

4.4 k-out-of-n systems

We recall that a k-out-of-n system is functioning if and only if at least k out of the n components are functioning. Thus, if (C, ϕ) is a k-out-of-n system with component state vector $\boldsymbol{X} = (X_1, \ldots, X_n)$, then $S = X_1 + \cdots + X_n$ is the number of functioning components, and the system is functioning if and only if $S \geq k$. From this it follows that the reliability of the system can be easily derived form the distribution of S. In the special case where $P(X_i = 1) = p$ for all $i \in C$, it follows that S is a binomially distributed variable. Thus, in this case the reliability of the system can be calculated using the formula (2.15).

In this section we consider the problem of computing the reliability of a k-out-of-n system in the general case where $P(X_i = 1) = p_i$, for $i \in C$, and where the p_is may be unequal. We also let $q_i = 1 - p_i$, for $i \in C$, and

introduce the following sequence of partial sums:

$$S_j = \sum_{i=1}^{j} X_i, \quad j = 1, \ldots, n.$$

Thus, in particular $S_1 = X_1$ while $S_n = S$. The distribution of S is found by computing sequentially the distributions of $S_1, \ldots, S_n$. Since $S_1 = X_1$, it follows that $P(S_1 = 0) = q_1$ and that $P(S_1 = 1) = p_1$. Thus, the distribution of S_1 is fully determined. We then assume that we have determined the distributions of $S_1, \ldots, S_j$ where $j < n$, and consider the distribution of S_{j+1}. By conditioning on X_{j+1} we get the following recursion formula for $s = 0, 1, \ldots, j + 1$:

$$P(S_{j+1} = s) = P(S_j = s) \cdot q_{j+1} + P(S_j = s - 1) \cdot p_{j+1}. \tag{4.15}$$

Note that if $s = 0$, we have $P(S_j = s - 1) = 0$, while if $s = j + 1$, we have $P(S_j = s) = 0$. Thus, in these cases (4.15) simplifies to:

$$P(S_{j+1} = 0) = P(S_j = 0) \cdot q_{j+1} \tag{4.16}$$

$$P(S_{j+1} = j + 1) = P(S_j = j) \cdot p_{j+1} \tag{4.17}$$

Using (4.15), (4.16) and (4.17), it is easy to see that given the distribution of S_j we need to do $2(j + 1)$ multiplications and j additions in order to compute the distribution of S_{j+1}. By repeating this process for $j = 2, \ldots, n$, we eventually arrive at the distribution of $S = S_n$. The total number of multiplications needed to calculate the distribution of S is then:

$$2 \cdot 2 + 2 \cdot 3 + \cdots 2 \cdot n = (n - 1)(n + 2),$$

while the number of additions needed is:

$$1 + 2 + \cdots + (n - 1) = \frac{(n - 1)n}{2}$$

Having found the distribution of S the reliability of the system is given by:

$$P(\phi = 1) = P(S \geq k) = \sum_{s=k}^{n} P(S = s)$$

The number of additions need to calculate this last expression is $n - k$. From this it follows that the total number of algebraic operations (multiplications

and additions) grows approximately proportional to n^2. Thus, we conclude that the order of this algorithm is $O(n^2)$. Since n^2 is a *polynomial* in n, this algorithm is a *polynomial time* algorithm, which means that we can easily handle very large systems of this type.

Example 4.4.1 *Let (C, ϕ) be a 6-out-of-8 system where the components have the following reliabilities:*

$$p_1 = 0.75, \quad p_2 = 0.80, \quad p_3 = 0.82, \quad p_4 = 0.65$$
$$p_5 = 0.88, \quad p_6 = 0.91, \quad p_7 = 0.92, \quad p_8 = 0.86$$

We also let $S = X_1 + \cdots + X_8$. Using (4.15), (4.16) and (4.17) we get that:

$$P(S = 6) = 0.2797, \quad P(S = 7) = 0.3701, \quad P(S = 8) = 0.2026$$

Hence, the reliability of the system is:

$$h = P(\phi = 1) = P(S \geq 6) = 0.2797 + 0.3701 + 0.2026 = 0.8524$$

A python script which can be used to calculate the reliability of this system, is given in Script B.1.1 in Appendix B.1.

4.5 Threshold systems

In the previous section we saw that the reliability of a k-out-of-n system can be calculated in polynomial time using a recursive relation. From this result one might think that it would be easy to obtain similar performance for a larger class of systems. A natural generalisation of k-out-of-n systems is the class of *threshold systems* defined as follows.

Definition 4.5.1 *A* threshold system *is a binary monotone system (C, ϕ), where the structure function has the following form:*

$$\phi(\boldsymbol{x}) = I(\sum_{i=1}^{n} a_i x_i \geq b), \tag{4.18}$$

where $a_1, \ldots, a_n$ and b are non-negative real numbers, and $I(\cdot)$ denotes the indicator function, i.e., a function defined for any event A which is 1 if A is true and zero otherwise. The parameters $a_1, \ldots, a_n$ are referred to as the weights *of the system, while b is referred to as the* threshold.

Note that if one of the weights is zero, the corresponding component is irrelevant. Thus, in a coherent threshold system, all the weights must at least be *positive*. However, if some weights are small compared to the other weights, and to the threshold, irrelevant components may still occur.

It should also be noted that the representation (4.18) is *not unique*. For any $c > 0$ we have:

$$I(\sum_{i=1}^{n} a_i x_i \geq b) = I(\sum_{i=1}^{n} c\, a_i x_i \geq c\, b) = I(\sum_{i=1}^{n} a'_i x_i \geq b'),$$

where $a'_i = c\, a_i$, $i = 1, \ldots, n$, and $b' = c\, b$ denote the resulting alternative weights and threshold respectively. If $a_1, \ldots, a_n$ are *rational numbers*, and c is the *least common denominator*, then the alternative weight will all be *integers*. In fact, it can be shown that it is always possible to make small adjustments to the weights without changing the structure. Thus, since the set of rational numbers is dense in $\mathbb{R}$, it is always possible to find a representation of a given threshold system where all the weights are rational. Hence, as explained above, we can also find a representation where all the weights are integers. This also implies that the class of threshold systems with integer weights is equal to the class of all threshold systems.

In cases where all the weights are integers with a *common divisor* D, this divisor can be factored out by choosing $c = D^{-1}$. It turns out that having *integer weights* and at the same time making these weights as *small* as possible can reduce the computational complexity significantly.

We observe that if (C, ϕ) is a threshold system where $a_1 = \cdots = a_n = 1$ and $b = k$, (C, ϕ) is a k-out-of-n system. Thus, the class of threshold systems is a generalisation of the class of k-out-of-n systems. Despite the similarity between threshold systems and k-out-of-n systems, it can be shown that calculating the reliability of a threshold system in general is NP-hard. See Winther [71]. However, by placing mild restrictions on the weights, it is possible to calculate the reliability of the system in polynomial time.

To study this in more detail, we consider a threshold system, (C, ϕ), where $a_1, \ldots, a_n$ are positive integers. As usual, we introduce the vector $\boldsymbol{X} = (X_1, \ldots, X_n)$ of stochastic state variables, and assume that $P(X_i = 1) = p_i$,

and that $q_i = 1 - p_i$, for all $i \in C$. Moreover, we let:

$$S = \sum_{i=1}^{n} a_i X_i,$$

and note that the system is functioning if and only if $S \geq b$. Thus, the reliability of the system can be derived from the distribution of S.

In order to determine this distribution we introduce the partial sums:

$$S_j = \sum_{i=1}^{j} a_i X_i, \quad j = 1, 2, \ldots, n.$$

It is also convenient to introduce:

$$d_j = \sum_{i=1}^{j} a_i, \quad j = 1, 2, \ldots, n.$$

By the assumptions it follows that $S_1, \ldots, S_n$ are integer valued stochastic variables, and that $S_1 = a_1 X_1$ while $S_n = S$. Moreover, denoting the set of possible values for S_j by $\mathcal{A}_j$, $j = 1, \ldots, n$, it follows that:

$$\mathcal{A}_j \subseteq \{0, 1, \ldots, d_j\}, \quad j = 1, \ldots, n. \tag{4.19}$$

More specifically, we have:

$$\begin{aligned} &\mathcal{A}_1 = \{0, a_1\}, \\ &\mathcal{A}_j = \mathcal{A}_{j-1} \cup \{s + a_j : s \in \mathcal{A}_{j-1}\}, \quad j = 2, \ldots, n. \end{aligned} \tag{4.20}$$

Like we did for k-out-of-n systems, we can find the distribution of S by computing sequentially the distributions of $S_1, \ldots, S_n$. Since $S_1 = a_1 X_1$, it follows that $P(S_1 = 0) = q_1$ and that $P(S_1 = a_1) = p_1$. Hence, it follows that the distribution of S_1 is fully determined. We then assume that we have determined the distributions of $S_1, \ldots, S_j$ where $j < n$, and consider the distribution of S_{j+1}. By conditioning on X_{j+1} we get the following recursion formula for $s \in \mathcal{A}_{j+1}$:

$$P(S_{j+1} = s) = P(S_j = s) \cdot q_{j+1} + P(S_j = s - a_{j+1}) \cdot p_{j+1}. \tag{4.21}$$

Note that if $s < a_{j+1}$, we have $P(S_j = s - a_{j+1}) = 0$, while if $s > d_j$, we have $P(S_j = s) = 0$. Thus, in these cases (4.21) simplifies to:

$$P(S_{j+1} = s) = P(S_j = s) \cdot q_{j+1}, \quad s < a_{j+1}, \tag{4.22}$$

$$P(S_{j+1} = s) = P(S_j = s - a_{j+1}) \cdot p_{j+1}, \quad s > d_j. \tag{4.23}$$

Thus, in order to calculate the distribution of S_{j+1}, (4.21) is applied at most $\max(0, d_j - a_{j+1} + 1)$ times, (4.22) is applied at most a_{j+1} times, while (4.23) is applied at most $d_{j+1} - d_j = a_{j+1}$ times. When (4.21) is applied, we need to do two multiplications and one addition, while (4.22) and (4.23) only a single multiplication is required. Hence, in the worst case we need to do $2 \cdot \max(0, d_j - a_{j+1} + 1) + 2 \cdot a_{j+1}$ multiplications and $\max(0, d_j - a_{j+1} + 1)$ additions.

In the case where $a_1 = \cdots = a_n = 1$, we have $d_j = j$, $j = 1, \ldots, n$. Thus, in this case we need to do $2 \cdot \max(0, j - 1 + 1) + 2 \cdot 1 = 2(j + 1)$ multiplications and $\max(0, j - 1 + 1) = j$ additions. Obviously, these are the same numbers of operations as we found in Section 4.4. Unfortunately, in other cases the number of operations may grow much faster.

Example 4.5.2 *Let (C, ϕ) be a threshold system with weights $a_1, \ldots, a_n$ given by:*

$$a_j = 2^{j-1}, \quad j = 1, \ldots, n.$$

We then have:

$$\begin{aligned}
&\mathcal{A}_1 = \{0, 2^0\} = \{0, 1\}, \\
&\mathcal{A}_2 = \mathcal{A}_1 \cup \{0 + 2^1, 2^0 + 2^1\} = \{0, 1, 2, 3\} \\
&\quad \cdots \\
&\mathcal{A}_j = \{0, 1, \ldots, (2^j - 1)\} \\
&\quad \cdots \\
&\mathcal{A}_n = \{0, 1, \ldots, (2^n - 1)\}
\end{aligned}$$

Hence, $|\mathcal{A}_j| = 2^j$, $j = 1, \ldots, n$, which means that the size of the $\mathcal{A}_j$ grows exponentially fast. In this case any algorithm for calculating the distribution of $S = S_n$ must be at least of order $O(2^n)$.

We note that the number of possible values for S is equal to the number of binary vectors $\boldsymbol{x} = (x_1, \ldots, x_n)$, i.e., 2^n. In fact each possible value s of S corresponds to a unique vector $\boldsymbol{x}(s) = (x_1(s), \ldots, x_n(s))$ where the entries represents the digits in the binary representation of the number s. That is:

$$s = \sum_{i=1}^{n} x_i(s) 2^{i-1}$$

Hence, it follows that the distribution of S can be determined by:

$$P(S = s) = \prod_{i=1}^{n} P(X_i = x_i(s)) = \prod_{i=1}^{n} p_i^{x_i(s)} q_i^{1-x_i(s)}, \quad s = 0, 1, \ldots, 2^n.$$

However, calculating the full distribution of S using this simple explicit formula still requires the calculation of 2^n such products.

It should be noted, however, that for a given threshold value b, it is possible to compute the reliability of the system without calculating the distribution of S. Assume e.g., that $b = 2^{k-1}$ for some integer $k \in \{1, \ldots, n\}$. In this case all sets of the form $P = \{j\}$, where $j \geq k$ will be minimal path sets for the system, while all components in the set $\{1, \ldots, k-1\}$ are irrelevant. Thus, the system is essentially a simple parallel system of the components $k, \ldots, n$, and the reliability of the system is given by:

$$P(\phi = 1) = \coprod_{j=k}^{n} p_j$$

By Example 4.5.2 we see that when the reliability of a threshold system is calculated using the distribution of S, the running time of the algorithm may in the worst cases grows exponentially in the number of components. In the next example, however, we show that by putting a mild restriction on the weights, it becomes possible to calculate the reliability of a threshold system in polynomial time.

Example 4.5.3 *Let (C, ϕ) be a threshold system with positive integer weights $a_1, \ldots, a_n$, and assume that:*

$$a_j \leq M, \quad j = 1, \ldots, n,$$

for some suitable fixed integer $M > 0$. In this case we have:

$$d_j = \sum_{i=1}^{j} a_i \leq Mj$$

Hence, since a_{j+1} is assumed to be a positive integer, we also have that:

$$d_j - a_{j+1} + 1 \leq Mj$$

Thus, if we use the recursive formulas to calculate the distribution of S_{j+1} from the distribution of S_j, an upper bound on the number of multiplications needed is:

$$2\max(0, d_j - a_{j+1} + 1) + 2a_{j+1} \leq 2Mj + 2M = 2M(j+1)$$

Similarly, an upper bound on the number of addition needed is:

$$\max(0, d_j - a_{j+1} + 1) \leq Mj$$

An upper bound on the total number of multiplications needed to calculate the distribution of S is then:

$$2 \cdot 2M + 2 \cdot 3M + \cdots 2 \cdot nM = M(n-1)(n+2),$$

while an upper bound on the number of additions needed is:

$$M + 2M + \cdots + (n-1)M = \frac{M(n-1)n}{2}$$

Since M is a fixed constant, it follows that the distribution of S, and hence also the reliability of the system for any threshold b, can be calculated in $O(n^2)$-time.

Example 4.5.4 *Let (C, ϕ) be a threshold system of 8 components with the following reliabilities:*

$$p_1 = 0.75, \quad p_2 = 0.80, \quad p_3 = 0.82, \quad p_4 = 0.65$$
$$p_5 = 0.88, \quad p_6 = 0.91, \quad p_7 = 0.92, \quad p_8 = 0.86$$

The component weights are:

$$a_1 = 2, \quad a_2 = 3, \quad a_3 = 5, \quad a_4 = 6$$
$$a_5 = 7, \quad a_6 = 8, \quad a_7 = 8, \quad a_8 = 11$$

and the threshold is $b = 32$.

We also let $S = a_1X_1 + \cdots + a_8X_8$, *and note that since* $a_1 + \cdots + a_8 = 50$, *the set of possible values for* S *is a subset of the set* $\{0, 1, 2, \ldots, 50\}$.

Using (4.21), (4.22) *and* (4.23) *we find the distribution of* S, *and get the reliability of the system:*

$$\begin{aligned} h &= P(\phi = 1) = P(S \geq b) \\ &= P(S = 32) + P(S = 33) + \cdots + P(S = 50) = 0.9238 \end{aligned}$$

A python script which can be used to calculate the reliability of this system, is given in Script B.1.2 in Appendix B.1.

4.6 Directed network systems

For some particular kinds of systems, the multiplication method and the inclusion-exclusion method can be significantly improved. The most important class of such systems are Source-to-K terminal systems, or *SKT-systems*:

Definition 4.6.1 *A Source-to-*K*-terminal-system (SKT-system) is a system defined relative to a directed network where the system functions if and only if a node* S *(called the source) can send information to a given set of* K *nodes* $T_1, \ldots, T_K$ *(called the terminals). The components of the system are the directed edges of the network, while the nodes are assumed to be functioning perfectly with probability one.*

Example 4.6.2 *Figure 4.1 shows an example of an S4T system with components* $1, 2, \ldots, 10$. *The node* S *is the source, while the nodes* T_1, T_2, T_3, T_4 *are the terminals.*

Note that the class of SKT-systems is closed under restrictions but *not under contractions*. Let e.g., (C, ϕ) be the system presented in Example 4.6.2. Then it can be shown that the contractions of (C, ϕ) with respect to components 3, 6 and 7 are *not* SKT-systems. For this reason pivotal decompositions are usually not applied for reliability calculations of such systems. Instead, we focus on improvements of the multiplication method. In the following theorem we recall that any structure function can be written on the form (4.3).

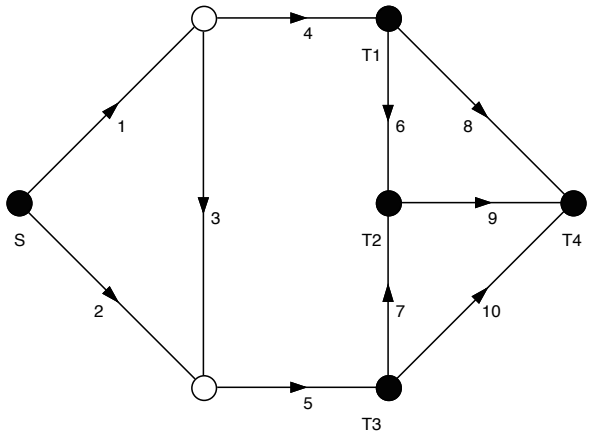

Figure 4.1: An S4T system

Theorem 4.6.3 *Let* $\phi(\boldsymbol{X}) = \sum_{A \subseteq C} \delta(A) \prod_{i \in A} X_i$ *be the structure function of an SKT system.*

(i) If A *can be expressed as a union of minimal path sets, and the subgraph spanned by* A *does not contain any directed cycle, we have:*

$$\delta(A) = (-1)^{|A|-v(A)+1}$$

where $v(A)$ *denotes the number of nodes in the subgraph spanned by* A*.*

(ii) In the opposite case we have:

$$\delta(A) = 0$$

Theorem 4.6.3 was first proved by Satyanarayana [63] using Theorem 4.3.2. However, a simpler proof based on the formula (4.5) is given in Huseby [32].

Prabhakar and Satyanarayana [65] also developed an efficient algorithm for identifying all sets $A \subseteq C$ such that $\delta(A) \neq 0$ directly, without using the minimal path sets. By using this algorithm in combination with Theorem 4.6.3, we get a very fast method for computing the reliability of an SKT system.

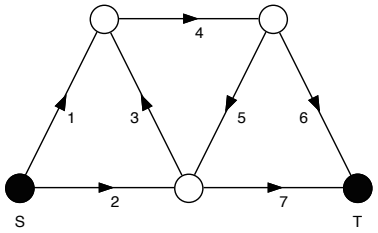

Figure 4.2: An S1T system

Example 4.6.4 *Consider the S1T-system shown in Figure 4.2. The component set* $C = \{1, \ldots, 7\}$ *consists of the directed edges in the networks. The system is functioning if the source* S *can send signals to the terminal* T *through the network. We assume that the component state variables are independent and that* $P(X_i = 1) = p_i$ *for* $i \in C$.

The minimal path sets of this system are $P_1 = \{1, 4, 6\}$, $P_2 = \{1, 4, 5, 7\}$, $P_3 = \{2, 3, 4, 6\}$ *and* $P_4 = \{2, 7\}$. *We calculate the reliability of this system by using the inclusion-exclusion formula* (4.7). *Since there are* 4 *minimal path sets, this formula will consist of* $2^4 - 1 = 15$ *terms before we simplify:*

$$\begin{aligned} h(\boldsymbol{p}) = {} & p_1p_4p_6 + p_1p_4p_5p_7 + p_2p_3p_4p_6 + p_2p_7 \\ & - p_1p_4p_5p_6p_7 - p_1p_2p_3p_4p_6 - p_1p_2p_4p_6p_7 \\ & - p_1p_2p_3p_4p_5p_6p_7 - p_1p_2p_4p_5p_7 - p_2p_3p_4p_6p_7 \\ & + p_1p_2p_3p_4p_5p_6p_7 + p_1p_2p_4p_5p_6p_7 + p_1p_2p_3p_4p_6p_7 + p_1p_2p_3p_4p_5p_6p_7 \\ & - p_1p_2p_3p_4p_5p_6p_7. \end{aligned}$$

By merging similar terms we obtain:

$$\begin{aligned} h(\boldsymbol{p}) = {} & p_1p_4p_6 + p_1p_4p_5p_7 + p_2p_3p_4p_6 + p_2p_7 \\ & - p_1p_4p_5p_6p_7 - p_1p_2p_3p_4p_6 - p_1p_2p_4p_6p_7 \\ & - p_1p_2p_4p_5p_7 - p_1p_2p_4p_5p_7 \\ & + p_1p_2p_4p_5p_6p_7 + p_1p_2p_3p_4p_6p_7. \end{aligned}$$

We observe that $\delta(A)$ *is either* $+1$, -1 *or zero for all* $A \subseteq C$. *In particular, since the network contains a directed cycle* $\{3, 4, 5\}$, *the highest order term will have* $\delta(C) = 0$. *Moreover, we can easily verify that the formula for* $\delta(A)$

given in Theorem 4.6.3 is correct:

$$\begin{aligned}
\delta(\{1,4,6\}) &= (-1)^{3-4+1} = +1,\\
\delta(\{1,4,5,7\}) &= (-1)^{4-5+1} = +1,\\
&\cdots\cdots\\
\delta(\{1,4,5,6,7\}) &= (-1)^{5-5+1} = -1,\\
\delta(\{1,2,3,4,6\}) &= (-1)^{5-5+1} = -1,\\
\delta(\{1,2,4,6,7\}) &= (-1)^{5-5+1} = -1,\\
&\cdots\cdots\\
\delta(\{1,2,4,5,6,7\}) &= (-1)^{6-5+1} = +1,\\
\delta(\{1,2,3,4,6,7\}) &= (-1)^{6-5+1} = +1.
\end{aligned}$$

We close this section by noting that SKT-systems are not the only types of systems where the signed domination is either $+1$, -1 or zero. In fact this class can be extended to a much larger class called *oriented matroid systems*. See Huseby [35] for details. Moreover, there are also other classes of systems with similar properties:

Example 4.6.5 *Let (C,ϕ) be a linear consecutive 2-out-of-5 system. That is, $C=\{1,\ldots,5\}$, and the minimal path sets are $P_1=\{1,2\}$, $P_2=\{2,3\}$, $P_3=\{3,4\}$, $P_4=\{4,5\}$. By using e.g., the inclusion-exclusion formula it is easy to see that the reliability of this system, assuming independent component state variables, is:*

$$\begin{aligned}
h(\boldsymbol{p}) &= p_1p_2 + p_2p_3 + p_3p_4 + p_4p_5\\
&\quad - p_1p_2p_3 - p_2p_3p_4 - p_3p_4p_5 - p_1p_2p_4p_5\\
&\quad + p_1p_2p_3p_4p_5.
\end{aligned}$$

It can be shown that this system is not an SKT-system. The proof is left as an exercise. Still, linear consecutive k-out-of-n systems share the property with SKT-systems (and the more general class of oriented matroid systems) that the signed domination function is either $+1$, -1 or zero for all subsets of the component set. See Calkin et. al [14].

4.7 Undirected network systems

In this section, we will present an algorithm for computing the exact system reliability called *the factoring algorithm* which is based on pivotal decom-

position. See Theorem 3.1.1. In principle, this algorithm can be applied to any binary monotone system where the component state variables are independent. However, since the algorithm uses pivotal decompositions, it is typically used for classes of systems which are *closed under minor operations*. An important such class is the class of *undirected network systems*. Such a system can be represented by an *undirected graph*, where the components of the system are the *edges* of the graph, and where the system is functioning if a set of terminal nodes can communicate through the graph. In an undirected network signals can be sent both ways along the edges. An example of such a system is illustrated in Figure 4.4. In addition to pivotal decomposition the method also uses the following result:

Theorem 4.7.1 *Let $i, j \in C$, $i \neq j$.*

(i) If i and j are connected in series, then $h(\boldsymbol{p})$ will only depend on p_i and p_j through $p_i \cdot p_j$. Hence, i and j can be replaced by a single component with reliability $p_i \cdot p_j$ without altering the system reliability. Such a reduction is called a series reduction.

(ii) If i and j are connected in parallel, then $h(\boldsymbol{p})$ will only depend on p_i and p_j through $p_i \amalg p_j$. Hence, i and j can be replaced by a single component with reliability $p_i \amalg p_j$ without altering the system reliability. Such a reduction is called a parallel reduction.

The common term for either series or parallel reductions is *s-p-reductions*. Note that since the class of undirected network systems is closed under minor operations, it is also closed under s-p-reductions. The reason for this is that a series reduction is graph-theoretically equivalent to a contraction, while a parallel reduction is graph-theoretically equivalent to a restriction.

Definition 4.7.2 *We say that a system is* s-p-reducible *if there are components in either series or parallel in the system. If not, the system is said to be* s-p-complex. *A system which can be s-p-reduced to a single component is called an* s-p-system.

The bridge structure shown in Figure 3.1 is an example of a complex system because there are no components in series or parallel.

The system in Figure 4.3 is an example of an s-p system because components 1 and 2 (in series) can be replaced with a component $1'$ with reliability

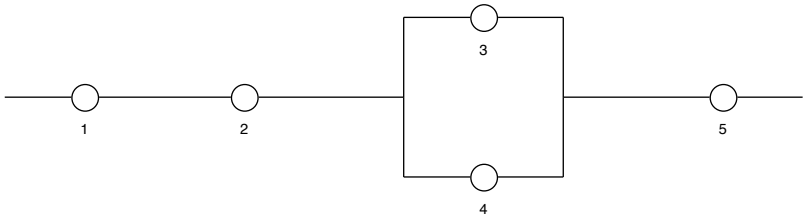

Figure 4.3: A reliability block diagram of a mixed parallel and series system.

$p_{1'} = p_1 \cdot p_2$. Similarly, components 3 and 4 can be replaced by a component $3'$ with reliability $p_{3'} = p_3 \amalg p_4$. The resulting system is a series structure of components $1', 3'$ and 5. These three components can be replaced by a single component with reliability $p_{1'} \cdot p_{3'} \cdot p_5$, which is the reliability of the whole system.

The factoring algorithm can be described as follows:

Algorithm 4.7.3 *Let (C, ϕ) be a binary monotone system. Assume that at least one of its components is relevant. To compute the reliability $h(\boldsymbol{p})$, proceed as follows:*

Step 1. *Perform all possible s-p-reductions. Let the reduced system be denoted by (C^r, ϕ^r). Then, (C^r, ϕ^r) must also have at least one relevant component (make sure you understand why).*

Step 2. *Now, one of the two following cases can happen:*

CASE 1. *(C^r, ϕ^r) contains precisely one relevant component with updated reliability p_e. Then, $h(\boldsymbol{p}) = p_e$.*

CASE 2. *(C^r, ϕ^r) contains several relevant components. In this case, choose a component $e \in C^r$ and do a pivotal decomposition. That is, compute $h(\boldsymbol{p})$ by using Theorem 3.1.1:*

$$h(\boldsymbol{p}^{C^r}) = p_e h(1_e, \boldsymbol{p}^{C^r}) + (1 - p_e) h(0_e, \boldsymbol{p}^{C^r}).$$

Then, compute $h(1_e, \boldsymbol{p}^{C^r})$ and $h(0_e, \boldsymbol{p}^{C^r})$ by repeated use of the algorithm.

Note that there is a choice involved in Case 2 of Algorithm 4.7.3. In general, the efficiency of the factoring algorithm depends on the choice of

pivoting component. For the bridge structure in Example 3.1.2, we actually used the factoring algorithm. There, we chose to pivot with respect to component 3. For this particular system the choice does not matter. In general, however, it can be shown that when the factoring algorithm is applied to undirected network systems, one should always pivot such that both resulting minors are coherent[1]. This was first proved by Satyanarayana and Chang [64], and later generalised to a more abstract class by Huseby [32].

Despite being a very easy and convenient method for reliability calculations, the factoring algorithm is still of order $O(2^n)$. Hence, in that sense it is not significantly better than e.g., the state space enumeration method. Still, in many cases, it is very efficient, even for manual calculations.

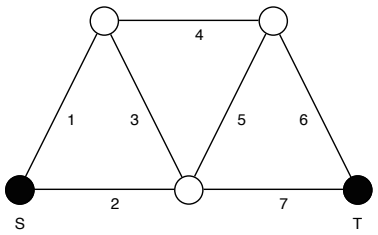

Figure 4.4: An undirected network system

Example 4.7.4 *Consider the undirected network system (C, ϕ) shown in Figure 4.4. The component set $C = \{1, \ldots, 7\}$ consists of the undirected edges in the networks. The system is functioning if the terminals S and T can communicate through the network. We assume that the component state variables are independent and that $P(X_i = 1) = p_i$ for $i \in C$.*

This system is not *s-p-reducible, so we need to do a pivotal decomposition. For this purpose we choose component* 4*. The resulting subsystems are shown in Figure 4.5, where the left-hand system, denoted $(C \setminus 4, \phi_{+4})$ and with reliability h_{+4}, is obtained by conditioning on that component* 4 *is functioning, while the right-hand system, denoted $(C \setminus 4, \phi_{-4})$ and with reliability h_{-4}, is*

[1]Note that the assumption that the system is an undirected network system is essential here. While the result can be generalised to a larger class of systems, see Huseby [32], there exists many other types of systems where this rule is *not* optimal.

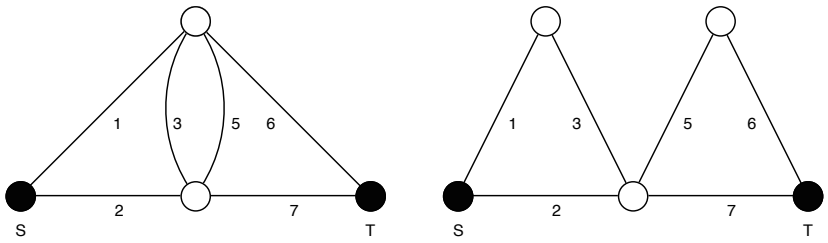

Figure 4.5: Subsystems $(C \setminus 4, \phi_{+4})$ and $(C \setminus 4, \phi_{-4})$ obtained by pivotal decomposition with respect to component 4

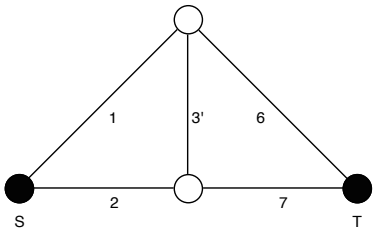

Figure 4.6: The subsystem $((C \setminus 4)^r, (\phi_{+4})^r)$ obtained by a parallel reduction of $(C \setminus 4, \phi_{+4})$ with respect to components 3 and 5

obtained by conditioning on that component 4 *is failed. The system reliability is then given by:*

$$h = p_4 \cdot h_{+4} + (1 - p_4) \cdot h_{-4}.$$

We observe that $(C \setminus 4, \phi_{-4})$ *is s-p-reducible with reliability* h_{-4} *given by:*

$$h_{-4} = [(p_1 \cdot p_3) \amalg p_2] \cdot [(p_5 \cdot p_6) \amalg p_7].$$

The subsystem $(C \setminus 4, \phi_{+4})$ *is s-p-reducible as well, but the only possible s-p-reduction is the parallel reduction of components* 3 *and* 5*. The resulting component is denoted* $3'$ *and has component reliability* $p_{3'} = p_3 \amalg p_5$*. The resulting system is shown in Figure 4.6, and is denoted by* $((C \setminus 4)^r, (\phi_{+4})^r)$*.*

In order to calculate the reliability of this system, we need to do another pivotal decomposition. This time we choose component $3'$*. The resulting subsystems, denoted respectively* $((C \setminus 4)^r \setminus 3', (\phi_{+4})^r_{+3'})$ *and* $((C \setminus 4)^r \setminus 3', (\phi_{+4})^r_{-3'})$

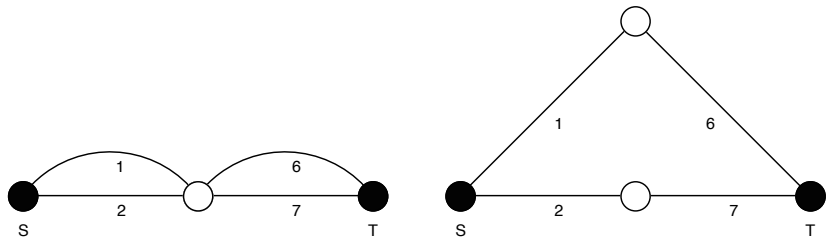

Figure 4.7: Subsystems $((C \setminus 4)^r \setminus 3', (\phi_{+4})^r_{+3'})$ and $((C \setminus 4)^r \setminus 3', (\phi_{+4})^r_{-3'})$ obtained from the subsystem $((C \setminus 4)^r, (\phi_{+4})^r)$ by a pivotal decomposition with respect to component $3'$

are shown in Figure 4.7. Both these subsystems are s-p-reducible, and we get that:

$$h_{+4} = (h_{+4})^r = p_{3'} \cdot (h_{+4})^r_{+3'} + (1 - p_{3'}) \cdot (h_{+4})^r_{-3'},$$

where:

$$(h_{+4})^r_{+3'} = (p_1 \amalg p_2) \cdot (p_6 \amalg p_7)$$
$$(h_{+4})^r_{-3'} = (p_1 \cdot p_6) \amalg (p_2 \cdot p_7)$$

This completes the calculations.

Note that it is obviously possible to combine all the various expressions in Example 4.7.4 into one unified large formula for the reliability h. However, when implementing the factoring algorithm, this is done recursively. This implies that only the resulting numbers, not the expanded symbolic expressions, are passed backwards through the algorithm. This is actually an essential point as this saves both time and space.

4.8 Binary decision diagrams

Binary decision diagrams or BDDs have many different applications in computer science, and dates back to the papers [45] and [1]. More efficient implementations were introduced in [12], [11] and [13]. Applications of BDDs in reliability theory were introduced in [20] and [61] and adapted to network reliability in [9]. For a survey of these and other related methods see [62].

A binary decision diagram can be interpreted as a representation of a binary function $\phi : \{0,1\}^n \to \{0,1\}$ in the form of a rooted directed acyclic graph. In our context the function ϕ is typically the structure function of some binary monotone system. This means that we only consider the special case where the binary function is non-decreasing in each argument. The *root* and the *intermediate* nodes are drawn as circular nodes, and labelled with indices of the binary input variables. The edges represent *decisions* regarding the values of the input variables, i.e., whether the value of an input variable is fixed to be either 0 or 1. The square *leaf* nodes represents cases where the associated binary function is trivial, and the labels in this case represent the corresponding binary value, i.e., either 0 or 1.

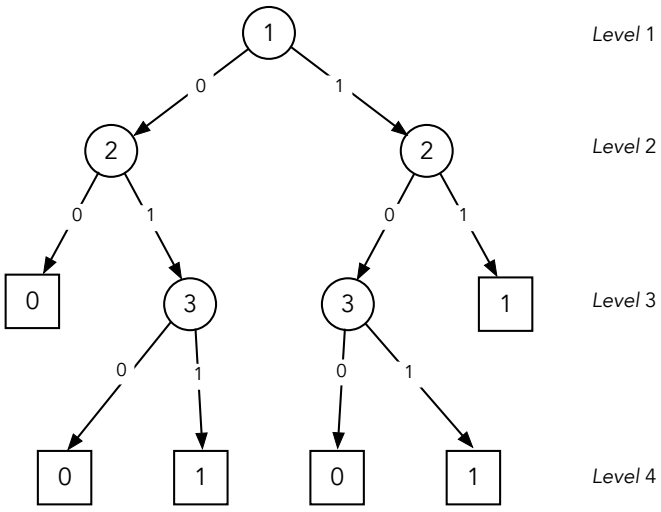

Figure 4.8: An ordered binary decision diagram of a 2-out-of-3 system.

Figure 4.8 shows a binary decision diagram of a 2-out-of-3 system. There are three binary input variables, x_1, x_2, x_3. The binary function ϕ, represents the structure function of the system, and is given by:

$$\phi(\boldsymbol{x}) = x_1x_2 + x_1x_3 + x_2x_3 - 2x_1x_2x_3$$

To each node in the diagram we associate a binary function defined as a function of the remaining binary variables whose values are not yet fixed. In particular, the root node in the diagram, labelled 1, is associated with the

function ϕ itself since at this stage none of the variables are fixed. The lower level nodes are associated with structure functions of minors of the system under consideration. Since each input variable has two possible values, 0 and 1, each node has exactly two *children*, i.e., nodes connected by edges to the *parent* node, where as a convention the lefthand child represents the binary function given that the respective input variable is fixed to be 0, while the righthand child represents the binary function given that the respective input variable is 1. In Figure 4.8 there are two nodes labelled 2 which are children of the root node 1. To the leftmost node we associate the function:

$$\phi(0_1, \boldsymbol{x}) = 0 \cdot x_2 + 0 \cdot x_3 + x_2x_3 - 2 \cdot 0 \cdot x_2x_3 = x_2x_3,$$

while the rightmost node we associate the function:

$$\phi(1_1, \boldsymbol{x}) = 1 \cdot x_2 + 1 \cdot x_3 + x_2x_3 - 2 \cdot 1 \cdot x_2x_3 = x_2 + x_3 - x_2x_3$$

At the next level of the diagram there are two intermediate nodes and two leaf nodes. From left to right these nodes correspond to the following binary functions:

$$\begin{aligned}
&\phi(0_1, 0_2, \boldsymbol{x}) = 0 \cdot x_3 = 0,\\
&\phi(0_1, 1_2, \boldsymbol{x}) = 1 \cdot x_3 = x_3,\\
&\phi(1_1, 0_2, \boldsymbol{x}) = 0 + x_3 - 0 \cdot x_3 = x_3,\\
&\phi(1_1, 1_2, \boldsymbol{x}) = 1 + x_3 - 1 \cdot x_3 = 1.
\end{aligned}$$

We observe the binary functions associated with the two leaf nodes are trivial with values 0 and 1 respectively. The binary functions associated with the two intermediate nodes depends on the remaining input variable 3. Thus, for these nodes we proceed to the final level consisting of four leaf nodes. From left to right these nodes correspond to the following binary functions:

$$\begin{aligned}
&\phi(0_1, 1_2, 0_3) = 0,\\
&\phi(0_1, 1_2, 1_3) = 1,\\
&\phi(1_1, 0_2, 0_3) = 0,\\
&\phi(1_1, 0_2, 1_3) = 1
\end{aligned}$$

Note that since the binary functions associated with the two intermediate nodes labelled 3 are identical, it follows that the subgraphs rooted at these

nodes are *isomorphic*[2].

We now consider the states of the input variables as independent stochastic variables, denoted X_1, X_2, X_3, and let $P(X_i = 1) = p_i$, $i = 1, 2, 3$. We also let $P(X_i = 0) = 1 - p_i = q_i$, $i = 1, 2, 3$. Then each node in the diagram corresponds to an *event* defined by these stochastic variables. Numbering these events by the diagram level and from left to right, we denote the events as follows:

$$E_{ij} = \text{The event corresponding to the } j\text{th node at level } i.$$

The event corresponding to the root node, i.e., $E_{1,1}$, obviously has probability $P(E_{1,1}) = 1$. At each of the lower levels, we can compute the event probabilities as products of parent event probabilities and probabilities related to the input variables. More specifically, let v be a node in the diagram, and let E is the corresponding event. Moreover, let E' be the event corresponding to the parent node of v. Assuming that E' is an event at level i, we have:

$$P(E) = P(E') \cdot P(E' \to E) \tag{4.24}$$

where $P(E' \to E) = q_i$ if v is a lefthand child of the parent node corresponding to E', and $P(E' \to E) = p_i$ if v is a righthand child of the parent node corresponding to E'.

We then apply (4.24) to compute the event probabilities. At level 2 in the diagram there are two intermediate nodes with corresponding events $E_{2,1}$ and $E_{2,2}$. The probabilities of these events are:

$$P(E_{2,1}) = P(E_{1,1}) \cdot P(E_{1,1} \to E_{2,1}) = q_1$$
$$P(E_{2,2}) = P(E_{1,1}) \cdot P(E_{1,1} \to E_{2,2}) = p_1$$

At level 3 there are two leaf nodes with corresponding events $E_{3,1}$ and $E_{3,4}$. The probabilities of these events are:

$$P(E_{3,1}) = P(E_{2,1}) \cdot P(E_{2,1} \to E_{3,1}) = q_1 q_2$$
$$P(E_{3,4}) = P(E_{2,2}) \cdot P(E_{2,2} \to E_{3,4}) = p_1 p_2$$

[2]Two directed graphs G and H with node sets $V(G)$ and $V(H)$ respectively are said to be *isomorphic* if there exists a bijective mapping, $\psi : V(G) \to V(H)$, such that G contains a directed edge from node u to node v if and only H contains a directed edge from node $\psi(u)$ to $\psi(v)$.

Furthermore, at level 3 there are also two intermediate nodes with corresponding events $E_{3,2}$ and $E_{3,3}$. The probabilities of these events are:

$$P(E_{3,2}) = P(E_{2,1}) \cdot P(E_{2,1} \to E_{3,2}) = q_1 p_2$$
$$P(E_{3,3}) = P(E_{2,2}) \cdot P(E_{2,2} \to E_{3,3}) = p_1 q_2$$

Finally, at level 4 of the diagram there are four leaf nodes with corresponding events $E_{4,1}, E_{4,2}, E_{4,3}, E_{4,4}$. The probabilities of these events are:

$$P(E_{4,1}) = P(E_{3,2}) \cdot P(E_{3,2} \to E_{4,1}) = q_1 p_2 q_3$$
$$P(E_{4,2}) = P(E_{3,2}) \cdot P(E_{3,2} \to E_{4,2}) = q_1 p_2 p_3$$
$$P(E_{4,3}) = P(E_{3,3}) \cdot P(E_{3,3} \to E_{4,3}) = p_1 q_2 q_3$$
$$P(E_{4,4}) = P(E_{3,3}) \cdot P(E_{3,3} \to E_{4,4}) = p_1 q_2 p_3$$

Given the probabilities of the leaf nodes, we find the probability distribution of $\phi(\boldsymbol{X})$ by adding the probabilities associated with the leaf nodes labelled 0 and 1 respectively:

$$\begin{aligned} P(\phi(\boldsymbol{X}) = 0) &= P(E_{3,1}) + P(E_{4,1}) + P(E_{4,3}) \\ &= q_1 q_2 + q_1 p_2 q_3 + p_1 q_2 q_3 \\ &= q_1 q_2 + q_1 q_3 + q_2 q_3 - 2 q_1 q_2 q_3 \end{aligned}$$

$$\begin{aligned} P(\phi(\boldsymbol{X}) = 1) &= P(E_{3,4}) + P(E_{4,2}) + P(E_{4,4}) \\ &= p_1 p_2 + q_1 p_2 p_3 + p_1 q_2 p_3 \\ &= p_1 p_2 + p_1 p_3 + p_2 p_3 - 2 p_1 p_2 p_3 \end{aligned}$$

In the diagram shown in Figure 4.8 there is a one-to-one correspondence between the input variables and the levels of the diagram. That is, input variable 1 is handled at level 1, input variable 2 is handled at level 2, etc. Such diagrams are referred to as *ordered binary decision diagrams*. When the diagram has a rooted tree structure, like the one shown in Figure 4.8, however, we do not need to handle the input variables in an ordered way. In Figure 4.9 the input variables 2 and 3 are handled in a different order in the lefthand and righthand parts of the diagram. Thus, we observe that the two nodes at level 2 do *not* refer to the same input variable. However, the probabilities of the events corresponding to these nodes have probabilities determined by the distribution of X_1, so we still have:

$$P(E_{2,1}) = P(E_{1,1}) \cdot P(E_{1,1} \to E_{2,1}) = q_1$$
$$P(E_{2,2}) = P(E_{1,1}) \cdot P(E_{1,1} \to E_{2,2}) = p_1$$

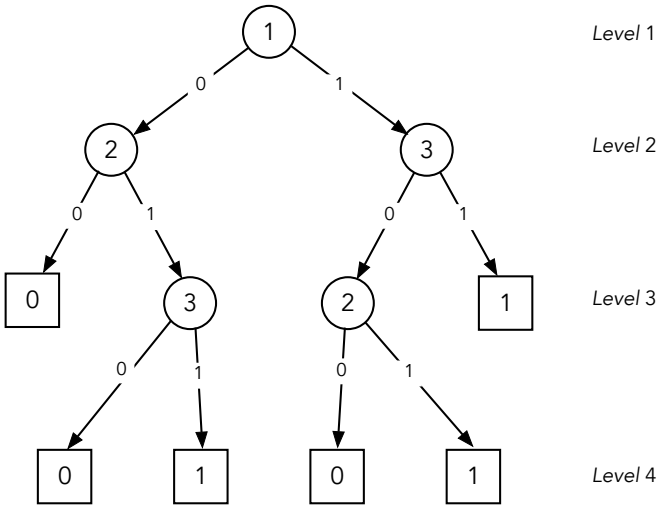

Figure 4.9: An unordered binary decision diagram of a 2-out-of-3 system.

At level 3 of the diagram the leaf nodes with corresponding events $E_{3,1}$ and $E_{3,4}$ have the following probabilities:

$$P(E_{3,1}) = P(E_{2,1})P(E_{2,1} \to E_{3,1}) = q_1 q_2$$
$$P(E_{3,4}) = P(E_{2,2})P(E_{2,2} \to E_{3,4}) = p_1 p_3$$

while the intermediate nodes with corresponding events $E_{3,2}$ and $E_{3,3}$ have the following probabilities:

$$P(E_{3,2}) = P(E_{2,1}) \cdot P(E_{2,1} \to E_{3,2}) = q_1 p_2$$
$$P(E_{3,3}) = P(E_{2,2}) \cdot P(E_{2,2} \to E_{3,3}) = p_1 q_3$$

Finally, at level 4 of the diagram the leaf nodes have the following probabilities:

$$P(E_{4,1}) = P(E_{3,2}) \cdot P(E_{3,2} \to E_{4,1}) = q_1 p_2 q_3$$
$$P(E_{4,2}) = P(E_{3,2}) \cdot P(E_{3,2} \to E_{4,2}) = q_1 p_2 p_3$$
$$P(E_{4,3}) = P(E_{3,3}) \cdot P(E_{3,3} \to E_{4,3}) = p_1 q_3 q_2$$
$$P(E_{4,4}) = P(E_{3,3}) \cdot P(E_{3,3} \to E_{4,4}) = p_1 q_3 p_2$$

Still, when we add the probabilities of the leaf nodes, we eventually arrive at the same probability distribution of $\phi(\boldsymbol{X})$.

$$\begin{aligned} P(\phi(\boldsymbol{X}) = 0) &= P(E_{3,1}) + P(E_{4,1}) + P(E_{4,3}) \\ &= q_1 q_2 + q_1 p_2 q_3 + p_1 q_3 q_2 \\ &= q_1 q_2 + q_1 q_3 + q_2 q_3 - 2 q_1 q_2 q_3 \\ P(\phi(\boldsymbol{X}) = 1) &= P(E_{3,4}) + P(E_{4,2}) + P(E_{4,4}) \\ &= p_1 p_3 + q_1 p_2 p_3 + p_1 q_3 p_2 \\ &= p_1 p_2 + p_1 p_3 + p_2 p_3 - 2 p_1 p_2 p_3 \end{aligned}$$

Note that in the above examples we have expanded the formulas for the distribution of $\phi(\boldsymbol{X})$ into a sum of products of probabilities. This is done in order to make it easy to verify that the resulting formulas are indeed correct. Since the binary function under consideration here is very simple, even the fully expanded formula is easily manageable. In more complex cases, however, the calculations will be done based on specific numbers for the distributions of the input variables. Then the resulting event probabilities will be computed event by event, and level by level in the diagram. In each calculation (4.24) is applied. Since this formula can be calculated in constant time, this means that the computational complexity of computing all event probabilities, and thus also the distribution of $\phi(\boldsymbol{X})$, is proportional to the number of nodes in the diagram.

Note also that when a binary decision diagram is constructed, this diagram represents the binary function of interest, regardless of the distributions of the input variables. Thus, we can use the same diagram to compute the distribution of $\phi(\boldsymbol{X})$ for any number of distributions of the input variables. This allows us to e.g., use the diagram for various types of sensitivity analysis.

In this particular example the two diagrams shown in Figure 4.8 and Figure 4.9 both have the same size. Thus, there is nothing to gain by using different variable orderings. In general, however, allowing different variable orderings in different parts of the diagram may have a significant effect on the size of the diagram, and hence also on the computational complexity.

If we e.g., calculate system reliability by representing the structure function as a binary decision diagram with a rooted tree structure like the one described above, the computational complexity of this method is essentially

equivalent to the computational complexity of the *factoring algorithm* without s-p-reductions. If the system is an undirected network system, we know that the optimal factoring strategy is to avoid incoherent minors. See Subsection 4.7. This strategy may sometimes require that different variable orderings are used in different parts of the diagram.

One simple way of reducing the size of a binary decision diagrams is merging all the leaf nodes into two nodes corresponding to the values 0 and 1 respectively. The following BDD-construction algorithm takes this into account.

Algorithm 4.8.1 *Let $\phi = \phi(\boldsymbol{x})$ be a binary function, and let $C = \{1, \ldots, n\}$ denote the indices of the input variables.*

STEP 1. *At level* 1 *create the root node with the function $\phi(\boldsymbol{x})$, and create two leaf nodes with functions* 0 *and* 1 *respectively.*

- *Select $i \in C$, and label the root node by this i.*

...

STEP k. *At level k create the lefthand and righthand children of each of the nodes at level $(k-1)$ with functions obtained by fixing the value of their respective input variables (indicated by their labels) to* 0 *and* 1 *respectively.*

- *Merge nodes with trivial functions with the relevant leaf nodes (if any).*
- *For each intermediate node select $i \in C$ among the variables that have not yet been fixed in this node's function, and label the node by this i.*

...

Algorithm 4.8.1 is implemented in python in Script B.2.1 (c_bdd.py) in Appendix B.2. Note that this is a *generic* implementation which can be used on different types of systems. However, in order to apply this script to a given system, some of the generic methods must be tailored to fit the system under consideration. In particular, the following methods must be implemented in a system specific way:

- A method for representing the binary functions obtained as a result of fixing the values of the input variables (*restriction* and *contraction*)
- A method for determining if a binary function is trivial.

- A method for selecting input variables for each node in the BDD

The different implementations of these methods depend strongly on the class of systems under consideration, and on the chosen representation of the binary functions.

In [12] a *reduced* form of a binary decision diagram was introduced. See also [11]. Such reduced diagrams are referred to as *reduced ordered binary decision diagrams* or ROBDDs. Such a diagram is obtained from an ordered binary decision diagram by merging nodes which correspond to equal binary functions. In particular, such diagrams only have two leaf nodes, one labelled 0 and one labelled 1. For these reductions to work, however, only ordered decision diagram are allowed. The reason for this is that if we use unordered diagrams, nodes at the same level will correspond to binary functions with different input variables. Thus, it is typically not possible to find nodes which correspond to equal binary functions.

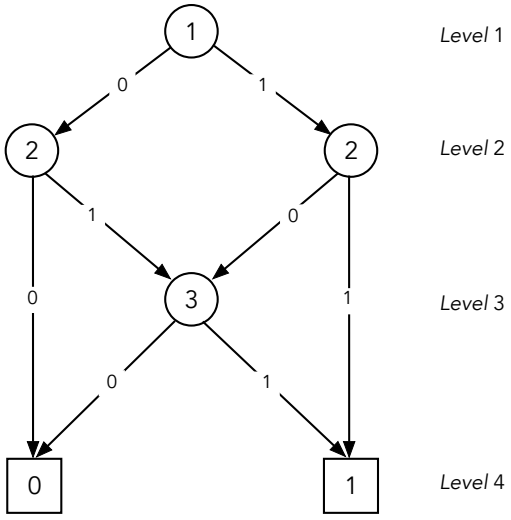

Figure 4.10: A reduced ordered binary decision diagram of a 2-out-of-3 system.

Figure 4.10 shows a reduced ordered binary decision diagram for a 2-out-of-3 system. This is obtained from the diagram shown in Figure 4.8 by merging the leaf nodes labelled 0 into one leaf node and by merging the

leaf nodes labelled 1 into one leaf node. Finally, the two intermediate nodes labelled 3 are merged into one intermediate node, where the last merger is justified since:

$$\phi(0_1, 1_2, x_3) = \phi(1_1, 0_2, x_3) = x_3$$

In order to calculate probabilities of the events in a reduced diagram, we need to generalize (4.24). For any event E associated with a node v in a binary decision diagram, we denote by $\mathcal{P}(E)$ the set of events associated to the parent nodes of the node v. If $E' \in \mathcal{P}(E)$, the event E' is said to be a *parent event* of the event E. When considering reduced diagrams, there will typically be events with more than one parent event. Moreover, the parent events may be located at different levels in the diagram. If v is a node in the diagram, and E is the corresponding event, the probability of this event is given by:

$$P(E) = \sum_{E' \in \mathcal{P}(E)} P(E') \cdot P(E' \to E), \tag{4.25}$$

where $P(E' \to E) = q_{i(E')}$ if v is a lefthand child of the parent node corresponding to E', $P(E' \to E) = p_{i(E')}$ if v is a righthand child of the parent node corresponding to E', and $i(E')$ denotes the level of the event E'.

Using (4.25) on the diagram shown in Figure 4.10 we get the following event probabilities:

$$\begin{aligned}
P(E_{2,1}) &= P(E_{1,1}) \cdot P(E_{1,1} \to E_{2,1}) = q_1 \\
P(E_{2,2}) &= P(E_{1,1}) \cdot P(E_{1,1} \to E_{2,2}) = p_1 \\
P(E_{3,1}) &= P(E_{2,1}) \cdot P(E_{2,1} \to E_{3,1}) + P(E_{2,2}) \cdot P(E_{2,2} \to E_{3,1}) \\
&= q_1 p_2 + p_1 q_2 \\
P(\phi(\boldsymbol{X}) = 0) &= P(E_{2,1}) \cdot P(E_{2,1} \to E_{4,1}) + P(E_{3,1}) \cdot P(E_{3,1} \to E_{4,1}) \\
&= q_1 q_2 + q_1 q_3 + q_2 q_3 - 2 q_1 q_2 q_3 \\
P(\phi(\boldsymbol{X}) = 1) &= P(E_{2,2}) \cdot P(E_{2,2} \to E_{4,2}) + P(E_{3,1}) \cdot P(E_{3,1} \to E_{4,2}) \\
&= p_1 p_2 + p_1 p_3 + p_2 p_3 - 2 p_1 p_2 p_3
\end{aligned}$$

Thus, we see that we get the same distribution for $\phi(\boldsymbol{X})$ as we did using the previous diagrams.

The reduction technique introduced in [12] can in many cases reduce the computational complexity of reliability calculations. As noted by [12], however, the ordering of the input variables, or components in our context, has a large impact on the size of the diagram. Unfortunately, finding an ordering that minimises the size of the graph is itself known to be a *co-NP-Complete* problem (see [12]). Thus, most algorithms for constructing reduced ordered binary decision diagrams rely on some sort of heuristic method for ordering the input variables. Here, we will not go further into the problem of finding efficient orderings, but instead assume that an ordering of the input variables has been chosen. Given an ordering of the input variables, the problem of constructing a reduced ordered binary decision diagram is still not trivial. One possible way of doing this, is to start out by constructing an ordered binary decision diagram, and then do the reductions by identifying isomorphic subgraphs. This is the method suggested in [12]. There are two main challenges with this method. Firstly, the size of the ordered binary decision diagram may be very large, as the number of nodes typically grows exponentially in the number of input variables. The second challenge is that identifying isomorphic subgraphs can be computationally difficult as well.

A different approach is to do the reductions along the way as a part of the initial construction. This can be done level by level as follows:

Algorithm 4.8.2 *Let $\phi = \phi(\boldsymbol{x})$ be a binary function, where $\boldsymbol{x} = (x_1, \ldots, x_n)$ is indexed in accordance with a given order of the input variables.*

STEP 1. *At level* 1 *create the root node with the function $\phi(\boldsymbol{x})$, and create two leaf nodes with functions* 0 *and* 1 *respectively.*

- *Label the root node by the index* 1.

...

STEP k. *At level k create the lefthand and righthand children of each of the nodes at level $(k-1)$ with functions obtained by fixing the value of x_{k-1} to* 0 *and* 1 *respectively.*

- *Merge nodes with equal functions (if any).*
- *Merge nodes with trivial functions with the relevant leaf nodes (if any).*
- *Label all the intermediate nodes by the index k.*

...

For Algorithm 4.8.2 to work, we need to be able to identify identical functions. This may seem like a trivial task, at least when considering simple

examples. However, in general determining whether or not two binary functions are identical is an NP-complete problem. Still, in special cases when we are able to represent the binary functions in an efficient way, it can be a manageable task. Note that such a representation needs to work not only for the original binary function $\phi(\boldsymbol{x})$, but also for all the *minors* obtained in the construction process.

Algorithm 4.8.2 is implemented in python in Script B.2.2 (c_robdd.py) in Appendix B.2. Just as for Script B.2.1 this is also a *generic* implementation which can be used on different types of systems. However, in order to apply this script to a given system, some of the generic methods must be tailored to fit the system under consideration. In particular, the following methods must be implemented in a system specific way:

- A method for representing the binary functions obtained as a result of fixing the values of the input variables (*restriction* and *contraction*).
- A method for determining if a binary function is trivial.
- A method for determining if two binary functions are identical.

Again the actual implementation of these methods depend strongly on the class of systems under consideration, and on the chosen representation of the binary functions.

In the following subsections we will consider more specific classes of systems, and show how these classes can be analysed using binary decision diagrams.

4.8.1 Threshold systems

We recall from Definition 4.5.1 in Section 4.5 that structure function of a threshold system has the following form:

$$\phi(\boldsymbol{x}) = I(\sum_{i=1}^{n} a_i x_i \geq b), \tag{4.26}$$

where $a_1, \ldots, a_n$ and b are non-negative real numbers. In order to analyse such systems we need to describe how to implement the methods listed in relation to the generic implementation. We start out by assuming that we are given a threshold system (C, ϕ), where $C = \{1, \ldots, n\}$, and with weights

$a_1, \ldots, a_n$ and threshold b. The structure function is uniquely defined by the weights and threshold, and we regard these quantities as a representation the system. That is, we say that (C, ϕ) is a threshold system with parameters $\boldsymbol{a}$ and b, and refer to such a system as a TS$(\boldsymbol{a}, b)$-system.

For threshold systems it is fairly obvious that the input variables should be ordered with respect to their weights, so the input variable 1 is the one with the highest weight, input variable 2 is the one with the second highest weight etc. In cases where two or more input variables have equal weights, we simply break ties arbitrarily.

In order to represent the binary functions obtained by fixing the value of say, the jth input variable, we apply (4.26) and get:

$$\phi(0_j, \boldsymbol{x}) = I(\sum_{i \neq j} a_i x_i \geq b),$$
$$\phi(1_j, \boldsymbol{x}) = I(\sum_{i \neq j} a_i x_i \geq b - a_j)$$

From this it follows that the restriction and the contraction of (C, ϕ) with respect to component j are both threshold systems. Thus, the class of threshold systems is closed under minor operations. More specifically, it follows that the restriction of (C, ϕ) with respect to j is a TS$(\boldsymbol{a}_{-j}, b)$-system, while the contraction of (C, ϕ) with respect to j is a TS$(\boldsymbol{a}_{-j}, b - a_j)$-system, where $\boldsymbol{a}_{-j} = (a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n)$. As more of the input variables get fixed values, the set of weights is reduced. At the same time the threshold is either kept unchanged in the case of restrictions and reduced by the weight of the selected input variable in the case of contractions.

If the sum of the remaining weights are less than the threshold, we know that the minor is trivial with structure function identical to 0. Conversely, if the threshold value is less than or equal to zero, then the minor is trivial with structure function identical to 1 . Thus, in order to check whether a binary function is trivial, we only need to compute the sum of the weights and compare it to the threshold.

Since the structure function of a threshold system is uniquely defined by the weights and threshold, it is very easy to determine whether or not two binary functions are identical. When constructing a ROBDD for a given threshold system, we need to choose an ordering of the input variables. As already mentioned the size of the diagram is minimised if the input variables are ordered in descending order with respect to their respective weights. In

order to simplify the implementation we assume that the input variables are indexed so that $a_1 \geq a_2 \geq \cdots \geq a_n$. Hence, at level j of the diagram all nodes will have threshold functions with identical weights $a_j, a_{j+1}, \ldots, a_n$. Hence, in order to identify nodes with identical functions, we only need to compare the threshold values.

A python implementation of the above methods can be found in Script B.2.3 (c_threshold.py) in Appendix B.2.

Example 4.8.3 *Let (C, ϕ) be a threshold system with weights $a_1 = 8$, $a_2 = 7$, $a_3 = 6$, $a_4 = 5$, $a_5 = 2$, $a_6 = 2$ and threshold $b = 20$. A python script for constructing a BDD and a ROBDD for this system as well as calculating the unreliability and reliability can be found in Script B.2.4 in Appendix B.2. Note that in order to run this script, the class files* c_bdd.py, c_robdd.py *and* c_threshold.py *are needed as well. The script uses a common component reliability of $p = 0.5$ for all components. The two lines in the script for calculating the unreliability and reliability are:*

```
result = sys.calculateReliability0(p)
```

The result of this calculation is:

$$P(\phi(\boldsymbol{X} = 0) = 0.703125$$
$$P(\phi(\boldsymbol{X} = 1) = 0.296875$$

In cases where the components have different reliabilities, one can modify the script by instead letting p be a vector, e.g.:

```
p = [0.80, 0.75, 0.82, 0.69, 0.91, 0.78]
```

The lines in the script for calculating the unreliability and reliability should then be replaced by:

```
result = sys.calculateReliability(p)
```

The result of this calculation is:

$$P(\phi(\boldsymbol{X} = 0) = 0.215607$$
$$P(\phi(\boldsymbol{X} = 1) = 0.784393$$

By running the script one gets a brief description of the BDD and the ROBDD in the form of lists of the intermediate nodes in the diagrams sorted

by level. For each node the corresponding event probability is given as well. Since both diagrams use the same ordering of the input variables, the number of intermediate nodes in the BDD is slightly higher than in the ROBDD. A similar example can be found in Script B.2.5 in Appendix B.2.

It can be shown that if we compute the reliability of a threshold system (C, ϕ) using a reduced ordered binary decision diagram, where the components are indexed so that $a_1 \geq a_2 \geq \cdots \geq a_n$, then the computational complexity is the same as if we use the algorithm presented in Section 4.5. In fact, a ROBDD-based algorithm is typically faster than the algorithm presented in Section 4.5 as only the events that contributes to the reliability (or unreliability) are included.

4.8.2 Consecutive threshold systems

The class of consecutive k-out-of-n systems have been studied very extensively in the literature. Among the first papers on this subject are [19] and [21]. More recent papers include [18], [66] and [26]. There are several variants of such systems including *linear* and *circular* systems. Here, we focus on the linear case only.

The concept of *consecutive systems* is based on the notion of a *consecutive set*. Thus, if $C = \{1, \ldots, n\}$ is a set of components, then a consecutive set $P \subseteq C$ is a set of the form $\{j, j+1, \ldots, j+r\}$. Note that the notion of a consecutive set is defined relative to the order of indices of the set C. A consecutive set is said to be functioning if all its components are functioning. Finally, a consecutive k-out-of-n system is a system which is functioning if and only if it contains a functioning consecutive set of size at least k.

More formally, we define a consecutive k-out-of-n system as follows[3]:

Definition 4.8.4 *Let (C, ϕ) be a binary monotone system of order n, and let $\boldsymbol{x} = (x_1, \ldots, x_n)$ denote the vector of component state variables. Moreover, let $s_0 = 0$, and let:*

$$s_i = (s_{i-1} + 1) \cdot x_i, \quad i = 1, \ldots n. \tag{4.27}$$

[3]The formal definition may seem to be unnecessarily complicated. However, the recursive relation (4.27) turns out to make it easier to compute the reliability of the system. In particular, this relation is useful when constructing BDDs and ROBDDs.

Then (C, ϕ) is said to be a consecutive k-out-of-n system if ϕ has the following form:

$$\phi(\boldsymbol{x}) = I(\max_{1 \le i \le n} s_i \ge k)$$

By (4.27) s_i can be interpreted as the length of the functioning consecutive set whose last index is i. If $x_i = 1$, then $s_i = s_{i-1} + 1$, while if $x_i = 0$, then $s_i = 0$. Furthermore, the variable $\max_{1 \le i \le n} s_i$ is equal to the length of the longest functioning consecutive set of the system.

As an illustration we let $\boldsymbol{x} = (1, 0, 1, 0, 1, 1, 1)$. In this case the longest functioning consecutive set is $\{5, 6, 7\}$ which is of size 3. Calculating $s_1, \ldots, s_7$ we get:

$$\begin{aligned}
s_1 &= (s_0 + 1) \cdot x_1 = 1 \cdot 1 = 1, \\
s_2 &= (s_1 + 1) \cdot x_2 = 3 \cdot 0 = 0, \\
s_3 &= (s_2 + 1) \cdot x_3 = 1 \cdot 1 = 1, \\
s_4 &= (s_3 + 1) \cdot x_4 = 2 \cdot 0 = 0, \\
s_5 &= (s_4 + 1) \cdot x_5 = 1 \cdot 1 = 1, \\
s_6 &= (s_5 + 1) \cdot x_6 = 2 \cdot 1 = 2, \\
s_7 &= (s_6 + 1) \cdot x_7 = 3 \cdot 1 = 3.
\end{aligned}$$

Hence, we indeed get that $\max_{1 \le i \le 7} s_i = s_7 = 3$. If e.g., $k = 3$, we get $\phi(\boldsymbol{x}) = 1$.

It is easy to see that the minimal path sets of a consecutive k-out-of-n system (C, ϕ) are the consecutive sets of size k. Hence, compared to an ordinary k-out-of-n system where all subsets of C of size k are minimal paths, a consecutive k-out-of-n system has much fewer minimal path sets. Thus, the reliability of a consecutive k-out-of-n system is typically significantly less than the reliability of an ordinary k-out-of-n system with the same component set.

In Section 4.5 we introduced threshold systems as a generalisation of k-out-of-n systems. In a similar way we now introduce *consecutive threshold systems* as a generalisation of consecutive k-out-of-n systems. Just as for ordinary threshold systems the components of a consecutive threshold system have non-negative *weights*. The weight of a functioning consecutive set is the sum of the weights of the components in this set. Finally, a consecutive threshold system is a system which is functioning if and only if it contains a

functioning consecutive set with weight at least b, where b is referred to as the *threshold* of the system.

More formally, we define a consecutive threshold system as follows:

Definition 4.8.5 *Let (C, ϕ) be a binary monotone system of order n, and let $\boldsymbol{x} = (x_1, \ldots, x_n)$ denote the vector of component state variables. Moreover, let $a_1, \ldots, a_n$ and b be non-negative real numbers, let $s_0 = 0$, and let:*

$$s_i = (s_{i-1} + a_i) \cdot x_i, \quad i = 1, \ldots n. \tag{4.28}$$

Then (C, ϕ) is said to be a consecutive *threshold system if ϕ has the following form:*

$$\phi(\boldsymbol{x}) = I(\max_{1 \leq i \leq n} s_i \geq b)$$

By (4.28) s_i can be interpreted as the weight of the functioning consecutive set whose last index is i. If $x_i = 1$, then $s_i = s_{i-1} + a_i$, while if $x_i = 0$, then $s_i = 0$. Furthermore, the variable $\max_{1 \leq i \leq n} s_i$ is equal to the largest weight of a functioning consecutive set of the system.

Note that a consecutive threshold system (C, ϕ) where all weights are equal to 1 and where the threshold is equal to k, is a consecutive k-out-of-n system. Thus, just as ordinary threshold systems are a generalisation of k-out-of-n systems, consecutive threshold systems are a generalisation of consecutive k-out-of-n systems.

As an illustration we let $\boldsymbol{x} = (1, 0, 1, 0, 1, 1, 1)$ and $\boldsymbol{a} = (5, 6, 3, 7, 2, 1, 1)$. In this case the functioning consecutive set with largest weight is $\{1\}$ which has a weight of 5. Calculating $s_1, \ldots, s_7$ we get:

$$\begin{aligned}
s_1 &= (s_0 + 5) \cdot x_1 = 5 \cdot 1 = 5, \\
s_2 &= (s_1 + 6) \cdot x_2 = 11 \cdot 0 = 0, \\
s_3 &= (s_2 + 3) \cdot x_3 = 3 \cdot 1 = 3, \\
s_4 &= (s_3 + 7) \cdot x_4 = 10 \cdot 0 = 0, \\
s_5 &= (s_4 + 2) \cdot x_5 = 2 \cdot 1 = 2, \\
s_6 &= (s_5 + 1) \cdot x_6 = 3 \cdot 1 = 3, \\
s_7 &= (s_6 + 1) \cdot x_7 = 4 \cdot 1 = 4.
\end{aligned}$$

Hence, $\max_{1 \leq i \leq 7} s_i = s_1 = 5$. If e.g., $b = 4$, we get $\phi(\boldsymbol{x}) = 1$.

Since a consecutive threshold system is uniquely defined by the weights $\boldsymbol{a}$ and the threshold b, we may regard these as a representation of the system. More formally, a binary monotone system (C, ϕ) defined as in Definition 4.8.5 is said to be a consecutive threshold system with parameters $\boldsymbol{a}$ and b, and we refer to such a system as a CTS$(\boldsymbol{a}, b)$-system.

It is easy to see that both the class of consecutive threshold systems and the smaller class of consecutive k-out-of-n systems are *not closed* under minor operations. This affects the implementation of the methods for constructing BDD and ROBDD representations of such systems. Fortunately, if we handle input variables in the same order as the index order of component set C, it is still possible to implement the necessary methods in an efficient way. In order to do so it is convenient to consider a slightly more general class of systems defined as follows:

Definition 4.8.6 *Let (C, ϕ) be a binary monotone system of order n, and let $\boldsymbol{x} = (x_1, \ldots, x_n)$ denote the vector of component state variables. Moreover, let $a_1, \ldots, a_n$, b and s_0 be non-negative real numbers, and let:*

$$s_i = (s_{i-1} + a_i) \cdot x_i, \quad i = 1, \ldots n. \tag{4.29}$$

Then (C, ϕ) is said to be an extended consecutive threshold system if ϕ has the following form:

$$\phi(\boldsymbol{x}) = I(\max_{1 \le i \le n} s_i \ge b)$$

The only difference between a consecutive threshold system and an extended consecutive threshold system is that for the latter class we allow s_0 to be an arbitrary non-negative number, not just 0. The value s_0 is referred to as the *initial value* of the system. More formally, a binary monotone system (C, ϕ) defined as in Definition 4.8.6 is said to be an extended consecutive threshold system with parameters $\boldsymbol{a}$, b and s_0, and we refer to such a system as an XCTS$(\boldsymbol{a}, b, s_0)$-system. Note that a CTS$(\boldsymbol{a}, b)$-system is the same as an XCTS$(\boldsymbol{a}, b, 0)$-system. That is, consecutive threshold systems are special cases of extended consecutive threshold systems.

We now let (C, ϕ) be an XCTS$(\boldsymbol{a}, b, s_0)$-system, and consider a binary monotone function corresponding to an intermediate node v at level j in a BDD or ROBDD. At this level the first $j-1$ input variables are fixed. We let x'_i denote the fixed value of x_i, $i = 1, \ldots, j-1$, and let $s'_i = (s'_{i-1} + a_i) \cdot x'_i$, $i = 1, \ldots, j-1$ denote the corresponding weight sums. If $s'_i \ge b$ for some

$0 \leq i \leq j-1$, it follows that the function is trivial with value 1. However, since the node v is assumed to be an intermediate node, not a leaf node, this cannot be the case. Thus, it follows that $s'_i < b$, $i = 0, 1, \ldots, j-1$. We then let $D = \{j, \ldots, n\}$, $\boldsymbol{x}_D = (x_j, \ldots, x_n)$, $\boldsymbol{a}_D = (a_j, \ldots, a_n)$, and let $\phi_D = \phi_D(\boldsymbol{x}_D)$ denote the binary function corresponding to the node v. Then it follows that:

$$\phi_D(\boldsymbol{x}_D) = \phi(x'_1, \ldots, x'_{j-1}, \boldsymbol{x}_D) = I(\max_{j \leq i \leq n} s_i \geq b)$$

From this it is easy to see that (D, ϕ_D) is an XCTS$(\boldsymbol{a}_D, b, s'_{j-1})$-system. Thus, we have shown that all binary functions associated with the intermediate nodes in the diagram are extended consecutive threshold systems.

By a similar argument it is easy to show that the restriction of (D, ϕ_D) with respect to component j is an XCTS$(\boldsymbol{a}_{D\setminus j}, b, 0)$-system, while the contraction of (D, ϕ_D) with respect to component j is an XCTS$(\boldsymbol{a}_{D\setminus j}, b, s'_{j-1} + a_j)$-system, where $\boldsymbol{a}_{D\setminus j} = (a_{j+1}, \ldots, a_n)$.

In order to explain how to check if an extended consecutive threshold system is trivial, we note that trivial systems always occur as a result of a restriction or a contraction. In fact a trivial system with value 0 can only occur as a result of a restriction, while a trivial system with value 1 can only occur as a result of a contraction. Thus, it is sufficient to describe how to check for trivial systems for the restriction and contraction of (D, ϕ_D). The restriction of (D, ϕ_D) with respect to component j is trivial with value 0 if and only if $a_{j+1} + \cdots + a_n < b$, while the contraction of (D, ϕ_D) with respect to component j is trivial with value 1 if and only if $s'_{j-1} + a_j \geq b$.

Since the structure function of an extended consecutive threshold system is uniquely defined by the weights, threshold and initial value, it is very easy to determine whether or not two binary functions are identical. At level j of the diagram all nodes will have identical weights $a_j, a_{j+1}, \ldots, a_n$ and threshold value b. Hence, in order to identify nodes with identical functions, we only need to compare the initial values.

A python implementation of the above methods can be found in Script B.2.6 (c_conthreshold.py) in Appendix B.2.

Example 4.8.7 *Let (C, ϕ) be a consecutive threshold system with weights $a_1 = 8$, $a_2 = 7$, $a_3 = 6$, $a_4 = 5$, $a_5 = 3$, $a_6 = 2$ and threshold $b = 10$. A python script for constructing a BDD and a ROBDD for this system as well as calculating the unreliability and reliability can be found in Script B.2.7 in*

Appendix B.2. Note that in order to run this script, the class files c_bdd.py, c_robdd.py *and* c_conthreshold.py *are needed as well. The script uses a common component reliability of* $p = 0.5$ *for all components. The two lines in the script for calculating the unreliability and reliability are:*

```
result = sys.calculateReliability0(p)
```

The result of this calculation is:

$$P(\phi(\boldsymbol{X} = 0) = 0.453125$$
$$P(\phi(\boldsymbol{X} = 1) = 0.546875$$

In cases where the components have different reliabilities, one can modify the script by instead letting p *be a vector, e.g.:*

```
p = [0.80, 0.75, 0.82, 0.69, 0.91, 0.78]
```

The lines in the script for calculating the unreliability and reliability should then be replaced by:

```
result = sys.calculateReliability(p)
```

The result of this calculation is:

$$P(\phi(\boldsymbol{X} = 0) = 0.100287$$
$$P(\phi(\boldsymbol{X} = 1) = 0.899713$$

4.8.3 Undirected network systems

Although the factoring algorithm presented in Section 4.7 is usually faster, binary decision diagrams have been used to analyse undirected network systems as well. See e.g., [9]. In Section 4.7 we noted that the class of such systems is closed under minor operations. This makes it easy to handle restrictions and contractions. In fact, the minor operations have nice graphical interpretations which makes it easy to implement these operations as part of an algorithm. See Figure 4.7 and Figure 4.7.

When implementing network algorithms, networks can be represented very efficiently using an object oriented framework with objects like nodes and edges which can be connected to each other using pointers. To see how this can be done in python we refer to the Script B.2.8 in Appendix B.2.

This framework also includes implementations of minor operations as well as standard methods for checking the state of a network.

Another convenient representation of an undirected network is the *node-edge incidence matrix* or simply the *incidence matrix*. This is a binary matrix with one row for each node, and one column for each edge. If $V = \{v_1, \ldots, v_m\}$ is the set of nodes, and $E = \{e_1, \ldots, e_n\}$ is the set of edges, then the incidence matrix of the network is a matrix, $M = M^{m \times n}$ defined as follows:

$$M_{ij} = \begin{cases} 1 & \text{if node } v_i \text{ and edge } e_j \text{ are connected} \\ 0 & \text{otherwise} \end{cases} \tag{4.30}$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. In most cases an edge is connected to *two nodes*, which implies that the column representing the edge contains two entries which are 1, while the remaining entries are 0. However, we also allow loops in the network, i.e., edges connected to just one node. For columns representing loops there is only one entry which is 1. Loops obviously have no impact on the connectivity of a network, so they could just as well be removed from the system. Thus, initially a network will typically be loop-free. However, loops may occur as results of contractions. Thus, even though a network of interest is loop-free, such loops may occur in minors.

Obviously, if we are given the incidence matrix M it is easy to implement an algorithm that constructs the corresponding network. Conversely, if we are given a network with node set V and edge set E where the nodes and edges are indexed in a suitable way, it is easy to derive the incidence matrix M.

In order to fully specify a K-terminal undirected network system, we also need to include the set, T of terminals, where T simply contains the indices of the K nodes which are terminals. Given M and T the undirected network system is uniquely determined, and we refer to suc a system as a $\mathrm{N}(M, T)$-system.

In the Script B.2.8 (c_graph.py) in Appendix B.2 we use both the $\mathrm{N}(M, T)$-system representation and the object orient framework. The object oriented framework is very efficient for implementing minor operations and for computing the state of the network. The $\mathrm{N}(M, T)$-system representation is useful as a generic way of specifying the system, as well as for identifying nodes in the binary decision network with identical functions. In general, one can permute the rows and the columns of a $\mathrm{N}(M, T)$-system without changing

the system. Comparing two N(M,T)-systems where one is a permutation of the other is not easy. Fortunately, it is possible to implement restrictions and contractions in such a way that such issues are avoided. As a result minors can be compared simply by comparing their respective incidence matrices and terminal sets.

Note that the Script B.2.8 (c_graph.py) in Appendix B.2 does not include methods for finding an optimal ordering of the input variables. Instead the input variables are handled according to the index order of the corresponding component set C.

Example 4.8.8 *Let (C,ϕ) be the 2-terminal undirected network system shown in Figure 4.12. The incidence matrix for this system is:*

$$M = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

and the terminal list is $T = \{1, 6\}$. A python script for constructing a BDD and a ROBDD for this system as well as calculating the unreliability and reliability can be found in Script B.2.10 in Appendix B.2. Note that in order to run this script, the class files c_bdd.py, c_robdd.py *and* c_graph.py *are needed as well. The script uses a common component reliability of $p = 0.5$ for all components. The two lines in the script for calculating the unreliability and reliability are:*

```
result = sys.calculateReliability0(p)
```

The result of this calculation is:

$$P(\phi(\boldsymbol{X} = 0) = 0.671875$$
$$P(\phi(\boldsymbol{X} = 1) = 0.328125$$

In cases where the components have different reliabilities, one can modify the script by instead letting p be a vector, e.g.:

```
p = [0.80, 0.75, 0.82, 0.69, 0.91, 0.78, 0.55, 0.78]
```

The lines in the script for calculating the unreliability and reliability should then be replaced by:

```
result = sys.calculateReliability(p)
```

The result of this calculation is:

$$P(\phi(\boldsymbol{X} = 0) = 0.192079$$
$$P(\phi(\boldsymbol{X} = 1) = 0.807921$$

Note that for this system the reduced ordered binary decision diagram is significantly smaller than the binary decision diagram.

4.9 Exercises

Exercise 1. Let (C, ϕ) be a 7-out-of-10 system where the components have the following reliabilities:

$$p_1 = 0.65, \quad p_2 = 0.70, \quad p_3 = 0.69, \quad p_4 = 0.61, \quad p_5 = 0.66$$
$$p_6 = 0.59, \quad p_7 = 0.72, \quad p_8 = 0.62, \quad p_9 = 0.59, \quad p_{10} = 0.76$$

Using Script B.1.1 in Appendix B.1 as a template, write a python script for calculating the reliability of this system. [*Answer*: $h = 0.5382$.]

Exercise 2. Let S be a stochastic variable with values in $\{0, 1, \ldots, n\}$. We then define the *generating function* of S as:

$$G_S(y) = E[y^S] = \sum_{s=0}^{n} y^s P(S = s). \tag{4.31}$$

a) Explain why $G_S(y)$ is a polynomial, and give an interpretation of the coefficients of this polynomial.
b) Let T be another non-negative integer valued stochastic variable with values in $\{0, 1, \ldots, m\}$ which is independent of S. Show that:

$$G_{S+T}(y) = G_S(y) \cdot G_T(y). \tag{4.32}$$

c) Let $X_1, \ldots, X_n$ be independent binary variables with $P(X_i = 1) = p_i$ and $P(X_i = 0) = 1 - p_i = q_i$, $i = 1, \ldots, n$. Show that:

$$G_{X_i}(y) = q_i + p_i y, \quad i = 1, \ldots, n. \tag{4.33}$$

d) Introduce:

$$S_j = \sum_{i=1}^{j} X_i, \quad j = 1, 2, \ldots, n,$$

and assume that we have computed $G_{S_j}(y)$. Thus, all the coefficients of $G_{S_j}(y)$ are known at this stage. We then compute:

$$G_{S_{j+1}}(y) = G_{S_j}(y) \cdot G_{X_{j+1}}(y).$$

How many algebraic operations (addition and multiplication) will be needed to complete this task?

e) Explain how generating functions can be used in order to calculate the reliability of a k-out-of-n system. What can you say about the order of this algorithm?

Exercise 3. Let (C, ϕ) be a threshold system with structure function:

$$\phi(\boldsymbol{X}) = I(\sum_{i=1}^{n} a_i X_i \geq b),$$

where $a_1, \ldots, a_n$ and b are non-negative real numbers. Show that the dual of this system is a threshold system as well.

Exercise 4. Let (C, ϕ) be a threshold system of 8 components with the following reliabilities:

$$\begin{aligned} p_1 = 0.75, \quad p_2 = 0.80, \quad p_3 = 0.82, \quad p_4 = 0.65 \\ p_5 = 0.88, \quad p_6 = 0.91, \quad p_7 = 0.92, \quad p_8 = 0.86 \end{aligned}$$

The component weights are:

$$\begin{aligned} a_1 = 26, \quad a_2 = 39, \quad a_3 = 65, \quad a_4 = 78 \\ a_5 = 91, \quad a_6 = 104, \quad a_7 = 104, \quad a_8 = 143 \end{aligned}$$

and the threshold is $b = 410$.

a) Using Script B.1.2 in Appendix B.1 as a template, write a python script for calculating the reliability of this system. [*Answer*: $h = 0.9238$.]

b) Let $S = \sum_{i=1}^{n} a_i X_i$. By considering the output of the python script used in (a) show that the set of possible values for S is a subset of the set $\{0, 1, \ldots, 650\}$.

c) From the output of the python script used in (a) we note that many of the values in the set $\{0, 1, \ldots, 650\}$ has zero probability. Thus, the set of possible values for S is only a *small* subset of the set $\{0, 1, \ldots, 650\}$. To avoid computing probabilities of impossible values, one should identify the largest common factor of the weights $a_1, \ldots, a_8$. We denote this common factor by c, and introduce $\tilde{a}_i = a_i/c$, $i = 1, \ldots, 8$. We also introduce $\tilde{S} = \sum_{i=1}^{n} \tilde{a}_i X_i$. Explain why the reliability of the system is equal to $P(c \cdot \tilde{S} \geq b)$.

d) A python script where this technique is utilised, can be found in B.1.3 in Appendix B.1. Run this script and compare the results to the output of the python script used in (a).

Exercise 5. Consider the bridge structure from Example 3.1.2 with independent component state variables and reliability function $h(\boldsymbol{p})$ as given in this example. Assume that the reliability of all five components is 0.9. What do the bounds (4.12) reduce to in this case? Comment on the result.

Exercise 6. Draw an example of the following systems:

(i) An S3T system.

(ii) An S5T system.

Exercise 7. Consider the S1T system in Figure 4.11.

a) Find the minimal path sets of the system.

b) How many terms will there be in the inclusion-exclusion formula for the reliability of this system before simplification?

c) How many of these terms will vanish in the final simplified expression?

Exercise 8. Show that the linear consecutive 2-out-of-5 system given in Example 4.6.5 cannot be an SKT-system. [*Hint:* Let δ denote the signed domination function of the system. Compare $\delta(\{1, 2, 3, 4\})$ and $\delta(\{1, 2, 3, 4, 5\})$ and interpret the result in the light of Theorem 4.6.3.]

Exercise 9. Consider the undirected network system in Figure 4.12 which functions if and only if the nodes S and T can communicate through the network.

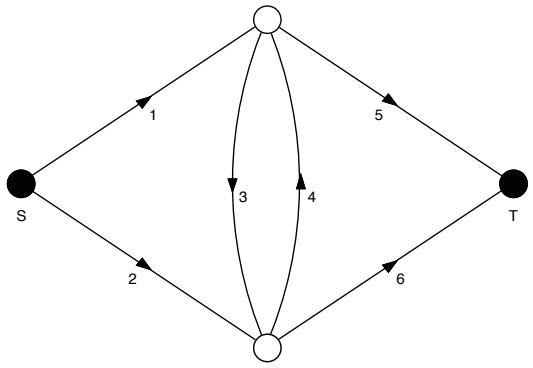

Figure 4.11: An S1T system.

a) Compute the reliability function of the system using the factoring algorithm.
b) Assume more specifically that $p_1 = \cdots = p_8 = 0.5$. Compute the reliability of the system.

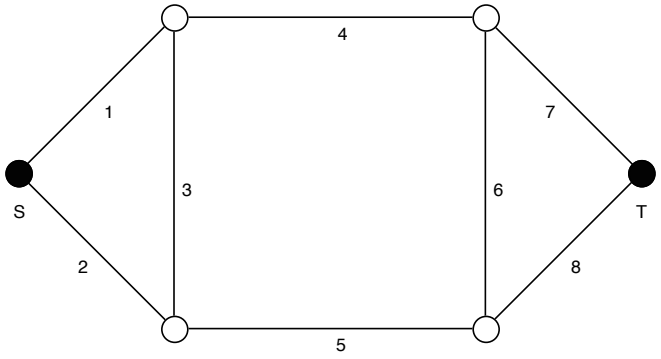

Figure 4.12: An undirected network system with components $1, 2, \ldots, 8$ and terminals S and T.

Exercise 10. Compute the reliability function of the undirected network

system in Figure 4.13 by using the factoring algorithm.

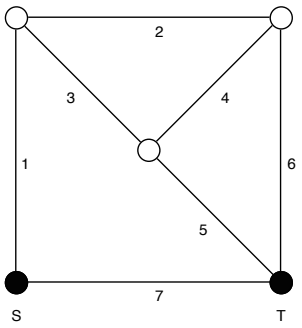

Figure 4.13: An undirected network system with components $1, 2, \ldots, 7$ and terminals S and T.

Since the system is complex, it has to be factored with respect to some component. Which component? Compare the computational work for different component choices.

Exercise 11. Consider the series connection of bridge structures in Figure 4.14.

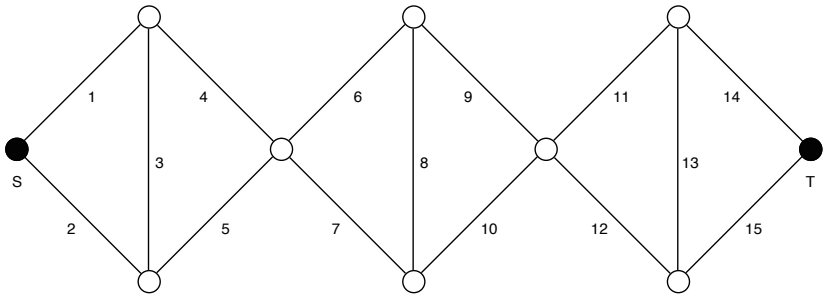

Figure 4.14: A series system of bridge structures.

a) Compute the reliability function of the system by using the factoring algorithm.

b) Instead, compute this by taking the product of the three reliability functions of the three bridge structures. Compare the computational effort with the one in a).

c) How many terms are there in the inclusion-exclusion formula for the reliability of this system? Comment this result.

Exercise 12. Prove Corollary 4.3.3.

Exercise 13. Construct a reduced ordered binary decision diagram of a 3-out-5-system.

Exercise 14. Explain briefly why the number of nodes in a reduced ordered binary decision diagram of a k-out-n-system grows polynomially in n.

Exercise 15. Let (C, ϕ) be a threshold system with weights $a_1 = 8$, $a_2 = 6$, $a_3 = 6$, $a_4 = 4$, $a_5 = 4$, $a_6 = 2$ and threshold $b = 15$.

a) Assume that all components have reliability $p = 0.5$. Using the Script B.2.4 in Appendix B.2 as a template write a python script for calculating the reliability of the system. [*Answer*: $P(\phi = 1) = 0.5$.]

b) Run the script from (a) again, but with the weights in reversed order, i.e., $a_1 = 2$, $a_2 = 4$, $\ldots$, $a_6 = 6$. Compare the number of intermediate nodes in the ROBDD with the number of nodes in ROBDD from (a).

c) Assume instead that $p_1 = 0.82$, $p_2 = 0.78$, $p_3 = 0.66$, $p_4 = 0.91$, $p_5 = 0.73$, $p_6 = 0.88$. Modify the script from (a) and calculate the reliability of the system with these component reliabilities. [*Answer*: $P(\phi = 1) = 0.932025$.]

Exercise 16. Let (C, ϕ) be a consecutive threshold system with weights $a_1 = 8$, $a_2 = 6$, $a_3 = 6$, $a_4 = 4$, $a_5 = 4$, $a_6 = 2$ and threshold $b = 15$.

a) Assume that all components have reliability $p = 0.5$. Using the Script B.2.7 in Appendix B.2 as a template write a python script for calculating the reliability of the system. [*Answer*: $P(\phi = 1) = 0.21875$.]

b) Run the script from (a) again, but with the weights in reversed order, i.e., $a_1 = 2$, $a_2 = 4$, $\ldots$, $a_6 = 6$. Compare the number of intermediate nodes in the ROBDD with the number of nodes in ROBDD from (a).

c) Assume instead that $p_1 = 0.82$, $p_2 = 0.78$, $p_3 = 0.66$, $p_4 = 0.91$, $p_5 = 0.73$, $p_6 = 0.88$. Modify the script from (a) and calculate the reliability of the system with these component reliabilities. [*Answer*: $P(\phi = 1) = 0.591342$.]

Exercise 17. Consider the 2-terminal undirected network system shown in Figure 4.15.

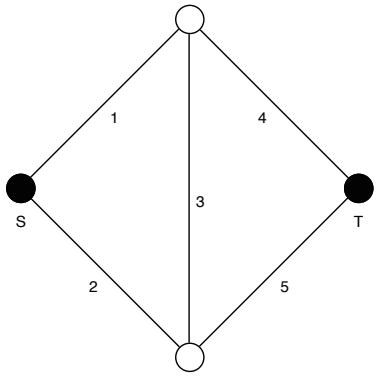

Figure 4.15: A bridge system.

a) Find the node-edge incidence matrix for this network.

b) Assume that all components have reliability $p = 0.5$. Using the Script B.2.10 in Appendix B.2 as a template write a python script for calculating the reliability of the system. Note that in the python script the four nodes of the system are indexed $0, 1, 2, 3$. Thus, the set of terminals should be $T = \{0, 3\}$ [*Answer*: $P(\phi = 1) = 0.5$.]

c) Assume instead that $p_1 = 0.82$, $p_2 = 0.78$, $p_3 = 0.66$, $p_4 = 0.91$, $p_5 = 0.73$. Modify the script from (b) and calculate the reliability of the system with these component reliabilities. [*Answer*: $P(\phi = 1) = 0.921304$.]

Chapter 5

Structural and reliability importance for components in binary monotone systems

By considering the structure- or reliability function of a binary monotone system, it is evident that not all the components are equally important for the system to function. For instance, a component connected in series or parallel with the rest of the system will typically be more important than the other components in the system. In this section, we will define various measures for importance of the components in a system. There are many reasons for considering such measures, but among the most important ones are the following:

1. A measure of importance can be used to identify components that should be improved in order to increase the system reliability.

2. A measure of importance can be used to identify components that most likely have failed, given that the system has failed.

5.1 Structural importance of a component

The measures of importance which we are about to introduce are all based on the notion of *criticality*. This is defined as follows:

Definition 5.1.1 *Let (C, ϕ) be a binary monotone system, and let $i \in C$. We say that component i is* critical for the system *if:*

$$\phi(1_i, \boldsymbol{x}) = 1 \text{ and } \phi(0_i, \boldsymbol{x}) = 0.$$

If this is the case, we also say that $(\cdot_i, \boldsymbol{x})$ is a critical vector *for component i.*

We observe that criticality is strongly related to the notion of relevance. Thus, a component i in a binary monotone system (C, ϕ) is relevant if and only if there exists at least one critical vector for i.

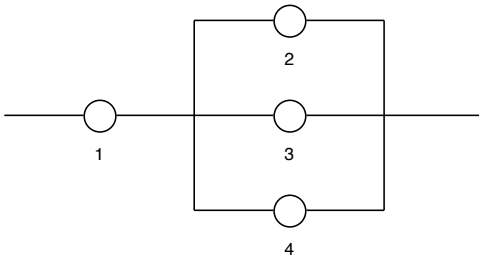

Figure 5.1:

Example 5.1.2 *Consider the binary monotone system (C, ϕ) shown in Figure 5.1, where $C = \{1, 2, 3, 4\}$, and where the structure function is given by:*

$$\phi(\boldsymbol{x}) = x_1(x_2 \amalg x_3 \amalg x_4)$$

We start out by considering component 1, and we see that this component is critical if and only if at least one of the other components are functioning. Thus, there are seven critical vectors for component 1:

$$\begin{array}{lll} (\cdot, 1, 0, 0), & (\cdot, 0, 1, 0), & (\cdot, 0, 0, 1) \\ (\cdot, 1, 1, 0), & (\cdot, 1, 0, 1), & (\cdot, 0, 1, 1) \\ (\cdot, 1, 1, 1). & & \end{array}$$

We then compare this to any of the other components, e.g., component 2. For this component to be critical component 1 must function, while both component 3 and 4 must be failed. Hence, the only critical vector for component 2 is $(1, \cdot, 0, 0)$.

By Definition 5.1.1 it follows that a component i of a binary monotone system (C, ϕ) is critical if and only if:

$$\phi(1_i, \mathbf{x}) - \phi(0_i, \mathbf{x}) = 1.$$

Hence, the number of critical vectors for i can be calculated by summing this expression over all possible vectors $(\cdot_i, \mathbf{x})$:

$$\sum_{(\cdot_i, \mathbf{x})} [\phi(1_i, \mathbf{x}) - \phi(0_i, \mathbf{x})].$$

Based on this Birnbaum [8] suggested the following measure of structural importance of a component in a binary monotone system:

Definition 5.1.3 *Let (C, ϕ) be a binary monotone system of order n, and let $i \in C$. The* Birnbaum measure for the structural importance *of component i, denoted $J_B^{(i)}$, is defined as:*

$$J_B^{(i)} := \frac{1}{2^{n-1}} \sum_{(\cdot_i, \boldsymbol{x})} [\phi(1_i, \boldsymbol{x}) - \phi(0_i, \boldsymbol{x})].$$

Note that the denominator, 2^{n-1} is the total number of states for the $n-1$ other components. Thus, $J_B^{(i)}$ can be interpreted as the fraction of all states for the $n-1$ other components where component i is critical. By computing $J_B^{(i)}$ for all of the components in a system, we can rank the components based on their respective structural importance.

Example 5.1.4 *Let ϕ be a 2-out-of-3 system. To compute the structural importance of component 1, we note that the critical vectors for this component are $(\cdot, 1, 0)$ and $(\cdot, 0, 1)$. Hence, we have:*

$$J_B^{(1)} = \frac{2}{2^{3-1}} = \frac{1}{2}.$$

By similar arguments, we find that:

$$J_B^{(2)} = J_B^{(3)} = \frac{1}{2}.$$

So in a 2-out-of-3 system, all of the components have the same structural importance. This is intuitively obvious since the structure function is symmetrical with respect to the components.

Example 5.1.5 *Consider the system ϕ with structure function:*

$$\phi(\boldsymbol{x}) = (x_1 \amalg x_2)x_3.$$

To find the structural importance of component 1, note that the only critical vector for this component is $(\cdot, 0, 1)$. Hence, we have:

$$J_B^{(1)} = \frac{1}{2^{3-1}} = \frac{1}{4}.$$

Since components 1 and 2 play completely similar roles in the system, it follows that we must have:

$$J_B^{(2)} = \frac{1}{4}$$

as well. Thus, components 1 and 2 have the same structural importance. However, this is not the case for component 3: The critical path vectors for component 3 are $(1, 0, \cdot)$, $(0, 1, \cdot)$ and $(1, 1, \cdot)$, so:

$$J_B^{(3)} = \frac{3}{2^{3-1}} = \frac{3}{4}.$$

From this, we see that:

$$J_B^{(1)} = J_B^{(2)} < J_B^{(3)}.$$

Hence, component 3, which is in series with the rest of the system, is more important than components 1 and 2.

5.2 Reliability importance of a component

While structural importance only depends on the structure function of the system, *reliability importance* takes into account the joint distribution of the component state variables as well. Thus, instead of considering the fraction of components state vectors where a given component is critical, we compute the probability that a given component is critical. This motivates the following definition introduced by Birnbaum [8]:

Definition 5.2.1 *Let (C, ϕ) be a binary monotone system, and let $i \in C$. Moreover, let $\boldsymbol{X}$ be the vector of component state variables. The Birnbaum measure for the reliability importance of component i, denoted $I_B^{(i)}$ is defined as:*

$$\begin{aligned} I_B^{(i)} &:= P(\textit{Component } i \textit{ is critical for the system}) \\ &= P(\phi(1_i, \boldsymbol{X}) - \phi(0_i, \boldsymbol{X}) = 1). \end{aligned}$$

Since the difference $\phi(1_i, \mathbf{X}) - \phi(0_i, \mathbf{X})$ is a binary variable, it follows that we may write:

$$I_B^{(i)} = E[\phi(1_i, \mathbf{X}) - \phi(0_i, \mathbf{X})] = E[\phi(1_i, \mathbf{X})] - E[\phi(0_i, \mathbf{X})]. \tag{5.1}$$

In particular, if the component state variables of the system are independent, and $P(X_i = 1) = p_i$ for $i \in C$, we get that:

$$I_B^{(i)} = h(1_i, \mathbf{p}) - h(0_i, \mathbf{p}). \tag{5.2}$$

By using the last identity we can show the following result which is sometimes taken as the definition of reliability importance in the case of independent components:

Theorem 5.2.2 *Let (C, ϕ) be a binary monotone system where the component state variables are independent, and $P(X_i = 1) = p_i$ for $i \in C$. Then:*

$$I_B^{(i)} = \frac{\partial h(\boldsymbol{p})}{\partial p_i}, \quad \text{for all } i \in C.$$

Proof: By Theorem 3.1.1 we have that:

$$h(\mathbf{p}) = p_i h(1_i, \mathbf{p}) + (1 - p_i) h(0_i, \mathbf{p})$$

By differentiating this identity with respect to p_i we get:

$$\frac{\partial h(\mathbf{p})}{\partial p_i} = h(1_i, \mathbf{p}) - h(0_i, \mathbf{p}).$$

Hence, the result follows by (5.2). □

We observe that Theorem 5.2.2 shows that the reliability importance of a component indicates how much the system reliability grows with respect to the reliability of this component.

Now, we will prove some further properties of the Birnbaum measure.

Theorem 5.2.3 *For a binary monotone system, (C, ϕ), we always have*

$$0 \leq I_B^{(i)} \leq 1. \tag{5.3}$$

Assume that the component state variables are independent, and $P(X_j = 1) = p_j$, where $0 < p_j < 1$ for all $j \in C$. If component i is relevant, we have:

$$0 < I_B^{(i)}. \tag{5.4}$$

Furthermore, if there exists at least one other relevant component, we also have:

$$I_B^{(i)} < 1. \tag{5.5}$$

Proof: We note that (5.3) follows directly from Definition 5.2.1 since the reliability importance is a probability.

We then assume that the component state variables are independent, and $P(X_j = 1) = p_j$, where $0 < p_j < 1$ for all $j \in C$. If component i is relevant, we know from Theorem 3.1.3 that h is strictly increasing in p_i. That is, we must have:

$$\frac{\partial h(\mathbf{p})}{\partial p_i} > 0.$$

Combining this with Theorem 5.2.2, we get (5.4).

Finally, we assume that there exists at least one other relevant component, say $k \in C$. In order to prove that this implies that $I_B^{(i)} < 1$, we assume conversely that $I_B^{(i)} = 1$. We will then show that this leads to a contradiction. From this assumption, it follows by Definition 5.2.1 that :

$$P(\phi(1_i, \mathbf{X}) - \phi(0_i, \mathbf{X}) = 1) = 1$$

Since $0 < p_j < 1$, for all $j \in C$, it follows that $P((\cdot_i, \mathbf{X}) = (\cdot_i, \mathbf{x})) > 0$ for all $(\cdot_i, \mathbf{x})$. Hence, we must have that:

$$\phi(1_i, \mathbf{x}) = 1 \text{ and } \phi(0_i, \mathbf{x}) = 0 \text{ for all } (\cdot_i, \mathbf{x}).$$

At the same time, since component k is relevant, there exists a vector $(\cdot_k, \mathbf{y})$ such that:

$$\phi(1_k, \mathbf{y}) = 1 \text{ and } \phi(0_k, \mathbf{y}) = 0.$$

If $y_i = 1$, it follows that $\phi(1_i, 0_k, \mathbf{y}) = 0$, contradicting that $\phi(1_i, \mathbf{x}) = 1$ for all $(\cdot_i, \mathbf{x})$. Similarly, if $y_i = 0$, it follows that $\phi(0_i, 1_k, \mathbf{y}) = 1$, contradicting that $\phi(0_i, \mathbf{x}) = 0$ for all $(\cdot_i, \mathbf{x})$. Hence, we conclude that for both possible values of y_i we end up with contradictions. Thus, the only possibility is that $I_B^{(i)} < 1$. □

We recall from Section 5.1 that $J_B^{(i)}$ can be interpreted as the fraction of all states for the $n-1$ other components where component i is critical. By Definition 5.2.1 $I_B^{(i)}$ is the probability that component i is critical. Thus, $J_B^{(i)}$ is just a special case of $I_B^{(i)}$ corresponding to the situation where all component state vectors have the same probability. This occurs when $P(X_i = 1) = \frac{1}{2}$ for all $i \in C$. Thus, we have the following result

Theorem 5.2.4 *Consider a binary monotone system (C, ϕ) where the component state variables are independent, and where $P(X_i = 1) = \frac{1}{2}$ for all $i \in C$. Then we have:*

$$I_B^{(i)} = J_B^{(i)}$$

We close this section by considering some examples. In all these examples we consider binary monotone systems (C, ϕ) where $C = \{1, \ldots, n\}$, where the component state variables are independent, and where $P(X_i = 1) = p_i$ for all $i \in C$. Without loss of generality we assume that the components are ordered so that:

$$p_1 \leq p_2 \leq \ldots \leq p_n. \tag{5.6}$$

Example 5.2.5 *Let (C, ϕ) be a series system. Then for all $i \in C$ we have:*

$$I_B^{(i)} = \frac{\partial \prod_{j=1}^n p_j}{\partial p_i} = \prod_{j \neq i} p_j.$$

Hence, by the ordering (5.6)*, we get that:*

$$I_B^{(1)} \geq I_B^{(2)} \geq \cdots \geq I_B^{(n)}.$$

Thus, in a series system the worst *component, i.e., the one with the smallest reliability, has the greatest reliability importance.*

Example 5.2.6 *Let (C, ϕ) be a parallel system. Then for all $i \in C$ we have:*

$$I_B^{(i)} = \frac{\partial \coprod_{j=1}^n p_j}{\partial p_i} = \prod_{j \neq i} (1 - p_j).$$

Hence, from the ordering (5.6)

$$I_B^{(1)} \leq I_B^{(2)} \leq \cdots \leq I_B^{(n)}.$$

Thus, in a parallel system the best *component, i.e., the one with the greatest reliability, has the greatest reliability importance.*

Example 5.2.7 *Let (C, ϕ) be a 2-out-of-3 system. It is then easy to show that:*

$$h(\boldsymbol{p}) = p_1 p_2 + p_1 p_3 + p_2 p_3 - 2 p_1 p_2 p_3.$$

This implies that:

$$I_B^{(1)} = p_2 + p_3 - 2p_2p_3,$$
$$I_B^{(2)} = p_1 + p_3 - 2p_1p_3,$$
$$I_B^{(3)} = p_1 + p_2 - 2p_1p_2.$$

We then consider the function $f(p,q) = p + q - 2pq$ *and note that* $I_B^{(1)} = f(p_2, p_3)$, $I_B^{(2)} = f(p_1, p_3)$, *and* $I_B^{(3)} = f(p_1, p_2)$. *Moreover, the partial derivatives of* f *are respectively:*

$$\frac{\partial f}{\partial p} = 1 - 2q, \qquad \frac{\partial f}{\partial q} = 1 - 2p.$$

If $p, q \leq \frac{1}{2}$, f *is non-decreasing in* p *and* q. *Thus, if* $p_1 \leq p_2 \leq p_3 \leq \frac{1}{2}$, *we have:*

$$f(p_1, p_2) \leq f(p_1, p_3) \leq f(p_2, p_3).$$

Hence, in this case we have:

$$I_B^{(3)} \leq I_B^{(2)} \leq I_B^{(1)}. \tag{5.7}$$

If $p, q \geq \frac{1}{2}$, f *is non-increasing in* p *and* q. *Thus, if* $\frac{1}{2} \leq p_1 \leq p_2 \leq p_3$, *we have:*

$$f(p_2, p_3) \leq f(p_1, p_3) \leq f(p_1, p_2).$$

Hence, in this case we have:

$$I_B^{(1)} \leq I_B^{(2)} \leq I_B^{(3)}. \tag{5.8}$$

However, other orderings are possible as well. Assume e.g., that $p_1 = \frac{1}{2} - z$, $p_2 = \frac{1}{2}$ *and* $p_3 = \frac{1}{2} + z$, *where* $z \in (0, \frac{1}{2})$. *In this case we get:*

$$I_B^{(1)} = (\frac{1}{2}) + (\frac{1}{2} + z) - 2 \cdot (\frac{1}{2})(\frac{1}{2} + z) = \frac{1}{2},$$
$$I_B^{(2)} = (\frac{1}{2} - z) + (\frac{1}{2} + z) - 2 \cdot (\frac{1}{2} - z)(\frac{1}{2} + z) = \frac{1}{2} + 2z^2,$$
$$I_B^{(3)} = (\frac{1}{2} - z) + (\frac{1}{2}) - 2 \cdot (\frac{1}{2} - z)(\frac{1}{2}) = \frac{1}{2},$$

Hence in this case we have:

$$I_B^{(1)} = I_B^{(3)} \leq I_B^{(2)}. \tag{5.9}$$

Note that this result holds also if $z \in (-\frac{1}{2}, 0)$ *in which case* $p_1 > p_2 > p_3$.

Example 5.2.7 shows that the reliability ordering of the components depends strongly on how reliable the components are. Thus, predicting the final ranking can often be difficult if we rely on our intuition only. Still some intuitive explanation can be obtained in special cases. Note e.g., that if $p_1 \leq p_2 \leq p_3 \leq \frac{1}{2}$, the components fail with a high probability. In this case the system behaves almost like a series system, which is reflected in (5.7) where the worst component is ranked as the most important component. Thus, in this case we have the same ordering as in Example 5.2.5. Similarly, if $\frac{1}{2} \leq p_1 \leq p_2 \leq p_3$, the components function with a high probability. In this case the system behaves almost like a parallel system, which is reflected in (5.8) where the best component is ranked on top. In this case we have the same ordering as in Example 5.2.6.

5.3 Exercises

Exercise 1. Let (C, ϕ) be a k-out-of-n system. Prove that all the components of this system have the same Birnbaum measure for structural importance.

Exercise 2. Compute $J_B^{(i)}$ for the components of the bridge structure in Example 3.1.2 and compare their structural importance.

Exercise 3. Let the ith component be in series with the rest of a monotone structure ϕ, while the jth component is not. Prove that

$$J_B^{(i)} > J_B^{(j)}.$$

Exercise 4. Compute $I_B^{(i)}$ for the components of the bridge structure in Example 3.1.2 and compare their reliability importance.

Exercise 5. Let (C, ϕ) be a threshold system of 4 components with the following reliabilities:

$$p_1 = 0.75, \quad p_2 = 0.80, \quad p_3 = 0.82, \quad p_4 = 0.65$$

The component weights are:

$$a_1 = 26, \quad a_2 = 39, \quad a_3 = 65, \quad a_4 = 78$$

and the threshold is $b = 80$. Using either Script B.1.2 in Appendix B.1 or Script B.2.4 in Appendix B.2 as a template, calculate $I_B^{(i)}$ for the components

of the system. [*Hint:* In order to compute the reliability importance of component i, calculate $h(1_i, \boldsymbol{p})$ and $h(0_i, \boldsymbol{p})$, and consider the difference between these two values.]

Chapter 6

Conditional Monte Carlo methods

As a result of the availability of high-speed computers, the use of Monte Carlo methods have accelerated. In many cases, however, it is necessary to use various clever tricks in order to make the methods converge faster. In particular this is true in situations where one needs to estimate something that involves events with very low probabilities. One such area is system reliability estimation. Since failure events often have very low probability, a large number of simulations is needed in order to obtain stable results. Sometimes, however, conditioning can be used to improve the convergence.

6.1 Monte Carlo simulation and conditioning

In the general case the principle of conditioning can be stated as follows: Let $\boldsymbol{X} = (X_1, \ldots, X_n)$ be a random vector with a known distribution, and assume that we want to calculate the expected value of $\phi = \phi(\boldsymbol{X})$. We denote this expectation by h. Assume furthermore that the distribution of ϕ cannot be derived analytically in polynomial time with respect to n. Using Monte Carlo simulations, however, we can proceed by generating a sample of size N, of independent vectors $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$, all having the same distribution as $\boldsymbol{X}$, and then estimate h by the simple *Monte Carlo* estimate:

$$\hat{h}_{MC} = \frac{1}{N} \sum_{r=1}^{N} \phi(\boldsymbol{X}_r). \tag{6.1}$$

The variance of this estimate is given by:

$$\mathrm{Var}\left(\hat{h}_{MC}\right) = \frac{1}{N^2}\sum_{r=1}^{N}\mathrm{Var}(\phi) = \frac{1}{N}\mathrm{Var}(\phi) \tag{6.2}$$

Now, assume that $S = S(\boldsymbol{X})$ is some other function whose distribution can be calculated analytically in polynomial time with respect to n. More specifically, we assume that S is a *discrete* variable with values in the set $\{s_1, \ldots, s_k\}$. We also introduce:

$$\theta_j = E[\phi \mid S = s_j], \quad j = 1, \ldots, k. \tag{6.3}$$

By the formula for conditional expectation we then have:

$$h = \sum_{j=1}^{k} E[\phi \mid S = s_j]P(S = s_j) = \sum_{j=1}^{k} \theta_j P(S = s_j) \tag{6.4}$$

Instead of generating N samples from the distribution of $\boldsymbol{X}$ as in (6.1), we divide the set into k groups, one for each θ_j, where the j-th group has size N_j, $j = 1, \ldots, k$, and $N_1 + \cdots + N_k = N$. The data in the j-th group are sampled from the conditional distribution of $\boldsymbol{X}$ given that $S = s_j$, and are used to estimate the conditional expectation θ_j, $j = 1, \ldots, k$. Denoting the data in the j-th group by $\{\boldsymbol{X}_{r,j} : r = 1, \ldots, N_j\}$, θ_j is estimated by:

$$\hat{\theta}_j = \frac{1}{N_j}\sum_{r=1}^{N_j}\phi(\boldsymbol{X}_{r,j}), \quad j = 1, \ldots, k. \tag{6.5}$$

The variances of these estimates are:

$$\mathrm{Var}(\hat{\theta}_j) = \frac{1}{N_j^2}\sum_{r=1}^{N_j}\mathrm{Var}(\phi|S = s_j) = \frac{1}{N_j}\mathrm{Var}(\phi|S = s_j), \quad j = 1, \ldots, k. \tag{6.6}$$

By combining $\hat{\theta}_1, \ldots, \hat{\theta}_k$, we get the *conditional Monte Carlo estimate*:

$$\hat{h}_{CMC} = \sum_{j=1}^{k}\hat{\theta}_j P(S = s_j). \tag{6.7}$$

Since $\hat{\theta}_1, \ldots, \hat{\theta}_k$ are independent, the variance of the conditional Monte Carlo estimate is:

$$\begin{aligned}\mathrm{Var}(\hat{h}_{CMC}) &= \mathrm{Var}[\sum_{j=1}^{k} \hat{\theta}_j P(S = s_j)] = \sum_{j=1}^{k} \mathrm{Var}(\hat{\theta}_j)[P(S = s_j)]^2 \\ &= \sum_{j=1}^{k} \frac{1}{N_j} \mathrm{Var}(\phi|S = s_j)[P(S = s_j)]^2.\end{aligned}$$

We observe that the variance of the conditional estimate depends on the choices of the N_j's. Ideally we would like N_j to be large if the product of the conditional variance and the squared probability is large, and small otherwise, as this would yield the most efficient partition of the total sample with respect to minimizing the total variance. In practice, however, this is difficult, since the conditional variance is typically not known. In order to compare the result with the variance of the original Monte Carlo estimate, we instead let:

$$N_j \approx N \cdot P(S = s_j), \quad j = 1, \ldots, k,$$

so that:

$$\sum_{j=1}^{k} N_j \approx \sum_{j=1}^{k} N \cdot P(S = s_j) = N \cdot \sum_{j=1}^{k} P(S = s_j) = N.$$

With this choice we get:

$$\begin{aligned}Var(\hat{h}_{CMC}) &\approx \sum_{j=1}^{k} \frac{1}{N \cdot P(S = s_j)} \mathrm{Var}(\phi|S = s_j)[P(S = s_j)]^2 \qquad (6.8) \\ &= \frac{1}{N} \sum_{j=1}^{k} \mathrm{Var}(\phi|S = s_j) P(S = s_j) = \frac{1}{N} E[\mathrm{Var}(\phi|S)] \\ &= \frac{1}{N} \big(\mathrm{Var}(\phi) - \mathrm{Var}[E(\phi|S)] \big) \leq \frac{1}{N} \mathrm{Var}(\phi) = \mathrm{Var}(\hat{h}_{MC}).\end{aligned}$$

From the above equation we see that the conditional estimate has smaller variance than the original Monte Carlo estimate provided that $\mathrm{Var}[E(\phi \mid S)]$ is positive. This quantity can be interpreted as a measure of how much information S contains relative to ϕ. Thus, when looking for good choices

for S we should look for variables containing as much information about ϕ as possible. However, there are some other important points that need to be considered. First of all S must have a distribution that can be derived analytically in polynomial time. Next, the number of possible values of S, i.e., k, must be polynomially limited by n. Finally, it must be possible to sample efficiently from the distribution of $\boldsymbol{X}$ given S.

The problem of sampling from conditional distributions has been studied recently by several authors. Lindqvist and Taraldsen [47] proposes a general method for sampling from conditional distributions in cases where S is a sufficient statistic. Of special relevance to the present paper, is Broström and Nilsson [10] who consider the problem of sampling independent Bernoulli variables given their sum. See also Nilsson [54]. In the remaining part of this paper we shall focus on applications in reliability theory, where $\boldsymbol{X}$ typically is a vector of independent Bernoulli variables. In relation to this we shall derive our own conditional sampling methods.

We now apply the above ideas in the context of system reliability. That is, we assume that $\boldsymbol{X}$ is a vector of independent Bernoulli variables, interpreted as the component state vector relative to a system of n components. A component is *failed* if its state variable is 0, and *functioning* if the state variable is 1. The function ϕ is also assumed to take values 0, 1, and interpreted as the *system state*. If ϕ is 0, the system is *failed*, and if ϕ is 1, the system is *functioning*. The function ϕ is referred to as the structure function of the system, and is assumed to be a nondecreasing function of the component state vector, $\boldsymbol{X}$. In general the calculation of ϕ for a given component state vector depends very much on the representation of this function. If e.g., the system is represented as some sort of communication network its state can usually be evaluated in $O(n)$ time or better. If on the other hand the system is specified in terms of a list of minimal path (or cut) sets, the evaluation can be very slow since the number of such sets in the worst cases grows exponentially with the number of components. For this study, however, we assume that ϕ can be calculated in $O(n)$ time for a given component state vector.

There are in general many different choices for the variable S which can be used in a reliability setting. In the next sections we will consider one possible choice. For other choices see Huseby et al. [34].

6.2 Conditioning on the sum of the component state variables

For the remaining part of the paper we focus on the case where S is the sum of all the component state variables. This approach was taken in in Naustdal [52]. In this situation the random variable S takes values in the set $\{0, 1, \ldots, n\}$. Thus, the uncertain quantities we need to estimate, are:

$$\theta_s = E[\phi|S = s], \qquad s = 0, 1, \ldots, n. \tag{6.9}$$

As in the previous section, we need to have an efficient way of sampling from the conditional distribution of $\boldsymbol{X}$ given $S = s$, $s = 0, 1, \ldots, n$. In relation to this we need the partial sums, $S_1, \ldots, S_n$, defined as:

$$S_m = \sum_{i=m}^{n} X_i, \quad m = 1, \ldots, n.$$

Using these quantities, we can develop the algorithm for sampling from the distribution of $\boldsymbol{X}$ given $S = s$. The idea is simply to start out by sampling X_1 from the conditional distribution of $X_1 \mid S = s$. We then continue by sampling X_2 from $X_2 \mid S = s$, $X_1 = x_1$, where x_1 denotes the sampled outcome of X_1, and so on. This turns out to be easy noting that:

$$\begin{aligned}
&P(X_m = x_m \mid X_1 = x_1, \ldots, X_{m-1} = x_{m-1}, S = s) \\
&\quad = \frac{P(X_m = x_m, S = s \mid X_1 = x_1, \ldots, X_{m-1} = x_{m-1})}{P(S = s \mid X_1 = x_1, \ldots, X_{m-1} = x_{m-1})} \\
&\quad = \frac{P(X_m = x_m, S_{m+1} = s - \sum_{j=1}^{m} x_j)}{P(S_m = s - \sum_{j=1}^{m-1} x_j)} \\
&\quad = \frac{p_m^{x_m}(1 - p_m)^{1-x_m} P(S_{m+1} = s - \sum_{j=1}^{m} x_j)}{P(S_m = s - \sum_{j=1}^{m-1} x_j)},
\end{aligned}$$

Assuming that the distributions of $S_1, \ldots, S_n$ are all calculated before running the simulations, we see that all the necessary conditional probabilities can be calculated when needed during the simulations without imposing additional computational complexity. Note that we only calculate those conditional probabilities we really need along the way during the simulations,

not the entire set of all possible conditional probabilities corresponding to all possible combinations of values of the X_j's. Thus, in each simulation run we calculate n probabilities, one for each X_j. Moreover, each probability can be calculated using a fixed number of operations (independent of n). Hence, it follows that sampling from the conditional distribution of $\boldsymbol{X}$ given $S = s$, can be done in $O(n)$ time.

The CMC-algorithm can now be summarized as follows: Divide the N simulations into $(n + 1)$ groups, one for each θ_s, where the s-th group has size N_s, $s = 0, 1, \ldots, n$. The data in the s-th group is sampled from the conditional distribution of $\boldsymbol{X}$ given that $S = s$, and is used to estimate the conditional expectation θ_s, $s = 0, 1, \ldots, n$. Denoting the data in the s-th group by $\{\boldsymbol{X}_r, s : r = 1, \ldots, N_s\}$, θ_s is estimated essentially by using (6.5), and combined into the CMC-estimate by using (6.7).

A remaining problem is of course how to choose the sample sizes for each of the $(n+1)$ groups. One possibility is to proceed as we did in Section 1 and let $N_s \approx N \cdot P(S = s)$, $s = 0, 1, \ldots, n$. In this case, however, it is possible to improve the results slightly. As in the previous section we denote the size of the smallest path set by d, and the size of the smallest cut set by c. By examining the system, it is often easy to determine d and c. Given these two numbers it is easy to see that $\theta_s = 0$ for $s < d$, and $\theta_s = 1$ for $s > n - c$. Moreover, $\mathrm{Var}(\phi \mid S = s) = 0$ for $s < d$, or $s > n - c$. Hence, there is no point in spending simulations on estimating θ_s for $s < d$, or $s > n - c$, so we let $N_s = 0$ for $s < d$, or $s > n - c$. As a result, we have more simulations to spend on the remaining quantities.

An extreme situation occurs when the system is a k-out-of-n-system, i.e., when $\phi(\boldsymbol{X}) = I(S \geq k)$. For such systems $d = k$, and $c = (n-k+1)$. In this situation *all* the θ_s's are known. Specifically, $\theta_s = 0$ for $s < k$ and $\theta_s = 1$ for $s \geq k$. Thus, the CMC-estimate is equal to the true value of the reliability, and can be calculated without doing any simulations at all. The reason for this is of course that in this case ϕ depends on $\boldsymbol{X}$ only through S.

For all other nontrivial systems, however, it is easy to see that we always have that $d \leq n - c$. In order to ensure that we get improved results, we assume that $P(d \leq S \leq n - c) > 0$, and let:

$$N_s \approx N \cdot P(S = s)/P(d \leq S \leq n - c), \quad s = d, \ldots, n - c\,. \tag{6.10}$$

Using a similar argument as we did in (6.8) we now get that:

$$\mathrm{Var}(\hat{h}_{CMC}) \leq P(d \leq S \leq n - c)\,\mathrm{Var}(\hat{h}_{MC}) \tag{6.11}$$

Hence, we see that if d and $(n-c)$ are close, i.e., there are few unknown θ_s's, the variance is reduced considerably.

6.3 System reliability when all the component state variables have identical reliabilities

If all the components in the system have the same reliability, i.e., $p_1 = \cdots = p_n = p$, it is possible to improve things even further. In this case we note that S has a binomial distribution. Moreover, the conditional distribution of $\boldsymbol{X}$ given S is given by:

$$P(\boldsymbol{X} = \boldsymbol{x} \mid S = s) = \frac{p^{\sum_{i=1}^n x_i}(1-p)^{n-\sum_{i=1}^n x_i}}{\binom{n}{s} p^s (1-p)^{n-s}} = \frac{1}{\binom{n}{s}} , \tag{6.12}$$

for all $\boldsymbol{x}$ such that $\sum_{i=1}^n x_i = s$. From this it follows that:

$$\theta_s = E[\phi \mid S = s] = \sum_{\{\boldsymbol{x} : \sum_{i=1}^n x_i = s\}} \phi(\boldsymbol{x}) P(\boldsymbol{X} = \boldsymbol{x} \mid S = s) = \frac{b_s}{\binom{n}{s}} , \tag{6.13}$$

where b_s = the number of path sets with s components, $s = 0, \ldots, n$. Finally, the system reliability, h, expressed as a function of p, is given by:

$$\begin{aligned} h(p) &= \sum_{s=0}^n \theta_s P(S = s) \\ &= \sum_{s=0}^n \frac{b_s}{\binom{n}{s}} \binom{n}{s} p^s (1-p)^{n-s} = \sum_{s=0}^n b_s p^s (1-p)^{n-s}. \end{aligned}$$

Note that the unknown quantities, $\theta_0, \ldots, \theta_n$, do not depend on p. Thus, by estimating these quantities, we get an estimate of the entire $h(p)$-function for all $p \in [0, 1]$. Moreover, we see that θ_s can be interpreted as the fraction of path sets of size s among all sets of size s, for $s = 0, 1, \ldots, n$. Thus, θ_s can be estimated by sampling random sets of size s and calculating the frequency of path sets among the sampled sets. It turns out that this can be done very efficiently as follows:

The idea is to sample sequences of sets of increasing size by sampling from the component set, $C = \{1, \ldots, n\}$, without replacements. The only set of

size 0 is of course $\emptyset$. If $\emptyset$ is a path set, we have a trivial system which is always functioning. Obviously, $\theta_0 = 1$ in this case. Moreover, the reliability of such a system is 1. If $\emptyset$ is not a path set, it follows that $\theta_0 = 0$. In both cases we do not need to sample anything to determine the value of θ_0. Thus, we can focus on estimating $\theta_1, \ldots, \theta_n$ by sampling sets of size s, for $s = 1, \ldots, n$.

Algorithm 6.3.1 *Let $\boldsymbol{T} = (T_1, \ldots, T_n)$ be a vector for storing results from the simulations. Before we start the simulations, this vector is initialised as $(0, \ldots, 0)$. Then for $i = 1, \ldots, N$ do:*

1. *Sample a component from the set C, say component i_1, and define $A_1 = \{i_1\}$. If A_1 is a path set, T_1 is incremented with 1.*

2. *Sample a component from the set $C \setminus A_1$, say component i_2, and define $A_2 = \{i_1, i_2\}$. If A_2 is a path set, T_2 is incremented with 1.*

   ...

n. *Sample the last remaining component, say component i_n, and define $A_n = C$. If A_n is a path set, T_n is incremented with 1.*

When all the simulations are carried out, the vector T contains the number of observed path sets of sizes $1, \ldots, n$. From this we get the resulting estimates of $\theta_1, \ldots, \theta_n$ simply as:

$$\theta_s = T_s/N, \quad s = 1, \ldots, n.$$

It is easy to see that sampling components randomly without replacements is equivalent to sampling the components according to a random permutation, $(i_1, \ldots, i_n)$, of the component set $\{1, \ldots, n\}$. Such a permutation can easily be generated using a well-known algorithm described in Knuth [44]. This algorithm works as follows:

Assume that we start out with a vector representing an initial arbitrary permutation of the components, say $(i_1, \ldots, i_n)$. One may e.g., simply use $(1, \ldots, n)$. We then generate a random permutation vector by modifying this initial permutation vector by running through n steps. In the k-th step we consider the element currently being the k-th element of the vector. We then sample a random index, j, in the interval $k, \ldots, n$, and let the k-th and the j-th elements switch places in the permutation vector. When all the n steps

are completed, it is easy to see that the resulting vector is indeed a random permutation, regardless of the initial state of the vector.

By using this algorithm, we are able to generate a complete sequence of n sets in just $O(n)$ time. However, since all sets in a sequence are derived from the same permutation, the θ_s-estimates become correlated. Still, the effect of this is more than compensated for since each θ_s-estimate now can be based on all N simulations, compared to just a subset of size N_s as was the case in the previous section.

When running this algorithm, much of the time is spent on the steps where the system state is evaluated. Thus, in order to minimize the running time of the algorithm, one would like to reduce the number of such evaluations. Since $A_1 \subset A_2 \subset \cdots \subset A_n$, it follows by the monotonicity of the structure function, ϕ, that if A_k is a path set, then so is A_{k+1}. Thus, we can stop evaluating the state of the system as soon as we have generated a path set, since we then know that all the remaining sets will be path sets as well. Still it is obviously important to carry out the system state evaluations as efficient as possible. The methodology for doing this may typically depend strongly on the representation of the given system.

In order to show how this can be done, we consider two different classes of systems. The first class we consider is the class of *threshold systems* (see Section 4.5), i.e., binary monotone systems (C, ϕ) where the structure function can be written in the following form:

$$\phi(\boldsymbol{X}) = I\Big(\sum_{i=1}^{n} a_i X_i \geq b\Big), \tag{6.14}$$

where $a_1, \ldots, a_n$ and b are positive numbers. The a_i's are called the *weights* associated with the components, while the number b is called the *threshold value*. Notice that if $a_1 = \cdots = a_n = 1$ and $b = k$, we get a standard k-out-of-n-system. For such systems it is very easy to carry out all the system state evaluations. To do so we keep track of the sum of the weights associated with the sampled components. When a new component is sampled, the sum is updated by adding the weight associated with this component. When the sum of weights is greater than or equal to the threshold value, we know that we have generated a path set. Using this method, each system evaluation is carried out in constant time. Thus, in this case the total computational complexity of the simulation algorithm is just $O(nN)$, where n is the number of components in the system, and N is the number of simulations.

The second class is the class of undirected network systems (see Section 4.7). The components of such a system are the edges of an undirected network. An edge is functioning if its *end-nodes* can "communicate" through it. The system is functioning if a subset of the nodes (with K elements), called the *terminals*, can communicate through the network.

If all the edges are functioning, we assume that the network is connected, i.e., all nodes can communicate with each other. If only a subset of the edges are functioning, however, the node set is partitioned into a set of equivalence classes such that a pair of nodes can communicate if and only if they belong to the same equivalence class. Moreover, the system is functioning if and only if all the terminals belong to the same equivalence class.

In order to minimize the running time spent on system state evaluation, we maintain lists of nodes belonging to each equivalence class as we sample the edges. For each list we also keep track of the number of terminals contained in the list. When a new edge is sampled, we investigate its end-nodes. If they belong to the same equivalence class, the system state is not changed. On the other hand, if the end-nodes belong to different equivalence classes, we merge the two classes and calculate the number of terminals included in the merged class. When we arrive at an equivalence class containing K terminals, (i.e., all the terminals), we know that we have generated a path set.

The most time-consuming part of this algorithm is the merging of equivalence classes. Assuming that the classes are stored as linked lists, the actual merging of the lists can be done in constant time. However, in order to minimize the time spent on checking whether or not two nodes belong to the same class, the nodes should be marked with a reference to its class list. When two classes, say A and B, are merged, all the nodes in the smallest class, say B must be marked with a reference to the class list of A instead. Updating such references can be done in $O(v)$ time, where v is the number of nodes in the network. It should be noted, however, that this is a very pessimistic estimate, as in most cases the number of updating operations is much smaller. Anyway, the total computational complexity of the simulation algorithm is $O(nvN)$ in this case.

We close this section by illustrating the effect of the improved method on a specific example. The system we consider is the 2-terminal undirected network system illustrated in Figure 6.1. We assume that all the 7 components have the same reliability p, and estimate the system reliability $h(p)$ for all $p \in [0, 1]$. Moreover, we let $N = 100$. With so few iterations it is easier to see

how much better the conditional Monte Carlo method is compared to crude Monte Carlo method.

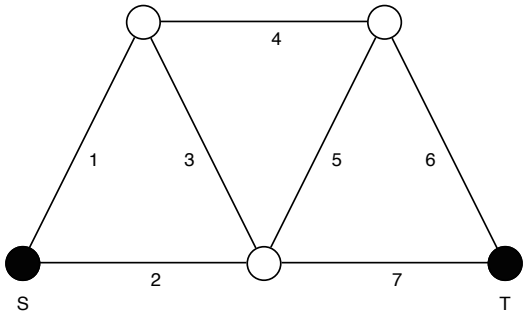

Figure 6.1: A 2-terminal undirected network system

For the conditional Monte Carlo method we know that we get an estimate of the entire $h(p)$-function by estimating the fractions $\theta_1, \ldots, \theta_n$. For the crude Monte Carlo estimate one possible approach is to proceed by obtaining estimates of $h(p)$ for each individual value of p. This would imply that we would need $N = 100$ iterations for each chosen value of p. Thus, with a suitable resolution of p-values, say $p \in \{0.01, 0.02, \ldots, 1.00\}$ we need $100N = 10000$ samples. Fortunately, it is possible to avoid this issue. In order to do so, the following sampling method can be used. In the jth iteration we generate uniformly distributed random variables $U_{1j}, \ldots, U_{7j}$. For a given value of $p \in [0, 1]$ we then define:

$$X_{ij}(p) = I(U_{ij} \leq p), \quad i = 1, \ldots, 7, \quad j = 1, \ldots N,$$

where $I(U_{ij} \leq p) = 1$ if $U_{ij} \leq p$ and zero otherwise. This implies that:

$$P(X_{ij}(p) = 1) = P(U_{ij} \leq p) = p, \quad i = 1, \ldots, 7, \quad j = 1, \ldots N.$$

We denote the structure function of the system by ϕ. Moreover, we denote the jth sampled component state vector, expressed as a function of the common component reliability p, by $\boldsymbol{X}_j(p) = (X_{1,j}(p), \ldots, X_{7,j}(p))$, $j = 1, \ldots, N$. We can then estimate $h(p)$ as follows:

$$\hat{h}(p) = \frac{1}{N} \sum_{j=1}^{N} \phi(\boldsymbol{X}_j(p)).$$

By varying p in the interval $[1, 0]$, we obtain an estimate of $h(p)$ for all p. Hence, we can use the *same* sample of uniformly distributed variables to obtain an estimate of the system reliability for each value of p. Thus, we only need $N = 100$ samples for this method as well, which saves us a lot of time. Moreover, it is easy to see that $\boldsymbol{X}_j(p)$ is a non-decreasing function of p for all j. Hence, since ϕ is a non-decreasing function of $\boldsymbol{X}_j$, the estimated curve, $\hat{h}(p)$, is non-decreasing as well, a property which the true function $h(p)$ also has.

Using a program called *CMCsim*[1] we can run the conditional Monte Carlo simulations. This program has an intuitive graphical user interface, and can be used to estimate reliability and component importance of any undirected network system.

In Figure 6.2 we have plotted the two estimates for $h(p)$ along with the true reliability function. The red curve is the crude Monte Carlo estimate, the green curve is the conditional Monte Carlo curve, while the blue curve represents the true reliability function. As we can see, the green and the blue curves are almost identical. Thus, even with as few as $N = 100$ iterations, we get a very precise estimate of the entire $h(p)$-function. The crude Monte Carlo estimate is far more irregular and deviates considerably from the true curve.

[1]CMCsim is a java program developed at the Department of Mathematics, University of Oslo. The program is freely available at http://www.riscue.org/cmcsim/.

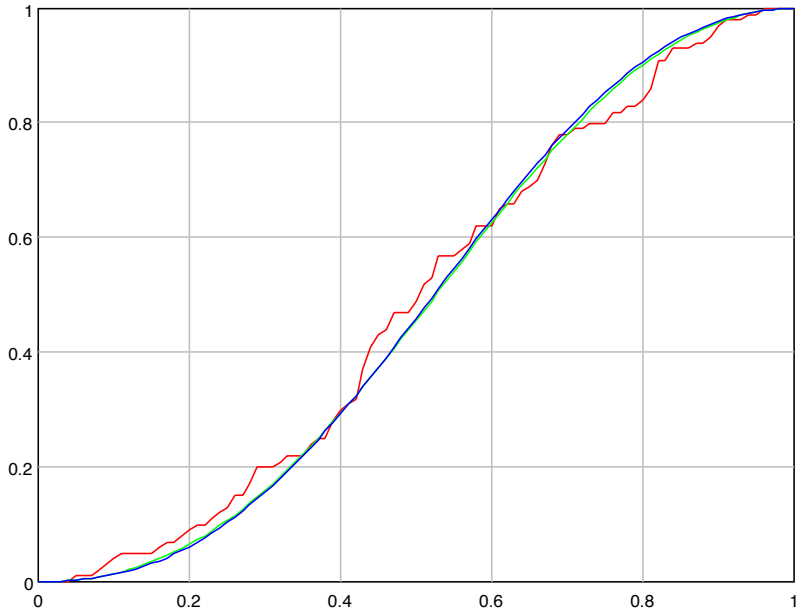

Figure 6.2: System reliability as a function of the common component reliability p. Crude Monte Carlo estimate (red curve), conditional Monte Carlo estimate (green curve) and true reliability (blue curve).

Chapter 7

Dynamic System Analysis

So far, we have only considered binary monotone systems at a specific point of time. In this chapter we will show how to analyse systems where the states of the components, and hence also the state of the system, evolve over time. Throughout this chapter we consider systems where we assume that the component state processes are *independent* of each other.

In Section 7.1 we start out by considering systems with components that *cannot* be repaired. In Section 7.3 and 7.4 we introduce two new measures of reliability importance, and discuss how these measures are related to the Birnbaum measure introduced in Chapter 5.

In Section 7.5 we discuss the concept of *pure jump processes*. Such processes serve as a framework for analysing systems with repairable components which will be handled in Section 7.6. The stationary properties of such systems can typically be analysed by using the stationary properties of the component processes. Thus, the reliability calculation methods are more or less identical to the methods we have covered in the previous chapters. In order to study the probabilistic properties more in detail, however, it is often necessary to use Monte Carlo simulation. Such simulations need to handle that a collection of stochastic processes, one for each component, operating in parallel. This can be done using what is called *discrete event simulations*. This technique, as well as some advanced estimation methods are described in Section 7.7 and 7.8.

7.1 Non-repairable binary monotone systems

As before we consider a binary monotone system (C, ϕ), where $C = \{1, \ldots, n\}$. In order to model systems with components that evolve over time we introduce:

$$X_i(t) = \text{The state of component } i \text{ at time } t, \quad t \geq 0, \quad i \in C \tag{7.1}$$

For any given $t \geq 0$, $X_i(t)$ is a random variable. The development of the state of component i over time, $\{X_i(t)\} = \{X_i(t) : t \geq 0\}$, is a stochastic process. As mentioned in the introduction to this chapter, we assume that the stochastic processes $\{X_1(t)\}, \ldots, \{X_n(t)\}$ are *independent* of each other. We also let:

$$\phi(\boldsymbol{X}(t)) = \text{The state of the system at time } t, \quad t \geq 0. \tag{7.2}$$

As for the components, $\phi(\boldsymbol{X}(t))$ is a random variable, and the development of the state of the system over time, $\{\phi(\boldsymbol{X}(t))\} = \{\phi(\boldsymbol{X}(t)) : t \geq 0\}$ is a stochastic process.

Time-dependent versions of component and system reliabilities are defined as follows for $t \geq 0$ and $i \in C$:

$$\begin{aligned} p_i(t) &= P(X_i(t) = 1) = \text{The reliability of component } i \text{ at time } t, \\ h(\boldsymbol{p}(t)) &= P(\phi(\boldsymbol{X}(t)) = 1) = \text{The reliability of the system at time } t. \end{aligned} \tag{7.3}$$

Since at this stage we do not consider the possibility of repairing components, these quantities are closely related to the lifetimes of the system and the components. To show this we let:

$$\begin{aligned} T_i &= \text{The lifetime of component } i, \text{ for } i \in C, \\ S &= \text{The lifetime of the system.} \end{aligned} \tag{7.4}$$

Now, let T_i have cumulative distribution function F_i, $i = 1, \ldots, n$ and S cumulative distribution function G. We then have the following relations:

$$p_i(t) = P(X_i(t) = 1) = P(T_i > t) = 1 - F_i(t) = \bar{F}_i(t), \tag{7.5}$$

$$h(t) = P(\phi(\boldsymbol{X}(t)) = 1) = P(S > t) = 1 - G(t) = \bar{G}(t). \tag{7.6}$$

and

$$\bar{G}(t) = h(\boldsymbol{p}(t)) = h(\bar{F}_1(t), \ldots, \bar{F}_n(t)). \tag{7.7}$$

The function $\bar{F}_i(t)$ is referred to as the *survival function* of the component lifetime T_i, $i = 1, \ldots, n$, while $\bar{G}$ is referred to as the *survival function* of the system lifetime S.

Let $\bar{F}$ be the survival function of the lifetime T of some component, and assume that $\bar{F}(t) > 0$ for all $t \geq 0$. Then the survival function can be written in the following form:

$$\bar{F}(t) = e^{-R(t)}, \quad t \geq 0, \tag{7.8}$$

where $R(t) = -\ln(\bar{F}(t))$. The function R is referred to as the *cumulative failure rate function* of T. Moreover, the derivative of $R(t)$, denoted $r(t)$, is called the *failure rate function* of T. The failure rate can be expressed in terms of the density and survival function as follows:

$$r(t) = \frac{\partial}{\partial t}(-\ln(\bar{F}(t))) = \frac{f(t)}{\bar{F}(t)}, \quad t \geq 0.$$

Thus, $r(t)dt$ can be interpreted as the conditional probability that the component fails in the interval $(t, t + dt]$, given that the component survived the interval $[0, t]$. Note that the survival function can be expressed in terms of the failure rate function as follows:

$$\bar{F}(t) = e^{-\int_0^t r(u)du}, \quad t \geq 0, \tag{7.9}$$

Thus, the lifetime distribution is uniquely determined by its failure rate function. By studying the failure rate of a lifetime distribution, many important features can be derived. In particular, if r is *increasing*, we say that the lifetime has an *increasing failure rate*. Similarly, if r is decreasing, we say that the lifetime has a *decreasing failure rate*. A lifetime model with increasing failure rate is typically used to describe *aging*. As the component gets older, it becomes more prone to failures. A decreasing failure rate can be used to describe a period of *infant mortality*. As the component gets older, early failures are eliminated or corrected.

7.1.1 The Weibull distribution

A commonly used lifetime distribution in reliability is the *Weibull distribution*[1]. This distribution will be used in many of the examples in this section,

[1]The Weibull distribution is named after the Swedish mathematician Waloddi Weibull, who introduced the distribution in 1951

so we include a brief introduction to this distribution here. If T is Weibull distributed with parameters $\alpha > 0$ and $\beta > 0$, the probability density of T is given by

$$f(t) = \frac{\alpha}{\beta}\left(\frac{t}{\beta}\right)^{\alpha-1} e^{-(t/\beta)^{\alpha}}, \quad t \geq 0. \tag{7.10}$$

The parameter α is called the *shape parameter*, while the β is called the *scale parameter*.

From the density it is easy to show that the cumulative distribution function of a Weibull distribution is given by:

$$F(t) = P(T \leq t) = \int_0^t f(u)du = 1 - e^{-(t/\beta)^{\alpha}}, \quad t \geq 0, \tag{7.11}$$

while the survival function is:

$$\bar{F}(t) = P(T > t) = 1 - F(t) = e^{-(t/\beta)^{\alpha}}, \quad t \geq 0. \tag{7.12}$$

The cumulative failure rate of a Weibull distribution is given by:

$$R(t) = -\ln(\bar{F}(t)) = \left(\frac{t}{\beta}\right)^{\alpha}, \quad t \geq 0, \tag{7.13}$$

while the failure rate is given by:

$$r(t) = \frac{f(t)}{\bar{F}(t)} = \frac{\alpha}{\beta}\left(\frac{t}{\beta}\right)^{\alpha-1} \tag{7.14}$$

By (7.14) it follows that the failure rate is decreasing when $0 < \alpha < 1$, and increasing when $\alpha > 1$. If $\alpha = 1$, the density simplifies to:

$$f(t) = \frac{1}{\beta}e^{-t/\beta}, \quad t \geq 0.$$

which is the density of an exponential distribution. When $\alpha = 1$, the cumulative failure rate is a linear function:

$$R(t) = -\ln(\bar{F}(t)) = \frac{t}{\beta}, \quad t \geq 0,$$

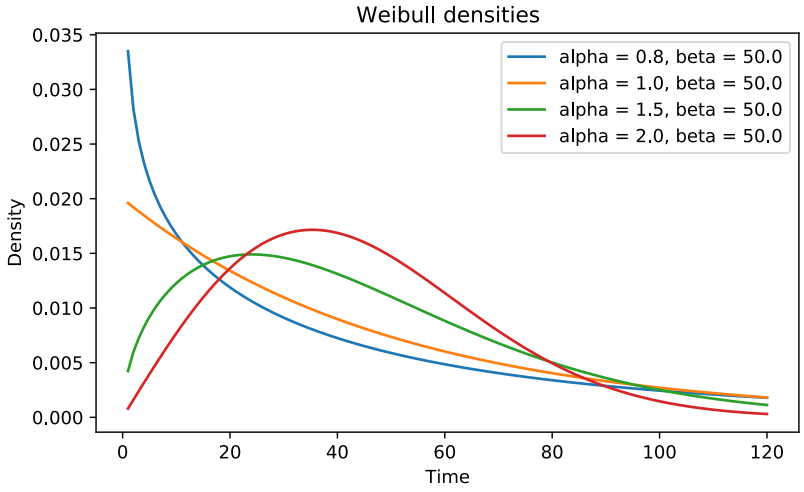

Figure 7.1: Weibull densities for different shape parameters.

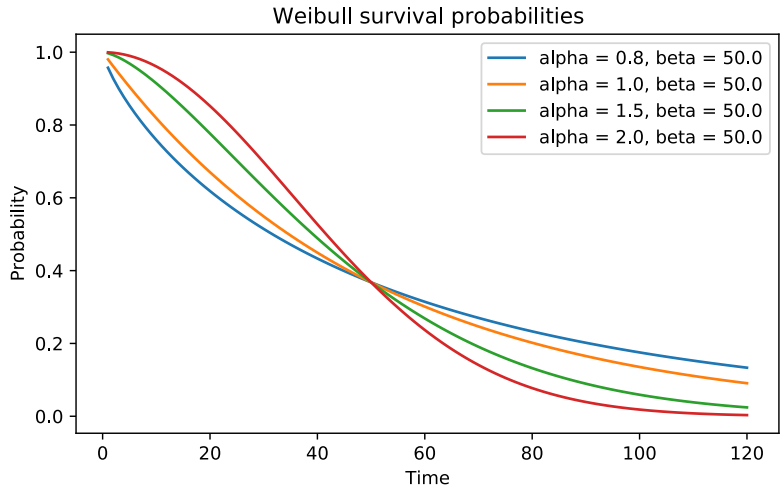

Figure 7.2: Weibull survival probabilities for different shape parameters.

while the failure rate is constant:

$$r(t) = \frac{f(t)}{\bar{F}(t)} = \frac{1}{\beta}$$

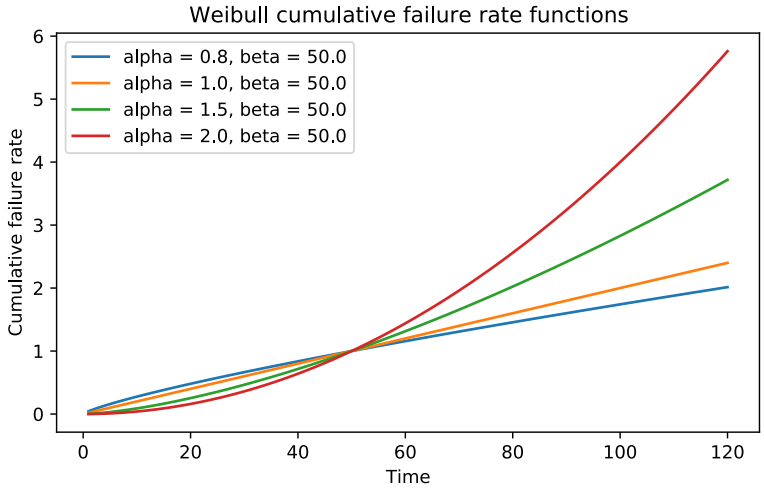

Figure 7.3: Weibull cumulative failure rates for different shape parameters.

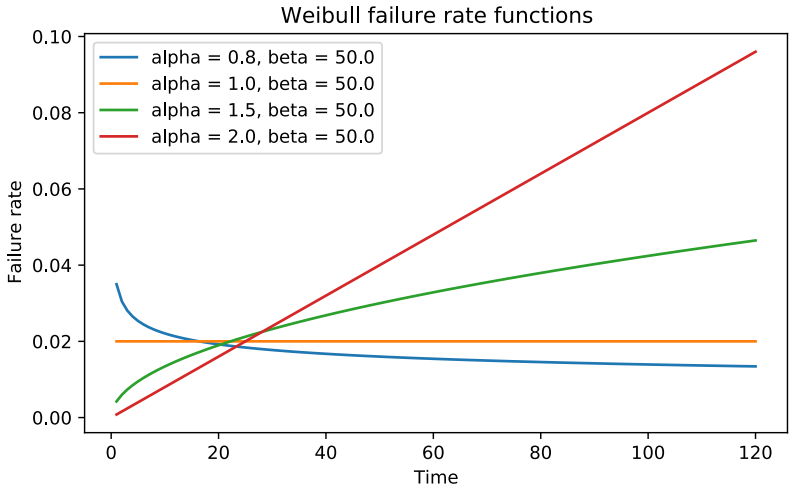

Figure 7.4: Weibull failure rates for different shape parameters.

Thus, the class of Weibull distributions includes both lifetimes with decreasing failure rates, increasing failure rate and constant failure rates. This makes this distribution class very flexible.

In Figure 7.1 we have plotted densities of Weibull distributions with α-values 0.8, 1.0, 1.5 and 2.0 respectively, and $\beta = 50$. The corresponding survival functions are plotted in Figure 7.2, while the cumulative failure rates and failure rates are plotted in Figure 7.3 and Figure 7.4.

By (7.12) and (7.13) it follows that:

$$\begin{aligned}\bar{F}(\beta) &= e^{-(\beta/\beta)^\alpha} = e^{-1},\\ R(\beta) &= 1\end{aligned}$$

Hence, since all the four chosen Weibull distributions have the same scale parameter, it follows that all the survival functions go through the point (β, e^{-1}), while all the cumulative failure rate functions go through the point $(\beta, 1)$.

If T is Weibull distributed with parameters $\alpha > 0$ and $\beta > 0$, then we have:

$$E[T] = \beta \cdot \Gamma(\frac{1}{\alpha} + 1), \tag{7.15}$$

$$Med[T] = \beta \cdot [\ln(2)]^{1/\alpha} \tag{7.16}$$

The proof is left as an exercise.

We close this section by presenting some important properties of Weibull distributions. For the first property, we assume that T is Weibull distributed with parameters α and β, and that $K > 0$ is a constant. Then the survival function of $U = KT$ is given by:

$$P(U > u) = P(KT > u) = P(T > u/K) = e^{-[(u/K)/\beta]^\alpha} = e^{-[u/(K\beta)]^\alpha}$$

We recognize this as the survival function of a Weibull distributed variable with parameters α and $K\beta$.

For the second property we assume that $T_1, \ldots, T_n$ are independent random variables, and that T_i is Weibull distributed with parameters α and β_i, $i = 1, \ldots, n$. We then let:

$$T = \min(T_1, \ldots, T_n)$$

The survival function of T is then given by:

$$P(T > t) = P(T_1 > t \cap \cdots \cap T_n > t) = \prod_{i=1}^{n} P(T_i > t)$$

$$= \prod_{i=1}^{n} e^{-(t/\beta_i)^\alpha} = \exp(-\sum_{i=1}^{n} (t/\beta_i)^\alpha)$$

$$= \exp(-t^\alpha \sum_{i=1}^{n} \beta_i^{-\alpha}) = e^{-(t/\gamma)^\alpha}$$

where $\gamma^{-\alpha} = \sum_{i=1}^{n} \beta_i^{-\alpha}$, or equivalently $\gamma = (\sum_{i=1}^{n} \beta_i^{-\alpha})^{-1/\alpha}$. This implies that T is Weibull distributed as well with parameters α and γ.

In the special case where $\beta_1 = \cdots = \beta_n = \beta$, we get that $\gamma = (n\beta^{-\alpha})^{-1/\alpha} = n^{-1/\alpha}\beta$. Thus, by combining the two above properties we get that $n^{1/\alpha}T$ is Weibull distributed with parameters α and β. Note that this property holds for any n, and thus, also when $n \to \infty$. This is linked to the fact that the Weibull distribution is a so-called *extreme value distribution*.

The following theorem summarizes the above results:

Theorem 7.1.1 *a) If T is a Weibull distributed random variable with parameters α and β, and $K > 0$ is a constant, then KT is Weibull distributed with parameters α and $K\beta$.*

b) If $T_1, \ldots, T_n$ are independent random variables, and T_i is Weilbull distributed with parameters α and β_i, $i = 1, \ldots, n$, then $\min(T_1, \ldots, T_n)$ is Weilbull distributed with parameters α and $\gamma = (\sum_{i=1}^{n} \beta_i^{-\alpha})^{-1/\alpha}$.

c) If $T_1, \ldots, T_n$ are independent and identically Weibull distributed random variables with parameters α and β, then $n^{1/\alpha} \min(T_1, \ldots, T_n)$ is Weilbull distributed with parameters α and β.

7.1.2 Plotting system reliability as a function of time

We now return to the calculation of the system reliability as a function of the time. In order to plot $h(\boldsymbol{p}(t))$ as a function of the time $t \geq 0$, we choose a sufficiently dense set of t-values, say $0 = t_0 < t_1 < \cdots < t_K$, and calculate $h(\boldsymbol{p}(t_j))$, for these t-values. The reliability curve for $t \in [0, t_K]$ is then obtained by drawing a curve through the points:

$$(t_0, h(\boldsymbol{p}(t_0))),\ (t_1, h(\boldsymbol{p}(t_1))),\ \ldots,\ (t_K, h(\boldsymbol{p}(t_K))).$$

If K is sufficiently large, a nice smooth curve is obtained.

Example 7.1.2 *Let (C, ϕ) be a 2-out-of-3 system with $C = \{1, 2, 3\}$ and structure function ϕ is given by:*

$$\phi(X_1, X_2, X_3) = X_1X_2 + X_1X_3 + X_2X_3 - 2X_1X_2X_3$$

From this it follows that the reliability of the system at time $t \geq 0$ is:

$$h(\boldsymbol{p}(t)) = p_1(t)p_2(t) + p_1(t)p_3(t) + p_2(t)p_3(t) - 2p_1(t)p_2(t)p_3(t)$$

We then assume that the lifetime of component i, T_i is Weibull-distributed with parameters α_i and β_i, for all $i \in C$. By inserting $p_i(t) = \bar{F}_i(t)$, $i = 1, 2, 3$

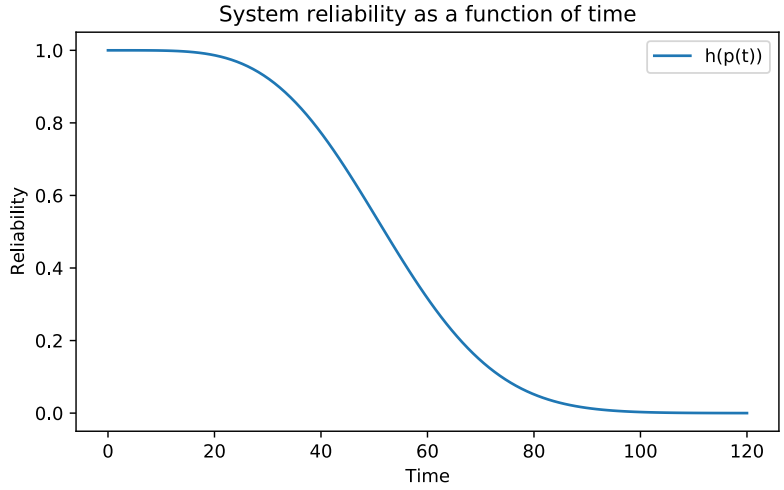

Figure 7.5: The reliability of a 2-out-of-3 system as a function of the time $t \geq 0$.

into the expression for $h(\boldsymbol{p}(t))$, we can calculate the system reliability for any $t \geq 0$. Assume more specifically that:

$$\alpha_1 = 2.0, \quad \alpha_2 = 2.5, \quad \alpha_3 = 3.0,$$
$$\beta_1 = 50.0, \quad \beta_2 = 60.0, \quad \beta_3 = 70.0.$$

The resulting reliability curve is plotted in Figure 7.5. A python script which can be used to produce this plot, is given in Script B.3.1 in Appendix B.3.

Given the minimal path sets $P_1, \ldots, P_p$ and the minimal cut sets $K_1, \ldots, K_k$ of the system, we can also express the relation between the component lifetimes, $T_1, \ldots, T_n$ and the system lifetime, S, as follows:

$$S = \min_{1 \le j \le k} \max_{i \in K_j} T_i = \max_{1 \le j \le p} \min_{i \in P_j} T_i. \tag{7.17}$$

The proof of this relation is left as an exercise. Note, however, the similarity between (7.17) and the relation (3.11) between the state of the system, ϕ, and the component state variables $X_1, \ldots, X_n$.

The formula (7.17) is useful when the system reliability is estimated using Monte Carlo simulation. In order to explain how this can be done, we consider a binary monotone system (C, ϕ), where $C = \{1, \ldots, n\}$, and where $P_i, \ldots, P_p$ are the *minimal path sets* of the system. We then generate N samples of component lifetimes $\{T_{1k}, \ldots, T_{nk}\}$, $k = 1, \ldots, N$, where T_{ik} denotes the lifetime of the ith component in the kth simulation. In each simulation we calculate the resulting system lifetime using (7.17). That is, we calculate system lifetimes $S_1, \ldots, S_N$ given by:

$$S_k = \max_{1 \le j \le p} \min_{i \in P_j} T_{ik}, \quad k = 1, \ldots, N.$$

Based on these lifetimes we then estimate the reliability of the system as a function of the time $t \ge 0$ using the following formula:

$$\hat{h}(t) = \frac{1}{N} \sum_{k=1}^{N} I(S_k > t)$$

where $I(\cdot)$ denotes the indicator function. It follows that:

$$E[\hat{h}(t)] = \frac{1}{N} \sum_{k=1}^{N} P(S_k > t) = \bar{G}(t) = h(\boldsymbol{p}(t))$$

Thus, $\hat{h}(t)$ is an unbiased estimator for the reliability of the system at time $t \ge 0$.

In order to calculate $\hat{h}(t)$ more efficiently for all $t \ge 0$, we sort the system lifetimes in ascending order:

$$S_{(1)} < \cdots < S_{(N)},$$

assuming for simplicity that all the lifetimes are distinct. We also define $S_{(0)} = 0$, and note that:

$$\hat{h}(S_{(m)}) = \frac{1}{N}\sum_{k=1}^{N} I(S_k > S_{(m)}) = \frac{N-m}{N}, \quad m = 0, 1, \ldots, N.$$

Hence, the function $\hat{h}(t)$ passes through the following $N+1$ points:

$$(S_{(0)}, \frac{N-0}{N}), (S_{(1)}, \frac{N-1}{N}), \ldots, (S_{(N)}, \frac{N-N}{N}).$$

For sufficiently large values of N a nice smooth estimate of the function $h(\boldsymbol{p}(t))$ can then be obtained by drawing a curve through these points.

Example 7.1.3 *Let (C, ϕ) be a binary monotone system with component set $C = \{1, 2, 3, 4, 5\}$ and minimal path sets $P_1 = \{1, 4\}$, $P_2 = \{1, 3, 5\}$, $P_3 = \{2, 3, 4\}$, $P_4 = \{2, 5\}$. The lifetime of component i, T_i is Weibull-distributed with parameters α_i and β_i, for all $i \in C$ where:*

$$\alpha_1 = 2.0, \quad \alpha_2 = 2.5, \quad \alpha_3 = 3.0, \quad \alpha_4 = 1.0, \quad \alpha_5 = 1.5,$$
$$\beta_1 = 50.0, \quad \beta_2 = 60.0, \quad \beta_3 = 70.0, \quad \beta_4 = 40.0, \quad \beta_5 = 50.0.$$

In order to estimate the reliability $h(\boldsymbol{p}(t))$ for $t \geq 0$, we run a Monte Carlo simulation with $N = 100000$ simulations. The resulting reliability curve is plotted in Figure 7.6. A python script which can be used to produce this plot, is given in Script B.3.2 in Appendix B.3.

In cases where the minimal path or cut sets have not been identified, it may still be possible to estimate the reliability function provided that we can calculate the structure function efficiently. The trick here is to take advantage of the fact that the lifetime of the system *always* coincides with the lifetime of one of the components.

To explain this in detail we consider a binary monotone system (C, ϕ), where $C = \{1, \ldots, n\}$. As above, the lifetime of the ith component is T_i, for all $i \in C$. By sorting the lifetimes we can identify the order at which the components fail. For simplicity we assume that all the component lifetimes are distinct. If the lifetimes are absolutely continuously distributed, this will happen with probability one anyway. If $T_{i_1} < \cdots < T_{i_n}$ are the sorted lifetimes, this means that the components fail in the order $i_1, \ldots, i_n$. For

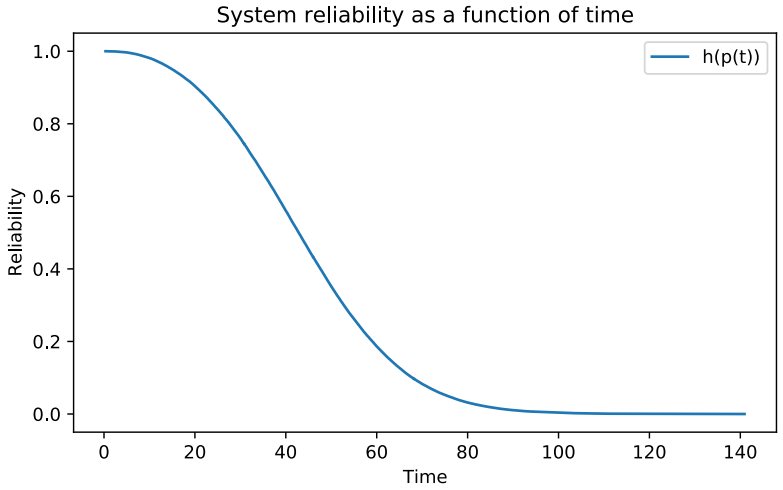

Figure 7.6: The reliability of a bridge system as a function of the time $t \geq 0$.

convenience we also define $T_{i_0} = 0$. We then let $\boldsymbol{X}_j$ denote the component state vector after the jth failure, $j = 1, \ldots, n$, and also let $\boldsymbol{X}_0 = \mathbf{1}$. Finally, we let:

$$j^* = \min\{j : \phi(\boldsymbol{X}_j) = 0\}$$

Then it follows that the lifetime of the system, S, is given by:

$$S = T_{i_{j^*}} \tag{7.18}$$

Note that in the trivial case where $\phi(\boldsymbol{X}_0) = \phi(\mathbf{1}) = 0$, we get that $j^* = 0$. Thus, $S = T_{i_0} = 0$ in such cases, which indeed is the correct result. In the other trivial case where $\phi(\boldsymbol{X}_n) = \phi(\mathbf{0}) = 1$, j^* becomes undefined. To avoid this issue we simply define S to be ∞ in such cases, which is the correct result.

Having found a method for calculating the system lifetime without the use of minimal path or cut sets, we can proceed as before and obtain an estimate of the system reliability as a function of time.

Example 7.1.4 *Let (C, ϕ) be a threshold system with component set $C = \{1, 2, 3, 4, 5\}$, and where the structure function is given by:*

$$\phi(\boldsymbol{X}) = I(\sum_{i=1}^{5} \omega_i X_i \geq 5.0)$$

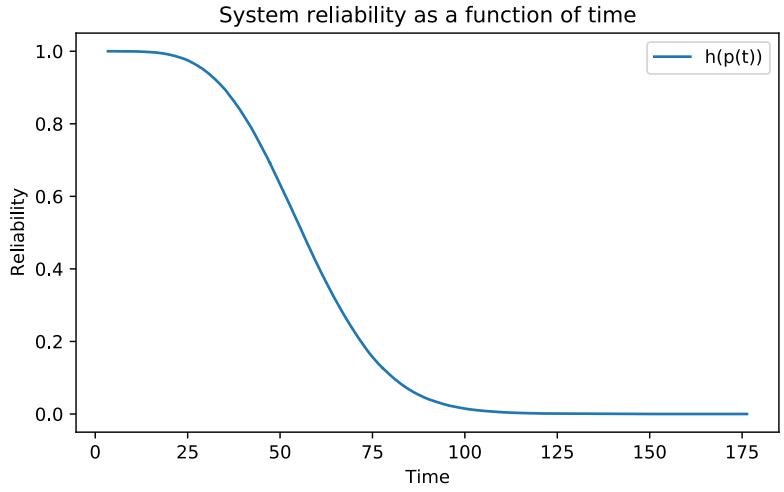

Figure 7.7: The reliability of a threshold system as a function of the time $t \geq 0$.

where the component weights $\omega_1, \ldots, \omega_5$ *are:*

$$\omega_1 = 2.0, \quad \omega_2 = 4.0, \quad \omega_3 = 2.3, \quad \omega_4 = 3.5, \quad \omega_5 = 1.9$$

The lifetime of component i, T_i *is Weibull-distributed with parameters* α_i *and* β_i, *for all* $i \in C$ *where:*

$$\begin{aligned} &\alpha_1 = 2.0, \quad \alpha_2 = 2.5, \quad \alpha_3 = 3.0, \quad \alpha_4 = 1.0, \quad \alpha_5 = 1.5, \\ &\beta_1 = 50.0, \quad \beta_2 = 60.0, \quad \beta_3 = 70.0, \quad \beta_4 = 40.0, \quad \beta_5 = 50.0. \end{aligned}$$

In order to estimate the reliability $h(\boldsymbol{p}(t))$ *for* $t \geq 0$, *we run a Monte Carlo simulation with* $N = 100000$ *simulations. The resulting reliability curve is plotted in Figure 7.7. A python script which can be used to produce this plot, is given in Script B.3.3 in Appendix B.3.*

Example 7.1.5 *Let* (C, ϕ) *be an undirected network system with component set* $C = \{1, \ldots, 7\}$, *and where the network is illustrated in Figure 4.4. The structure function of this system is given by:*

$$\phi(\boldsymbol{X}) = X_4 \cdot \phi(1_4, \boldsymbol{X}) + (1 - X_4) \cdot \phi(0_4, \boldsymbol{X}),$$

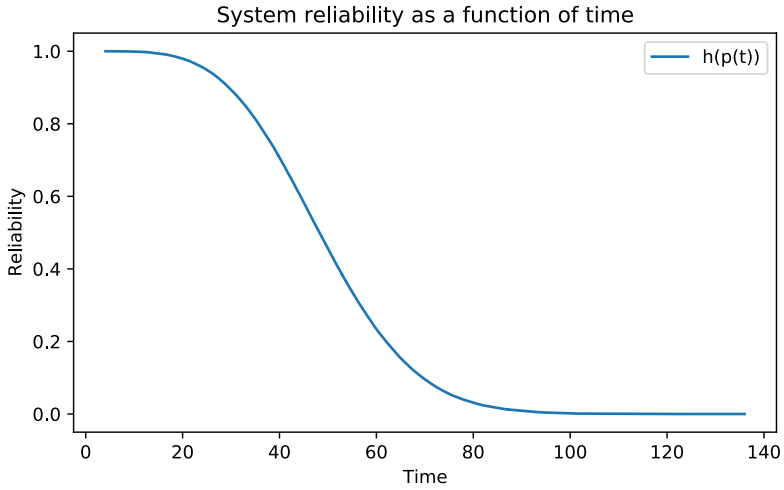

Figure 7.8: The reliability of an undirected network system as a function of the time $t \geq 0$.

where:

$$\begin{aligned}\phi(1_4, \boldsymbol{X}) &= (X_3 \amalg X_5) \cdot (X_1 \amalg X_2) \cdot (X_6 \amalg X_7) \\ &\quad + (1 - X_3 \amalg X_5) \cdot ((X_1 X_6) \amalg (X_2 X_7))\end{aligned}$$

$$\phi(0_4, \boldsymbol{X}) = ((X_1 X_3) \amalg X_2) \cdot ((X_5 X_6) \amalg X_7)$$

The lifetime of component i, T_i is Weibull-distributed with parameters α_i and β_i, for all $i \in C$ where:

$$\begin{aligned}&\alpha_1 = 2.0, \quad \alpha_2 = 2.5, \quad \alpha_3 = 3.0, \quad \alpha_4 = 1.0, \\ &\alpha_5 = 1.5, \quad \alpha_6 = 2.0, \quad \alpha_7 = 2.5,\end{aligned}$$

$$\begin{aligned}&\beta_1 = 50.0, \quad \beta_2 = 60.0, \quad \beta_3 = 70.0, \quad \beta_4 = 40.0, \\ &\beta_5 = 50.0, \quad \beta_6 = 55.0, \quad \beta_7 = 65.0.\end{aligned}$$

The reliability $h(\boldsymbol{p}(t))$ for $t \geq 0$ is estimated by running a Monte Carlo simulation with $N = 100000$ simulations. The resulting reliability curve is plotted in Figure 7.8. A python script which can be used to produce this plot, is given in Script B.3.5 in Appendix B.3.

7.2 The time-dependent Birnbaum measure of reliability importance

In this section we introduce the time-dependent version of the Birnbaum importance measure. Using the same notation as in the previous section this measure is defined as follows:

$$I_B^{(i)}(t) = h(1_i, \boldsymbol{p}(t)) - h(0_i, \boldsymbol{p}(t)), \quad t \geq 0, \quad i \in C. \tag{7.19}$$

To obtain a plot of $I_B^{(i)}(t)$ we use exactly the same procedure as for the reliability function. That is, we choose a sufficiently dense set of t-values, say $0 = t_0 < t_1 < \cdots < t_K$, and calculate $I_B^{(i)}(t)$, for these t-values. The importance curve for $t \in [0, t_K]$ is then obtained by drawing a curve through the points:

$$(t_0, I_B^{(i)}(t_0)),\ (t_1, I_B^{(i)}(t_1)),\ \ldots,\ (t_K, I_B^{(i)}(t_K)).$$

If K is sufficiently large, a nice smooth curve is obtained.

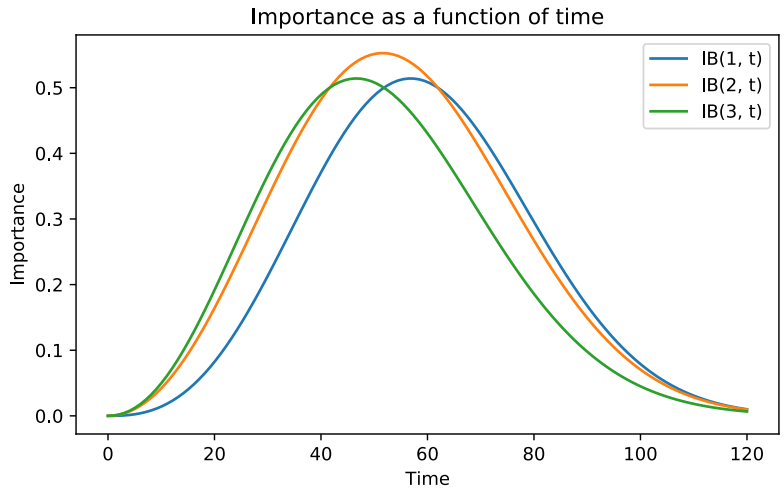

Figure 7.9: The Birnbaum importance of the components of a 2-out-of-3 system as a function of the time $t \geq 0$.

Example 7.2.1 *We consider the 2-out-of-3 system introduced in Example 7.1.2. By inserting $p_i(t) = \bar{F}_i(t)$, $i = 1, 2, 3$ into the expression for $I_B^{(i)}(t)$, we can calculate the importance of the three components for any $t \geq 0$. The resulting importance curves are plotted in Figure 7.9. A python script which can be used to produce this plot, is given in Script B.3.6 in Appendix B.3.*

The time-dependent Birnbaum measure can also be estimated using Monte Carlo simulation. However, this is a bit more challenging than estimating the system reliability as a function of time. As above our goal is to estimate the importance curves for $t \in [0, t_K]$. Thus, we choose a sufficiently dense set of t-values, say $0 = t_0 < t_1 < \cdots < t_K$, and estimate $I_B^{(i)}(t)$, for these t-values. The estimated importance values for the ith component are denoted $\hat{I}_B^{(i)}(t_0), \hat{I}_B^{(i)}(t_1), \ldots, \hat{I}_B^{(i)}(t_K)$, $i \in C$. The resulting estimated importance curve for $t \in [0, t_K]$ for the ith component is then obtained by drawing a curve through the points:

$$(t_0, \hat{I}_B^{(i)}(t_0)),\ (t_1, \hat{I}_B^{(i)}(t_1)),\ \ldots,\ (t_K, \hat{I}_B^{(i)}(t_K)).$$

As we did for the reliability estimation, we start out by generating N samples of component lifetimes $\{T_{1k}, \ldots, T_{nk}\}$, $k = 1, \ldots, N$, where T_{ik} denotes the lifetime of the ith component in the kth simulation. Furthermore, we assume that we are able to calculate the system lifetime as a function of the component lifetimes, either based on the minimal path or cut sets by using (7.17), or by using the formula (7.18).

We now consider the component lifetimes generated in the kth simulation, i.e., $\{T_{1k}, \ldots, T_{nk}\}$, and focus on component i. For component i we calculate two system lifetimes, L_{ik} and U_{ik}, where L_{ik} is obtained by replacing T_{ik} by 0, while U_{ik} is obtained by replacing T_{ik} by ∞. Thus, L_{ik} is the lifetime of the system given that component i is failed all the time, while U_{ik} is the lifetime of the system given that component i is functioning all the time.

We then consider a point of time $t \in [0, t_K]$. If $t < L_{ik}$, we know that the system is *functioning* regardless of the state of component i. Similarly, if $t \geq U_{ik}$, we know that the system is *failed* regardless of the state of component i. However, if $L_{ik} \leq t < U_{ik}$, the system is functioning if component i is functioning, and failed if component i is failed. Thus, we conclude that in the kth simulation component i is *critical* for the system if and only if $L_{ik} \leq t < U_{ik}$.

In each of the simulations we keep track of the points of time when component i is critical, and introduce the following statistics for $i \in C$ and

$j = 0, 1, \ldots, K$:

$$D_{ij} = \text{The number of times component } i \text{ is critical at time } t_j.$$

Based on these statistics we obtain the following unbiased estimates for the importance of component i at the points of time $t_0, t_1, \ldots, t_K$:

$$\hat{I}_B^{(i)}(t_j) = \frac{D_{ij}}{N}, \quad j = 0, 1, \ldots, K, \quad i \in C. \tag{7.20}$$

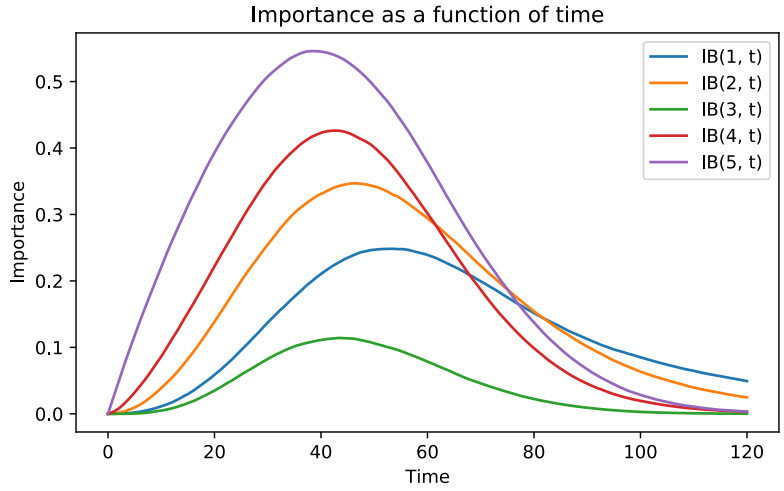

Figure 7.10: The Birnbaum importance of the components of a bridge system as a function of the time $t \geq 0$. See Example 7.1.3.

Example 7.2.2 *We consider the bridge system introduced in Example 7.1.3. By inserting $p_i(t) = \bar{F}_i(t)$, $i = 1, \ldots, 5$ into the expression for $I_B^{(i)}(t)$, we can calculate the importance of the five components for any $t \geq 0$. The resulting importance curves are plotted in Figure 7.10. A python script which can be used to produce this plot, is given in Script B.3.7 in Appendix B.3.*

Example 7.2.3 *We consider the threshold system introduced in Example 7.1.4. By inserting $p_i(t) = \bar{F}_i(t)$, $i = 1, \ldots, 5$ into the expression for $I_B^{(i)}(t)$, we can calculate the importance of the five components for any $t \geq 0$. The resulting importance curves are plotted in Figure 7.11. A python script which can be used to produce this plot, is given in Script B.3.8 in Appendix B.3.*

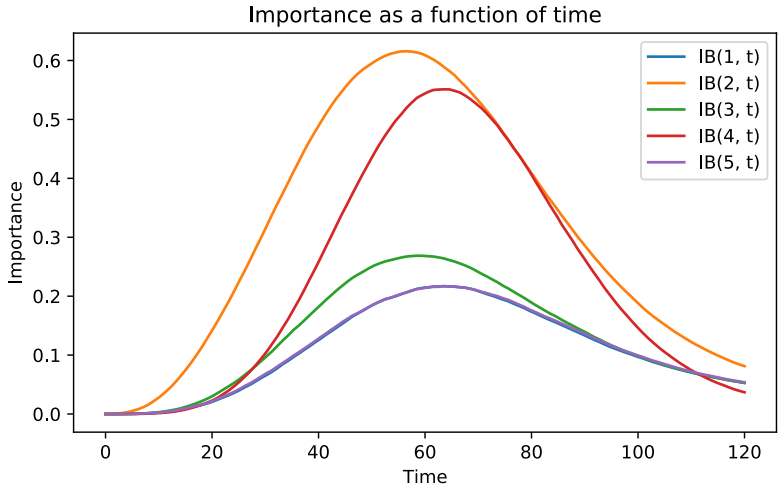

Figure 7.11: The Birnbaum importance of the components of a threshold system as a function of the time $t \geq 0$. See Example 7.1.4.

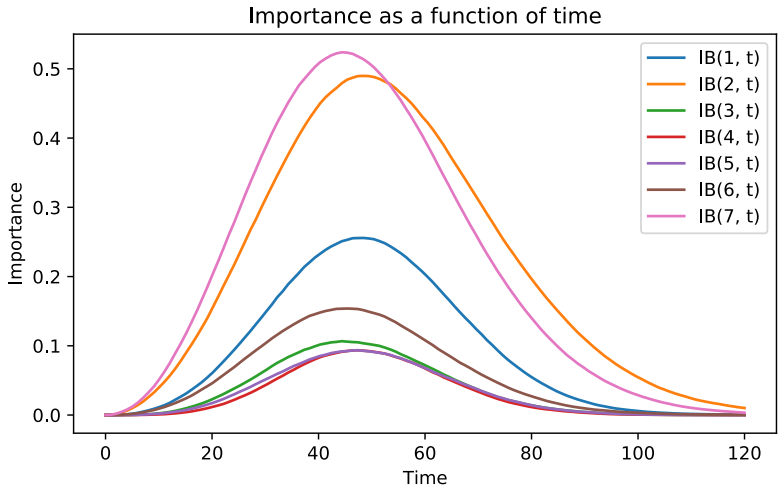

Figure 7.12: The Birnbaum importance of the components of an undirected network system as a function of the time $t \geq 0$. See Example 7.1.5.

Example 7.2.4 *We consider the undirected network system introduced in Example 7.1.5. By inserting* $p_i(t) = \bar{F}_i(t)$, $i = 1, \ldots, 7$ *into the expression for* $I_B^{(i)}(t)$, *we can calculate the importance of the seven components for any* $t \geq 0$. *The resulting importance curves are plotted in Figure 7.12. A python script which can be used to produce this plot, is given in Script B.3.10 in Appendix B.3.*

7.3 The Barlow-Proschan measure of reliability importance

One disadvantage with the time dependent Birnbaum measure for the reliability importance is that the ranking of the reliability importance of the components may change over time. This implies that in order to compare the components with respect to importance, one must decide which point of time that should be used for the comparison. Motivated by this issue Barlow and Proschan [5] introduced the following *time-independent* importance measure:

Definition 7.3.1 *Consider a non-repairable binary monotone system* (C, ϕ). *Moreover, let* T_i *denote the lifetime of component* i, $i \in C$, *and let* S *denote the lifetime of the system. Then for all* $i \in C$ *the* Barlow-Proschan measure of the reliability importance *of component* i *is defined as:*

$$\begin{aligned} I_{B-P}^{(i)} &= P(\textit{Component } i \textit{ fails at the same time as the system}) \\ &= P(T_i = S) \end{aligned} \tag{7.21}$$

Note that if component i fails at the same time as the system, this may be interpreted as i is the component eventually causing the system to fail.

Before we proceed we need to discuss the concept of *absolute continuity*. We say that a real-valued stochastic variable, T has an absolutely continuous distribution if $P(T \in A) = 0$ for all measurable sets $A \subseteq \mathbb{R}$ such that $m_1(A) = 0$, where m_1 denotes the Lebesgue measure[2] in $\mathbb{R}$. It can be shown

[2]The Lebesgue measure in $\mathbb{R}$, denoted m_1, is a function defined on the family of all measurable subsets A of $\mathbb{R}$. If $A = [a, b]$, then $m_1(A) = b - a$. Thus, the Lebesgue measure m_1 generalizes the length of intervals to arbitrary measurable subsets of $\mathbb{R}$. Similarly, the Lebesgue measure in $\mathbb{R}^n$, denoted m_n, is a function defined on the family of all measurable subsets A of $\mathbb{R}^n$. If $A = [a_1, b_1] \times \cdots \times [a_n, b_n]$, then $m_n(A) = \prod_{i=1}^n (b_n - a_n)$. Thus, the Lebesgue measure m_n generalizes the volume of n-dimensional rectangular sets to arbitrary measurable subsets of $\mathbb{R}^n$.

that if $T_1, \ldots, T_n$ are independent and absolutely continuously distributed, then the vector $\boldsymbol{T} = (T_1, \ldots, T_n)$ is absolutely continuously distributed in $\mathbb{R}^n$. That is, $P(\boldsymbol{T} \in A) = 0$ for all (measurable) sets $A \subseteq \mathbb{R}^n$ such that $m_n(A) = 0$, where m_n denotes the Lebesgue measure in $\mathbb{R}^n$. In particular, if $A = \{\boldsymbol{t} : t_i = t_j\}$, where $i \neq j$, then $m_n(A) = 0$. Hence, $P(T_i = T_j) = 0$ when $i \neq j$.

The following theorem uses the concept of absolute continuity in order to obtain some fundamental properties of the Barlow-Proschan measure.

Theorem 7.3.2 *Let (C, ϕ) be a non-trivial binary monotone system where the components are never repaired where $C = \{1, \ldots, n\}$, and let T_i denote the lifetime of component i, $i = 1, \ldots, n$, while S denotes the lifetime of the system. Moreover, assume that $T_1, \ldots, T_n$ are independent, absolutely continuously distributed. Then S is absolutely continuously distributed as well, and we have:*

$$\sum_{i=1}^{n} I_{B-P}^{(i)} = 1.$$

Proof: Since we have assumed that the system is non-trivial, we know by (7.17) that the lifetime of the system, S can be expressed as:

$$S = \max_{1 \leq j \leq p} \min_{i \in P_j} T_i, \tag{7.22}$$

where $P_1, \ldots, P_p$ are the minimal path sets of the system. This implies that:

$$P(\bigcup_{i=1}^{n} \{T_i = S\}) = 1. \tag{7.23}$$

Let $A \subseteq \mathbb{R}$ be an arbitrary measurable set such that $m_1(A) = 0$. Since we have assumed that $T_1, \ldots, T_n$ are absolutely continuously distributed, we get that:

$$0 \leq P(S \in A) \leq P(\bigcup_{i=1}^{n} \{T_i \in A\}) \leq \sum_{i=1}^{n} P(T_i \in A) = 0,$$

which implies that S is absolutely continuously distributed as well.

Since $T_1, \ldots, T_n$ are absolutely continuously distributed, it follows that the probability of having two or more components failing at the same time is zero. This implies e.g., that $P(\{T_i = S\} \cap \{T_j = S\}) = 0$ for $i \neq j$. Thus, when calculating the probability of the union of the events $\{T_i = S\}$,

$i = 1, \ldots, n$, all intersections can be ignored as they have zero probability of occurring. Hence, by (7.23) we get:

$$1 = P(\bigcup_{i=1}^{n}\{T_i = S\}) = \sum_{i=1}^{n} P(T_i = S) = \sum_{i=1}^{n} I_{B-P}^{(i)},$$

where the second equality follows by ignoring all intersections of the events $\{T_i = S\}$, $i = 1, \ldots, n$. The last equality follows by the definition of $I_{B-P}^{(i)}$. Hence, the proof is complete. □

The next result shows how the Barlow-Proschan measure is related to the Birnbaum measure.

Theorem 7.3.3 *Let (C, ϕ) be a non-trivial binary monotone system where the components are never repaired where $C = \{1, \ldots, n\}$. Moreover, assume that the lifetimes of the components, $T_1, \ldots, T_n$, are independent and absolutely continuously distributed with densities $f_1, \ldots, f_n$ respectively. Then, we have:*

$$I_{B-P}^{(i)} = \int_0^\infty I_B^{(i)}(t) \cdot f_i(t)dt, \quad i \in C. \tag{7.24}$$

where $I_B^{(i)}(t)$ denotes the Birnbaum measure of the reliability importance of component i at time t, i.e., the probability that component i is critical at time t.

Proof: From the definitions of the Barlow-Proschan measure and the Birnbaum measure, it follows that:

$$\begin{aligned} I_{B-P}^{(i)} &= P(\text{Component } i \text{ fails at the same time as the system}) \\ &= \int_0^\infty P(\text{Component } i \text{ is critical at time } t) \cdot f_i(t)dt \\ &= \int_0^\infty I_B^{(i)}(t) \cdot f_i(t)dt. \end{aligned}$$

□

We observe that according to Theorem 7.3.3 the Barlow-Proschan measure, $I_{B-P}^{(i)}$, can be expressed as a weighted average of the Birnbaum measure

$I_B^{(i)}(t)$ with respect to the component lifetime densities, $f_i(t)$. Hence, according to the Barlow-Proschan measure, the points of times where $f_i(t)$ is large are viewed as important.

In order to compute the Barlow-Proschan measure the formula (7.24) is not always easy to use. For this purpose it is actually better to use the original definition with the formula (7.21). Using Monte Carlo simulation it is easy to estimate the probability that a given component fails at the same time as the system. By running N simulations we simply count the number of times this happens for each of the components. If W_i is the number of times component i fails at the same time as the system, $i \in C$, then the Barlow-Proschan measure of importance can be estimated by the following unbiased estimator:

$$\hat{I}_{B-P}^{(i)} = \frac{W_i}{N}, \quad i \in C. \tag{7.25}$$

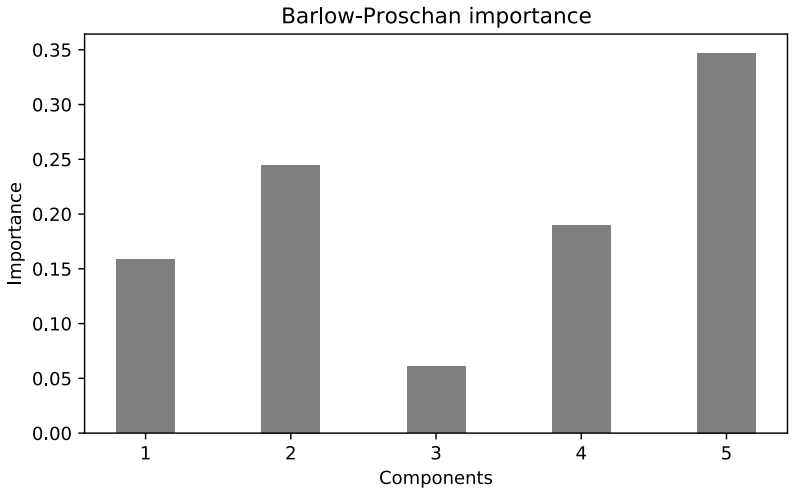

Figure 7.13: The Barlow-Proschan importance measure for the components in a bridge system.

Example 7.3.4 *We consider the bridge system introduced in Example 7.1.3. To estimate* $I_{B-P}^{(i)}$ *for* $i \in C$*, we run a Monte Carlo simulation with* $N =$

100000 *simulations and use* (7.25). *The resulting estimated importance measures are plotted in Figure 7.13. A python script which can be used to produce this plot, is given in Script B.3.11 in Appendix B.3.*

From Figure 7.13 we see that the bridge component, i.e., component 3, *clearly is the least important component in the system. The probability that this component fails at the same time as the rest of the system is just about* 5%.

In a bridge system all the other components have the same structural importance. However, as can be seen in Figure 7.13, component 5 *is the component that has the highest probability of failing at the same time as the system, i.e., approximately* 35%.

To understand why this may be reasonable, we calculate the expected lifetimes for the different components using (7.15)*:*

$$E[T_1] = 44.31, \quad E[T_2] = 55.09, \quad E[T_3] = 62.51,$$
$$E[T_4] = 40.00, \quad E[T_5] = 45.14$$

The minimal cut sets of the system are $K_1 = \{1, 2\}$, $K_2 = \{1, 3, 5\}$, $K_3 = \{2, 3, 4\}$ *and* $K_4 = \{4, 5\}$. *The minimal cut set which is most likely to cause system failure is* K_4, *since it contains only two components, and since these components have relatively short expected lifetimes compared to the components in* K_1. *Comparing the expected lifetimes of components* 4 *and* 5, *we see that component* 5 *has the longest expected lifetime of the two. Hence, component* 5 *is more likely to be the last one in* K_4 *to fail and thus, cause system failure.*

Example 7.3.5 *We consider the threshold system introduced in Example 7.1.4. To estimate* $I^{(i)}_{B-P}$ *for* $i \in C$, *we run a Monte Carlo simulation with* $N = 100000$ *simulations. The resulting estimated importance measures are plotted in Figure 7.14. A python script which can be used to produce this plot, is given in Script B.3.12 in Appendix B.3.*

Example 7.3.6 *We consider the undirected network system introduced in Example 7.1.5. To estimate* $I^{(i)}_{B-P}$ *for* $i \in C$, *we run a Monte Carlo simulation with* $N = 100000$ *simulations. The resulting estimated importance measures are plotted in Figure 7.15. A python script which can be used to produce this plot, is given in Script B.3.13 in Appendix B.3.*

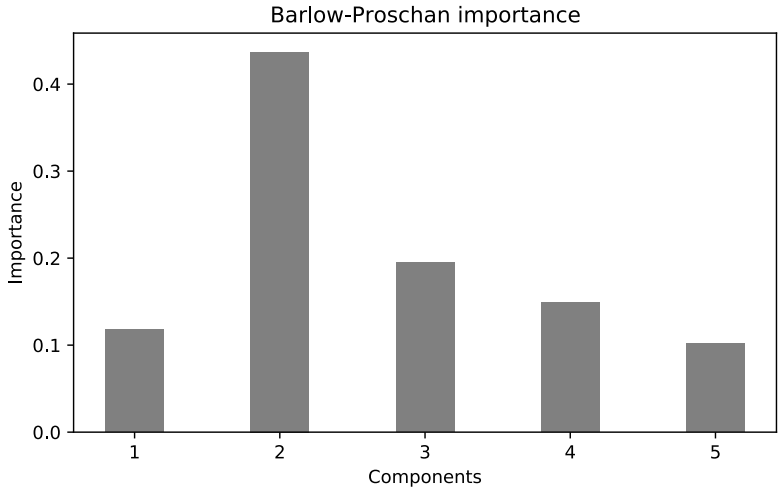

Figure 7.14: The Barlow-Proschan importance measure for the components in a threshold system.

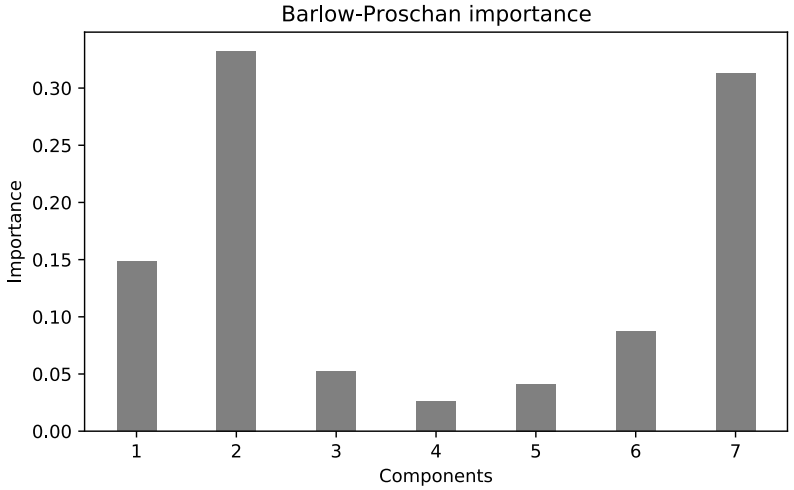

Figure 7.15: The Barlow-Proschan importance measure for the components in an undirected network system.

7.4 The Natvig measure of reliability importance

The Barlow-Proschan measure of reliability importance focusses on the component which has the highest probability to cause system failure. However, sometimes a component may contribute significantly to the reliability of the system even though it is not the last component to fail. Motivated by this, Natvig [49] introduced an alternative importance measure. This measure is based on the concept of a *minimal repair*.

To explain this concept we let T denote the lifetime of some component. The survival distribution function of T is denoted $\bar{F}(t) = P(T > t)$. Now, we assume that we have observed that $T = \tau$. That is, at time $\tau \geq 0$, the component failed. We then let $t \geq \tau$. If the component had *not failed* at time τ, the conditional probability that it would have survived time t as well would have been:

$$P(T > t|T > \tau) = \frac{\bar{F}(t)}{\bar{F}(\tau)}, \quad t \geq \tau$$

A *minimal repair* of the component at time τ, means that the component is given an additional lifetime U such that the total lifetime of the component, i.e., $\tau + U$, has the same distribution as the conditional distribution of T given that the component did *not fail* at time τ. That is, we have:

$$P(\tau + U > t) = P(U > t - \tau) = \frac{\bar{F}(t)}{\bar{F}(\tau)}, \quad t \geq \tau. \tag{7.26}$$

By substituting $u = t-\tau$ and $t = u+\tau$, it follows that (7.26) can be expressed as:

$$P(U > u) = \frac{\bar{F}(u + \tau)}{\bar{F}(\tau)}, \quad u \geq 0. \tag{7.27}$$

Intuitively, we may think of a minimal repair as having an extra component as a *hot stand-by*. A stand-by component is a component identical to the original component that can replace the original as soon as this fails. A *cold* stand-by component does not age while in stand-by mode, and can be considered as good as new when it replaces the original. A *hot* stand-by component, on the other hand, cannot fail when it is in stand-by mode,

but it ages at the same speed as the original component. Thus, when it replaces the original component, it has a survival function which is equal to the conditional survival function of the original component immediately before its failure. This is precisely the relation given in (7.27).

The total lifetime of the component including the lifetime of the hot standby component, is $T + U$. The resulting survival function of the component then becomes $P(T + U > t)$. In order to calculate this probability, we split the event $\{T + U > t\}$ into two disjoint parts:

$$A = \{T + U > t\} \cap \{T > t\} \text{ and } B = \{T + U > t\} \cap \{T \leq t\}$$

Since the event $\{T > t\}$ implies the event $\{T + U > t\}$, it follows that:

$$P(A) = \bar{F}(t)$$

To compute $P(B)$ we condition on the point τ when the original component fails. Given this point of time, it follows that U must be greater than $t - \tau$ for B to hold. Thus, by (7.26) we get:

$$P(B) = \int_0^t P(U > t - \tau)\, f(\tau) d\tau = \bar{F}(t) \int_0^t \frac{f(\tau)}{\bar{F}(\tau)} d\tau$$

By substituting $w = \bar{F}(\tau)$ and $dw = -f(\tau)$, the integral becomes:

$$P(B) \;=\; -\bar{F}(t) \int_1^{\bar{F}(t)} \frac{dw}{w} \;=\; -\bar{F}(t) \ln(\bar{F}(t))$$

Thus, the survival function of component which is given a minimal repair, is given by:

$$P(T + U > t) = P(A) + P(B) = \bar{F}(t)[1 - \ln(\bar{F}(t))] \tag{7.28}$$

The idea behind the Natvig measure of reliability importance is that the component that by failing on average has the largest negative impact on the system lifetime, should be considered most important. This idea can also be interpreted in terms of minimal repairs: The component that by being given a minimal repair on average has the largest positive impact on the system lifetime, should be considered most important. More specifically, the Natvig measure of reliability importance is defined as follows:

Definition 7.4.1 *Consider a non-repairable binary monotone structure* (C, ϕ)*, where* $C = \{1, \ldots, n\}$*, and let:*

$S_i =$ *The system lifetime if component* i *is given a minimal repair.*
$Z_i = S_i - S$

Assuming that $E[Z_i] < \infty$ *for all* $i \in C$*, the* Natvig measure *of the reliability importance of component* i*, denoted* $I_N^{(i)}$*, is given by:*

$$I_N^{(i)} = \frac{E[Z_i]}{\sum_{j=1}^{n} E[Z_j]} \tag{7.29}$$

We note that by this definition it immediately follows that $0 \leq I_N^{(i)} \leq 1$ and that $\sum_{i=1}^{n} I_N^{(i)} = 1$. Thus, the result given in Theorem 7.3.2 is valid for the Natvig measure as well.

To study the properties of this measure in more detail, we need the following well-known lemma:

Lemma 7.4.2 *Let* X *be a non-negative absolutely continuously distributed random variable with density* $f(x)$*. Then the expected value of* X *is given by:*

$$E[X] = \int_0^\infty P(X > y)\, dy. \tag{7.30}$$

Proof: *We start out by noting that:*

$$E[X] = \int_0^\infty x f(x) dx = \int_0^\infty [\int_0^x dy\,]\, f(x)\, dx$$

We then interchange the integrals and get:

$$E[X] = \int_0^\infty [\int_y^\infty f(x) dx\,]\, dy = \int_0^\infty P(X > y)\, dy$$

□

We then consider the system lifetimes S and S_i introduced above. By Lemma 7.4.2 and pivotal decomposition with respect to component i we get

that:

$$E[S] = \int_0^\infty h(\boldsymbol{p}(t))dt \tag{7.31}$$
$$= \int_0^\infty \{\bar{F}_i(t)\, h(1_i, \boldsymbol{p}(t)) + (1 - \bar{F}_i(t))\, h(0_i, \boldsymbol{p}(t))\}dt$$

and using (7.28):

$$E[S_i] = \int_0^\infty h(\boldsymbol{p}(t))dt \tag{7.32}$$
$$= \int_0^\infty \{\bar{F}_i(t)[1 - \ln(\bar{F}_i(t))]\, h(1_i, \boldsymbol{p}(t)) + (1 - \bar{F}_i(t)[1 - \ln(\bar{F}_i(t))])\, h(0_i, \boldsymbol{p}(t))\}dt$$

We recall that by Theorem 7.3.3 the Barlow-Proschan measure could be expressed as a weighted average of the Birnbaum measure. Natvig [50] showed that there is a similar connection between the Natvig measure and the Birnbaum measure:

Theorem 7.4.3 *Consider a binary monotone structure (C, ϕ), where $C = \{1, \ldots, n\}$. Moreover, assume that the lifetimes of the components, $T_1, \ldots, T_n$, are independent and absolutely continuously distributed. Then we have:*

$$E[Z_i] = \int_0^\infty I_B^{(i)}(t) \cdot \bar{F}_i(t)(-\ln(\bar{F}_i(t)))dt, \quad i \in C. \tag{7.33}$$

Proof: The result follows directly by combining (7.31) and (7.32) and noting that by (7.19):

$$I_B^{(i)}(t) = h(1_i, \boldsymbol{p}(t)) - h(0_i, \boldsymbol{p}(t)), \quad i \in C.$$

□

From this theorem it follows that $E[Z_i]$ is a weighted average of the Birnbaum measure, $I_B^{(i)}(t)$ with $\bar{F}_i(t)(-\ln(\bar{F}_i(t)))$ as weight for all $i \in C$. By

(7.29) it follows that $I_N^{(i)}$ is proportional to $E[Z_i]$ for all $i \in C$. Thus, the ranking of the components with respect to the Natvig importance measure is the same as the ranking of the expected values of $Z_1, \ldots, Z_n$.

Just as for the Barlow-Proschan measure, the formula (7.33) is not always easy to use. In order to estimate the Natvig measure we instead go back to the original definition with the formula (7.29). Using Monte Carlo simulation it is easy to estimate $E[Z_i]$ for all $i \in C$. This is done by running N simulations of the system. More specifically, we generate N sets of component lifetimes, $\{T_{1k}, \ldots, T_{nk}\}$, where T_{ik} is sampled from the lifetime distribution of the ith component, $i = 1, \ldots, n$ and $k = 1, \ldots, N$. We also generate N sets of additional component lifetimes, $\{U_{1k}, \ldots, U_{nk}\}$, such that:

$$P(U_{ik} > u) = \frac{\bar{F}_i(u + \tau_{ik})}{\bar{F}_i(\tau_{ik})}, \quad u \geq 0, \quad i = 1, \ldots, n, \quad k = 1, \ldots, N,$$

where τ_{ik} denotes the observed value of T_{ik}, $i = 1, \ldots, n$ and $k = 1, \ldots, N$. In order to generate U_{ik}, it is typically easier to generate $\tau_{ik} + U_{ik}$, sampled such that:

$$P(\tau_{ik} + U_{ik} > t) = \frac{\bar{F}_i(t)}{\bar{F}_i(\tau_{ik})}, \quad u \geq 0, \quad i = 1, \ldots, n, \quad k = 1, \ldots, N.$$

This means that $\tau_{ik} + U_{ik}$ is sampled from the conditional distribution of T_{ik} given that $T_{ik} > \tau_{ik}$. This can be done by using a so-called *rejection method* where we sample repeatedly from the unconditional distribution of T_{ik} until we get a value T'_{ik} such that $T'_{ik} > \tau_{ik}$. The resulting value of U_{ik} is then given by $T'_{ik} - \tau_{ik}$, $i = 1, \ldots, n$ and $k = 1, \ldots, N$.

We now consider the kth simulation with the generated values $\{T_{1k}, \ldots, T_{nk}\}$ and $\{U_{1k}, \ldots, U_{nk}\}$. Using one of the methods described in Section 7.1.2 we calculate the system lifetime S_k as well as system lifetimes $\{S_{1k}, \ldots, S_{nk}\}$, where S_{ik} denotes the system lifetime given that the ith component is given a minimal repair, meaning that T_{ik} is replaced by $T_{ik} + U_{ik}$, $i = 1, \ldots, n$, $k = 1, \ldots, N$. Finally, we calculate $\{Z_{1k}, \ldots, Z_{nk}\}$, where $Z_{ik} = S_{ik} - S_k$, $i = 1, \ldots, n$, $k = 1, \ldots, N$.

The Natvig importance measure can now be estimated by:

$$\hat{I}_N^{(i)} = \frac{\frac{1}{N}\sum_{k=1}^{N} Z_{ik}}{\sum_{j=1}^{n} \frac{1}{N}\sum_{k=1}^{N} Z_{jk}} = \frac{\sum_{k=1}^{N} Z_{ik}}{\sum_{j=1}^{n}\sum_{k=1}^{N} Z_{jk}} \tag{7.34}$$

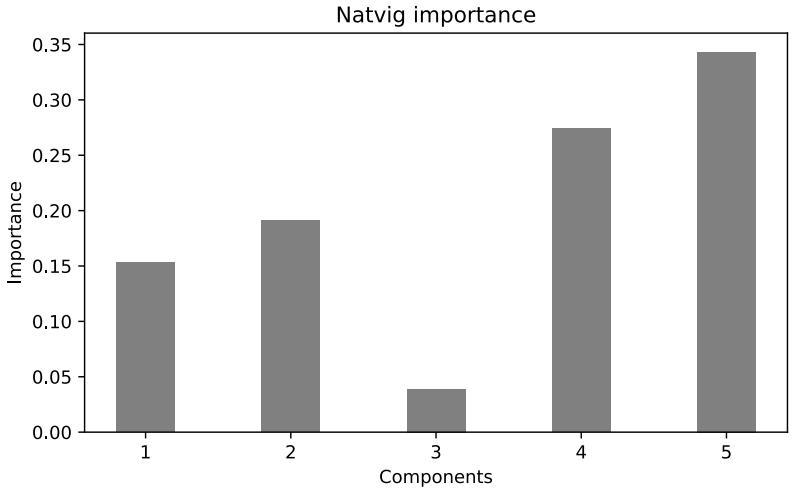

Figure 7.16: The Natvig importance measure for the components in a bridge system.

Example 7.4.4 *We consider the bridge system introduced in Example 7.1.3. To estimate* $I_N^{(i)}$ *for* $i \in C$*, we run a Monte Carlo simulation with* $N = 100000$ *simulations. The resulting estimated importance measures are plotted in Figure 7.16. A python script which can be used to produce this plot, is given in Script B.3.14 in Appendix B.3.*

Example 7.4.5 *We consider the threshold system introduced in Example 7.1.4. To estimate* $I_N^{(i)}$ *for* $i \in C$*, we run a Monte Carlo simulation with* $N = 100000$ *simulations. The resulting estimated importance measures are plotted in Figure 7.17. A python script which can be used to produce this plot, is given in Script B.3.15 in Appendix B.3.*

Example 7.4.6 *We consider the undirected network system introduced in Example 7.1.5. To estimate* $I_N^{(i)}$ *for* $i \in C$*, we run a Monte Carlo simulation with* $N = 100000$ *simulations. The resulting estimated importance measures are plotted in Figure 7.18. A python script which can be used to produce this plot, is given in Script B.3.16 in Appendix B.3.*

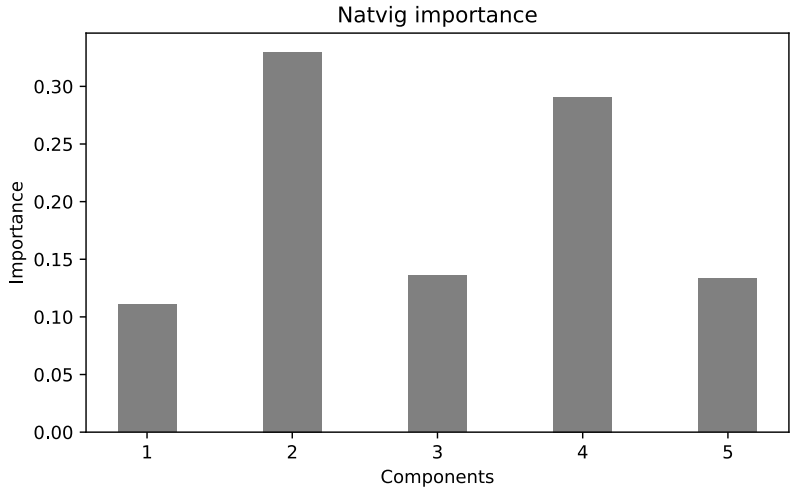

Figure 7.17: The Natvig importance measure for the components in a threshold system.

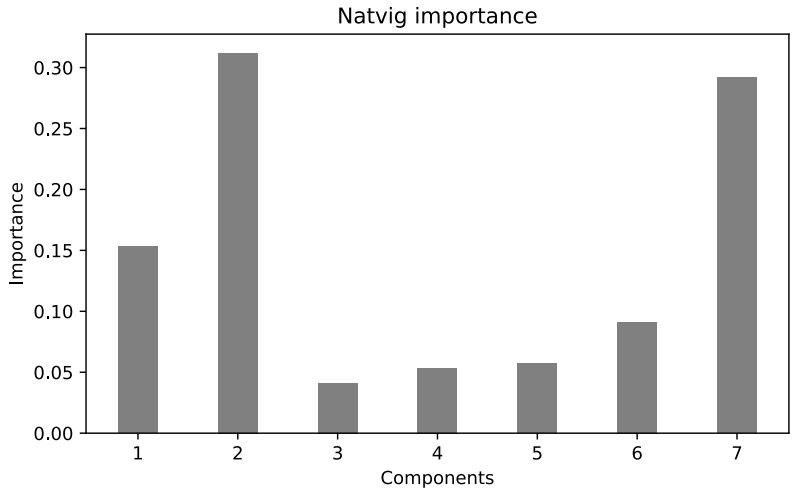

Figure 7.18: The Natvig importance measure for the components in an undirected network system.

7.5 Pure jump processes

In this section we formalise the concept of a pure jump process. Thus, let $\{S(t)\}$ be a stochastic process where $S(t)$ denotes the state of the process at time $t \geq 0$. Moreover, we let $T_1 < T_2 < \cdots$ denote the points of time of the events affecting the process, and let $T_0 = 0$. Following Klebaner [43] a pure jump process is a process where $S(t)$, can be written as:

$$S(t) = S(0) + \sum_{j=1}^{\infty} I(T_j \leq t) J_j, \qquad t \geq 0, \tag{7.35}$$

where $I(\cdot)$ denotes the indicator function, and J_j denotes the change in the state (positive or negative) of the process at time T_j. The representation (7.35) implies that the state function $S(t)$ is piecewise constant and right-continuous in t, with jumps at $T_1 < T_2 < \cdots$. In particular, for $k = 0, 1, \ldots$, we have:

$$S(t) = S(0) + \sum_{j=1}^{k} J_j = S(T_k), \qquad \text{for all } t \in [T_k, T_{k+1}). \tag{7.36}$$

Thus, in order to keep track of how the process evolves and update the value of the state function, only the points of time where the events happen need to be considered. This property is convenient during simulations.

The infinite sum in (7.35) indicates that the number of events occurring in the interval $[0, t]$ is unbounded. The possibility of having an infinite number of events in $[0, t]$, however, may cause various technical difficulties. In particular, this may cause simulations to break down since an infinite number of events need to be generated and handled. See Glasserman [29] for a further discussion of this issue. To avoid these difficulties, we always assume that the number of events occurring in any finite interval is finite with probability one. A pure jump process satisfying this assumption is said to be *regular*. Note, however, that the regularity assumption does not imply the existence of a fixed number $N < \infty$ such that the number of events in the interval $[0, t]$ is bounded by N with probability one. The number of events in $[0, t]$ will typically be a random variable with a distribution that ranges over all non-negative integers. When the simulation procedure is chosen, this must be taken into account.

We now present a few basic results on regularity of pure jump processes which we need later. We consider a pure jump process $\{S(t)\}$ with jumps at

$T_1 < T_2 < \cdots$. We also introduce the times between the events defined as:

$$\Delta_j = T_j - T_{j-1}, \qquad j = 1, 2, \ldots. \tag{7.37}$$

Using these quantities the event times can be expressed as:

$$T_k = \sum_{j=1}^{k} \Delta_j, \qquad k = 1, 2, \ldots. \tag{7.38}$$

Obviously, the process $\{S(t)\}$ is regular if and only if $T_\infty = \infty$ almost surely. Thus, it follows that a necessary and sufficient criterion for regularity is that the series $\sum_{j=1}^{\infty} \Delta_j$ is *divergent* with probability one. This condition can often be verified using the following simple result:

Proposition 7.5.1 *Let $\{S(t)\}$ be a pure jump process with jumps at $T_1 < T_2 < \cdots$. Moreover, we let $T_0 = 0$ and introduce the non-negative random variables $\Delta_j = T_j - T_{j-1}$, $j = 1, 2, \ldots$. Assume then that the sequence $\{\Delta_j\}$ contains an infinite subsequence $\{\Delta_{k_j}\}$ of independent, identically distributed random variables such that $E[\Delta_{k_j}] = d > 0$. Then S is regular.*

Proof: By the strong law of large numbers it follows that:

$$P(\lim_{n\to\infty} n^{-1} \sum_{j=1}^{n} \Delta_{k_j} = d) = 1.$$

This implies that the series $\sum_{j=1}^{\infty} \Delta_{k_j}$ is divergent with probability one. Hence, since obviously $\sum_{j=1}^{\infty} \Delta_{k_j} \leq \sum_{j=1}^{\infty} \Delta_j$, the result follows. □

The regularity property implies that the set of points where the process jumps almost surely does not have any accumulation points. The following result utilizes this to show the existence of left limits of the state function of a regular pure jump process.

Proposition 7.5.2 *Let $\{S(t)\}$ be a regular pure jump process with jumps at $T_1 < T_2 < \cdots$. Then $\lim_{t\to s^-} S(t)$ exists for every $s > 0$ with probability one.*

Proof: Let $0 \leq t < s < \infty$. We then consider the set $\mathcal{T} = \{T_j : t \leq T_j < s\} \cup \{t\}$. Since S is assumed to be regular, the number of elements in $\mathcal{T}$ is finite with probability one. Moreover, $\mathcal{T}$ is non-empty since $t \in \mathcal{T}$. Thus,

this set contains a maximal element, which we denote by t'. Moreover, since every element in $\mathcal{T}$ is less than s, then so is t'. From this it follows that the interval (t', s) is nonempty. At the same time (t', s) does not contain any jumps, so $S(t)$ is constant throughout this interval. Hence, $\lim_{t\to s^-} S(t)$ exists. Since s was arbitrary chosen, this holds for any $s > 0$. □

Regularity is also of importance when considering the integral of a pure jump process:

Proposition 7.5.3 *Let $\{S(t)\}$ be a regular pure jump process with jumps at $T_1 < T_2 < \cdots$, and let $0 \leq u < v < \infty$. Assume that $\{T_j : u < T_j < v\} = \{T^{(1)}, \ldots, T^{(k)}\}$, where $T^{(1)} < \cdots < T^{(k)}$. Moreover, we let $T^{(0)} = u$ and $T^{(k+1)} = v$. Then we have:*

$$\int_u^v S(t)dt = \sum_{j=0}^{k} S(T^{(j)})(T^{(j+1)} - T^{(j)}).$$

Proof: We first note that since S is assumed to be regular, the number of elements in the set $\{T_j : u < T_j < v\}$ is finite with probability one. Thus, this set can almost surely be written in the form $\{T^{(1)}, \ldots, T^{(k)}\}$, for some suitable $k < \infty$. Since S is right-continuous and piecewise constant, it follows that $S(t) = S(T^{(j)})$ for all $t \in [T^{(j)}, T^{(j+1)})$, $j = 0, 1, \ldots, k$. Thus, we have:

$$\int_{T^{(j)}}^{T^{(j+1)}} S(t)dt = S(T^{(j)})(T^{(j+1)} - T^{(j)}), \quad j = 0, 1, \ldots, k.$$

The result then follows by adding up the contributions to the integral from each of the $k + 1$ intervals $[T^{(0)}, T^{(1)}), \ldots, [T^{(k)}, T^{(k+1)})$ □

We then consider a system consisting of a collection of n regular pure jump processes, $S_1, \ldots, S_n$. The state of the system is then typically expressed as a function of the states of the elementary processes. It is easy to see that the system state also evolves as a regular pure jump process. That is, we have:

Proposition 7.5.4 *Let $\{S_1(t)\}, \ldots, \{S_n(t)\}$ be n regular pure jump processes, and let $H(t) = H(\boldsymbol{S}(t))$, where $\boldsymbol{S}(t) = (S_1(t) \ldots, S_n(t))$, $t \geq 0$. Then $\{H(t)\}$ is a regular pure jump process as well. That is, $H(t) = H(\boldsymbol{S}(t))$ is piecewise constant and right-continuous in t, and the number of jumps in any finite interval is finite with probability one.*

Proof: Let $\mathcal{T}_i$ be the set of time points corresponding to the jumps of the process S_i, $i = 1, \ldots, n$, and let $\mathcal{T}$ be the set of time points corresponding to the jumps of the process $\{H(t)\}$. Since the state value of H cannot change unless there is a change in the state value of at least one of the elementary processes, it follows that $\mathcal{T} \subseteq (\mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n)$. Thus, $H(t)$ is piecewise constant and right-continuous in t. Moreover, for any finite interval $[t, s]$ we also have:

$$\mathcal{T} \cap [t, s] \subseteq [(\mathcal{T}_1 \cap [t, s]) \cup \cdots \cup (\mathcal{T}_n \cap [t, s])].$$

Since by regularity $(\mathcal{T}_i \cap [t, s])$ is finite for $i = 1, \ldots, n$, it follows that $\mathcal{T} \cap [t, s]$ is finite as well. Hence, we conclude that $\{H(t)\}$ is regular. □

7.6 Repairable binary monotone systems

As we shall see, stationary statistical properties of a pure jump process, can often be calculated by working directly on the stationary probability distributions of the elementary processes. Sometimes, however, one needs to estimate how the expected value of the process evolves over time. In such cases it is often necessary to use some sort of Monte Carlo simulation method.

In order to illustrate how this can be done we consider a binary monotone system (C, ϕ) of n *repairable* components. The elementary processes are the component state processes, denoted $\{X_1(t)\}, \ldots, \{X_n(t)\}$, where $X_i(t)$ represents the state of component i at time $t \geq 0$, $i = 1, \ldots, n$. That is, $X_i(t) = 1$ if component i is functioning at time t, and zero otherwise, $i = 1, \ldots, n$. We also introduce the component state vector at time $t \geq 0$, $\boldsymbol{X}(t) = (X_1(t), \ldots, X_n(t))$. Thus, $\phi = \phi(\boldsymbol{X}(t)) = 1$ if the system is functioning at time t and zero otherwise.

We now introduce the lifetimes and repair times for the components. That is, for $i = 1, \ldots, n$ and $j = 1, 2, \ldots$ we let:

$$U_{ij} = \text{The } j\text{th lifetime of the } i\text{th component.} \tag{7.39}$$

$$D_{ij} = \text{The } j\text{th repair time of the } i\text{th component.} \tag{7.40}$$

We assume that U_{ij} has an absolutely continuous distribution F_i with a positive mean value $\mu_i < \infty$, while D_{ij} has an absolutely continuous distribution G_i with a positive mean value $\nu_i < \infty$, $i = 1, \ldots, n$, $j = 1, 2, \ldots$. All lifetimes and repair times are assumed to be independent. Thus, in particular the component states $X_1(t), \ldots, X_n(t)$ are independent for each $t \geq 0$.

We then introduce the *availability* of the ith component at time t, denoted $A_i(t)$, as the probability that the component is functioning at time t. That is, for $i = 1, \ldots, n$ we have:

$$A_i(t) = \Pr(X_i(t) = 1) = E[X_i(t)].$$

It can then be shown (see Barlow and Proschan [6]) that the corresponding *stationary availabilities* can be expressed as:

$$A_i = \lim_{t\to\infty} A_i(t) = \frac{\mu_i}{\mu_i + \nu_i}, \qquad i = 1, \ldots, n. \tag{7.41}$$

Introduce $\boldsymbol{A}(t) = (A_1(t), \ldots, A_n(t))$ and $\boldsymbol{A} = (A_1, \ldots, A_n)$. The system availability at time t is given by:

$$A_\phi(t) = \Pr(\phi(\boldsymbol{X}(t)) = 1) = E[\phi(\boldsymbol{X}(t))] = h(\boldsymbol{A}(t)),$$

where h is the system's reliability function. The corresponding stationary availability is given by:

$$A_\phi = \lim_{t\to\infty} A_\phi(t) = h(\boldsymbol{A}). \tag{7.42}$$

We now extend Definition 5.1.1 to repairable components in the obvious way by stating that component i is *critical* at time t if:

$$\psi_i(\boldsymbol{X}(t)) = \phi(1_i, \boldsymbol{X}(t)) - \phi(0_i, \boldsymbol{X}(t)) = 1. \tag{7.43}$$

We will refer to $\psi_i(\boldsymbol{X}(t))$ as the *criticality state* of component i at time t. Moreover, we extend Definition 5.2.1 to this context by defining the Birnbaum measure of importance of a repairable component i at time t, as the probability that component i is critical at time t. This is denoted $I_B^{(i)}(t)$ and given by:

$$\begin{aligned} I_B^{(i)}(t) &= \Pr(\psi_i(\boldsymbol{X}(t)) = 1) = E[\psi_i(\boldsymbol{X}(t))] \\ &= h(1_i, \boldsymbol{A}(t)) - h(0_i, \boldsymbol{A}(t)). \end{aligned} \tag{7.44}$$

The corresponding stationary measure is given by:

$$I_B^{(i)} = \lim_{t\to\infty} I_B^{(i)}(t) = h(1_i, \boldsymbol{A}) - h(0_i, \boldsymbol{A}). \tag{7.45}$$

7.7 Simulating repairable systems

Discrete event models are typically used in simulation studies to model and analyze *pure jump processes*. For an extensive introduction to discrete event models we refer to Glasserman [28]. See also Klebaner [43]. A discrete event model can be viewed as a system consisting of a collection of stochastic processes, where the states of the individual processes change as results of various kinds of *events* occurring at random points of time. Between these events the states of the processes are assumed to be constant. We refer to the processes included in the collection, as the *elementary processes* of the system. In our context we always assume that each event only affects *one* of the elementary processes.

Asymptotic system availability and component criticality, can easily be estimated by running a single discrete event simulation on the system over a sufficiently long time horizon, or by working directly on the stationary component availabilities. Sometimes, however, one needs to estimate how these properties evolve over time. In such cases it is necessary to run many simulations to obtain stable availability and criticality estimates. One possible approach to this problem is to sample the system state at fixed points of time, and then use the mean values of the states at these points as curve estimates. With this approach all information about the process between the sampling points is thrown away. In the present paper we propose an alternative sampling procedure where we utilize process data between the sampling points as well. This simulation method is particularly useful when estimating various kinds of component importance measures for repairable systems.

Discrete event simulation is a very well established technique for analyzing systems of pure jump processes with applications in many different areas such as queueing and inventory systems and reliability. For an introduction to this field see Banks et al. [4].

As in the previous section we consider a binary monotone system (C, ϕ) with component state processes, denoted $\{X_1(t)\}, \ldots, \{X_n(t)\}$. The events affecting the ith component are denoted by $E_{i1}, E_{i2}, \ldots$, listed in chronological order, $i = 1, \ldots, n$. Since we assumed that all lifetimes and repair times have absolutely continuous distributions, all these events happen at distinct points of time almost surely. We let $T_{i1} < T_{i2}, \ldots$ be the corresponding points of time for these events. We also let $T_{i0} = 0$, $i = 1, \ldots, n$. As in (7.35) the

component state processes can then be expressed as:

$$X_i(t) = X_i(0) + \sum_{j=1}^{\infty} I(T_{ij} \leq t) J_{ij}, \qquad t \geq 0, ^i = 1, \ldots, n, \tag{7.46}$$

where the jumps J_{ij} are either -1 if E_{ij} is a failure event, or $+1$ if E_{ij} is a repair event. We assume that all components start out by being functioning. Thus, we have $X_i(0) = 1$, and $J_{ij} = (-1)^j$, for $i = 1, \ldots, n$ and $j = 1, 2, \ldots$. Finally, for $i = 1, \ldots, n$ we introduce the times between the events defined as:

$$\Delta_{ij} = T_{ij} - T_{ij-1}, \qquad i = 1, \ldots, n, ^j = 1, 2, \ldots. \tag{7.47}$$

Then for $i = 1, \ldots, n$ we have:

$$\Delta_{i1} = U_{i1}, \quad \Delta_{i2} = D_{i1}, \quad \Delta_{i3} = U_{i2}, \quad \ldots \tag{7.48}$$

Since $U_{i1}, U_{i2}, \ldots$ are independent and identically distributed with positive mean value μ_i, it follows by Proposition 7.5.1 that X_i is a regular pure jump process, $i = 1, \ldots, n$. Hence, by Proposition 7.5.4 the system state $\phi = \phi(\boldsymbol{X})$ as well as the criticality states $\psi_1(\boldsymbol{X}), \ldots, \psi_n(\boldsymbol{X})$ are regular pure jump processes.

At the system level the event set is the *union* of all the component event sets. Let $E^{(1)}, E^{(2)}, \ldots$ denote these events sorted with respect to their respective points of time, and let $T^{(1)} < T^{(2)} < \cdots$ be the corresponding points of time. Note that since we assumed that all lifetimes and repair times have absolutely continuous distributions, each system event corresponds almost surely to a unique component event.

In order to simulate such a system, we use an *object oriented approach* where the components as well as the system are represented as *objects*. The component objects are equipped with methods for generating failure and repair events according to their respective life- and repair time distributions. The system object determines the state of the system as a function of the component states. To keep track of the events and process them in the correct order, they are organized in a dynamic queue sorted with respect to the points of time of the events. The component processes place their upcoming events into the queue where they stay until they are processed.

More specifically, at time zero each component starts out by being functioning, and places its first failure event into the queue. As soon as all these failure events have been placed into the queue, the first event in the queue

is processed. That is, the *system time* is set to the time of the first event, and the event is taken out of the queue and passed on to the component responsible for handling this event. The component then updates its state, generates a new event, in this case a repair event, which is placed into queue, and notifies the system about its new state so that the system state can be updated as well. Then the next event in the queue is processed in the same fashion, and so forth until the system time reaches a certain predefined point of time. Note that since the component events are generated as part of the event processing, the number of events in the queue stays constant.

Although the system state and component states stay constant between events, it may still be of interest to log the state values at predefined points of time. In order to facilitate this, we introduce yet another type of event, called a *sampling event*. Such sampling events will typically be spread out evenly on the timeline. Thus, if $e_1, e_2, \ldots$ denote the sampling events, and $t_1 < t_2 < \cdots$ are the corresponding points of time, we would typically have $t_j = j \cdot \Delta$ for some suitable number $\Delta > 0$.

The sampling events will be placed into the queue in the same way as for the ordinary events. As a sampling event is processed, the next sampling event will be placed into the queue. Thus, at any time only one sampling event needs to be in the queue.

In principle one must update the system state every time there is a change in the component states. For large complex systems, these updates may slow down the simulations considerably. Thus, whenever possible one should avoid computing the system state. Fortunately, since the structure function of a binary monotone system is non-decreasing in each argument, it is possible to reduce the updating to a minimum. To explain this in detail, we consider the event E_{ij} affecting component i. Let T_{ij} be the corresponding point of time, and let $\boldsymbol{X}(T_{ij}^-)$ denote the value of the component state vector immediately before E_{ij} occurs, i.e., $\boldsymbol{X}(T_{ij}^-) = \lim_{t \to T_{ij}^-} \boldsymbol{X}(t)$. Note that by Proposition 7.5.2 these limits exist since the component state processes are regular.

If E_{ij} is a failure event of component i, i.e., $X_i(T_{ij}^-) = 1$ and $X_i(T_{ij}) = 0$, then the event cannot change the system state if the system is already failed, i.e., $\phi(\boldsymbol{X}(T_{ij}^-)) = 0$. Similarly, if E_{ij} is a repair event of component i, i.e., $X_i(T_{ij}^-) = 0$ and $X_i(T_{ij}) = 1$, this event cannot change the system state if the system is already functioning, i.e., $\phi(\boldsymbol{X}(T_{ij}^-)) = 1$. Thus, we see that we only need to recalculate the system state whenever:

$$\phi(\boldsymbol{X}(T_{ij}^-)) \neq X_i(T_{ij}). \tag{7.49}$$

Hence, the number of times we need to recalculate the system state is drastically reduced.

In cases where we keep track of the criticality state of each of the components, we can simplify the calculations even further by noting that the system state is changed as a result of the event E_{ij} if and only if component i is critical at the time of the event. Moreover, if i is critical, and E_{ij} is a failure event, it follows that the system fails as a result of this event, i.e., $\phi(\boldsymbol{X}(T_{ij})) = 0$. If on the other hand i is critical, and E_{ij} is a repair event, it follows that the system becomes functioning as a result of this event, i.e., $\phi(\boldsymbol{X}(T_{ij})) = 1$. Thus, we see that in this setup all the calculations we need to carry out, are related to the updating of the criticality states.

A similar technique can be used when updating the criticality states of the components. Thus, we consider the event E_{ij} affecting the state of component i. We first note that the criticality state function of component i, $\psi_i(\boldsymbol{X}(t)) = \phi(1_i, \boldsymbol{X}(t)) - \phi(0_i, \boldsymbol{X}(t))$ does not depend on the state of component i. Thus, the event E_{ij} does not have any impact on the criticality state of i. However, E_{ij} may still change the criticality state of other components in the system even when the system state remains unchanged. Thus, let $k \neq i$ be another component, and consider its criticality state function $\psi_k(\boldsymbol{X}(T_{ij}))$.

If $X_k(T_{ij}) = 1$ and $\phi(\boldsymbol{X}(T_{ij})) = 0$, it follows that:

$$\phi(1_k, \boldsymbol{X}(T_{ij})) = \phi(0_k, \boldsymbol{X}(T_{ij})) = 0. \tag{7.50}$$

Thus, in this case we must have $\psi_k(\boldsymbol{X}(T_{ij})) = 0$. On the other hand, if $X_k(T_{ij}) = 0$ and $\phi(\boldsymbol{X}(T_{ij})) = 1$, it follows that:

$$\phi(1_k, \boldsymbol{X}(T_{ij})) = \phi(0_k, \boldsymbol{X}(T_{ij})) = 1. \tag{7.51}$$

Thus, we must have $\psi_k(\boldsymbol{X}(T_{ij})) = 0$ in this case as well. Hence, we see that a necessary condition for component k to be critical at time T_{ij} is that:

$$\phi(\boldsymbol{X}(T_{ij})) = X_k(T_{ij}). \tag{7.52}$$

Utilizing these observations reduces the need to recalculate the criticality states.

7.8 Estimating availability and importance

Stationary availability and importance measures are typically easy to derive. If the system under consideration is not too complex, these quantities can

be calculated analytically using (7.41), (7.42) and (7.45). For larger complex systems one may estimate the availability and importance using Monte Carlo simulations. A fast simulation algorithm for this is provided in [?]. Alternatively, estimates can be obtained by running a *single* discrete event simulation on the system over a sufficiently long time horizon.

Here, however, we focus on the problem of estimating the system availability $A_\phi(t)$ and the component importance measures $I_B^{(1)}(t), \ldots, I_B^{(n)}(t)$ as functions of t. Ideally we would like to estimate these quantities for any $t \geq 0$. For practical purposes, however, we have to limit the estimation to a finite set of points. More specifically, we will estimate $A_\phi(t)$ for $t \in \{t_1, \ldots, t_N\}$, i.e., the set of the N first sampling points. For the points of time between the sampling points, we just use linear interpolation to obtain the curve estimate.

A simple approach to this problem is to run M simulations on the system, where each simulation covers the time interval $[0, t_N]$. In each simulation we sample the values of ϕ and $\psi_1, \ldots, \psi_n$ at each sampling point $t_1, \ldots, t_N$. We denote the sth simulated value of the component state vector process at time $t \geq 0$ by $\boldsymbol{X}_s(t)$, $s = 1, \ldots, M$, and obtain the following estimates for $j = 1, \ldots, N$:

$$\hat{A}_\phi(t_j) = \frac{1}{M} \sum_{s=1}^{M} \phi(\boldsymbol{X}_s(t_j)), \tag{7.53}$$

$$\hat{I}_B^{(i)}(t_j) = \frac{1}{M} \sum_{s=1}^{M} \psi_i(\boldsymbol{X}_s(t_j)). \tag{7.54}$$

We will refer to these estimates as *pointwise estimates*. It is easy to see that for $j = 1, \ldots, N$, $\hat{A}_\phi(t_j)$ and $\hat{I}_B^{(i)}(t_j)$ are unbiased and strongly consistent estimates of $A_\phi(t_j)$ and $I_B^{(i)}(t_j)$ respectively. In order to estimate $A_\phi(t)$ and $I_B^{(i)}(t)$ between the sampling points, one may use interpolation. Using a sufficiently high sampling rate, i.e., a small value of Δ, a satisfactory estimate of the full curve can be obtained. Still, all information about the process between the sampling points is thrown away.

We now present an alternative approach where we utilize process data between the sampling points as well. As above we assume that the system is simulated M times over the interval $[0, t_N]$, and let $\boldsymbol{X}_s(t)$ denote the sth simulated value of the component state vector process at time $t \geq 0$, $s = 1, \ldots, M$. Then let $E_s^{(1)}, E_s^{(2)}, \ldots$ denote the events in the interval $[0, t_N]$ in the sth simulation, *including* sampling events at times $t_1, \ldots, t_N$, and let

$T_s^{(1)} < T_s^{(2)} < \cdots$ be the corresponding points of time, $s = 1, \ldots, M$. In this case we also include an extra sampling event in each simulation at time $t_0 = 0$, denoted $E_s^{(0)}$, and let $T_s^{(0)} = 0$, $s = 1, \ldots, M$.

The idea now is to use average simulated availability and criticalities from each interval $[t_{j-1}, t_j)$, $j = 1, \ldots, N$ as respective estimates for the availability and criticalities at the midpoints of these intervals. By using Proposition 7.5.3, we obtain the following estimates for $j = 1, \ldots, N$:

$$\tilde{A}_\phi(\bar{t}_j) = \frac{1}{M} \sum_{s=1}^{M} \frac{1}{\Delta} \sum_{k \in \mathcal{E}_{sj}} \phi(\boldsymbol{X}_s(T_s^{(k)}))(T_s^{(k+1)} - T_s^{(k)}), \tag{7.55}$$

$$\tilde{I}_B^{(i)}(\bar{t}_j) = \frac{1}{M} \sum_{s=1}^{M} \frac{1}{\Delta} \sum_{k \in \mathcal{E}_{sj}} \psi_i(\boldsymbol{X}_s(T_s^{(k)}))(T_s^{(k+1)} - T_s^{(k)}), \tag{7.56}$$

where $\mathcal{E}_{sj}$ denotes the index set of the events in $[t_{j-1}, t_j)$ in the sth simulation, and where we have introduced the interval midpoints $\bar{t}_j = (t_{j-1} + t_j)/2$, $j = 1, \ldots, N$. We will refer to these estimates as *interval estimates*.

By Proposition 7.5.3, it follows that for $j = 1, \ldots, N$, $\tilde{A}_\phi(\bar{t}_j)$ and $\tilde{I}_B^{(i)}(\bar{t}_j)$ are unbiased and strongly consistent estimates of the corresponding average availability and criticality in the intervals $[t_{j-1}, t_j)$ respectively. By choosing Δ so that the availabilities and criticalities are relatively stable within each interval, the interval estimates are approximately unbiased estimates for $A_\phi(\bar{t}_j)$ and $I_B^{(i)}(\bar{t}_j)$ as well. In fact the resulting interval estimates tend to stabilize much faster than the pointwise estimates. In order to estimate $A_\phi(t)$ and $I_B^{(i)}(t)$ between the interval midpoints, one may again use interpolation. Note that since all process information is used in the estimates, satisfactory curve estimates can be obtained for a much higher value of Δ than the one needed for the pointwise estimates.

In order to illustrate this we consider a simple bridge system shown in Figure 7.19. The components of this system are the five edges in the graph, labeled $1, \ldots, 5$. The system is functioning if the source node s can communicate with the terminal node t through the graph. All the components in the system have exponential lifetime and repair time distributions with mean values 1 time unit. The objective of the simulation is to estimate $A_\phi(t)$ and $I_B^{(1)}(t), \ldots, I_B^{(5)}(t)$ for $t \in [0, t_N]$, where $t_N = 1000$.

All the simulations were carried out using a program called *Eventcue*[3].

[3]Eventcue is a java program developed at the Department of Mathematics, University

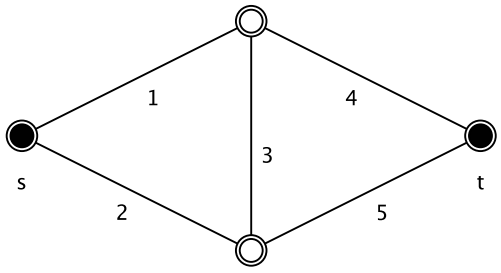

Figure 7.19: A bridge system.

This program has an intuitive graphical user interface, and can be used to estimate availability and criticality of any undirected network system.

Since all the lifetimes and repair times are exponentially distributed with the *same mean*, it is easy to derive explicit analytical expressions for the component availabilities. To see this, we consider the ith component at a given point of time t and introduce $N_i(t)$ as the number of failure and repair events affecting component i in $[0, t]$. With times between events being independent and exponentially distributed with mean 1 it follows that $N_i(t)$ has a Poisson distribution with mean t. Moreover, component i is functioning at time t if and only if $N_i(t)$ is even. Thus, the ith component availability at time t is given by:

$$A_i(t) = \sum_{k=0}^{\infty} \Pr(N_i(t) = 2k) = \sum_{k=0}^{\infty} \frac{t^{2k}}{(2k)!} e^{-t}. \tag{7.57}$$

Using (7.57) one can verify numerically that all the component availabilities converge very fast towards their common stationary value, 0.5. As a result of this the system availability, $A_\phi(t)$, converges very fast towards its stationary value, 0.5, as well. In fact, for $t > 20$, numerical calculations show that $|A_\phi(t) - 0.5| < 10^{-15}$. Similarly, the Birnbaum measures of importance converges so that for $t > 20$, $|I_B^{(i)}(t) - 0.375| < 10^{-15}$, $i = 1, 2, 4, 5$, while $|I_B^{(3)}(t) - 0.125| < 10^{-15}$. Thus, for $t > 20$ the true values of all the curves are approximately constant. This makes it easy to evaluate and compare the quality of the different Monte Carlo estimates in this particular case.

of Oslo. The program is freely available at http://www.riscue.org/eventcue/.

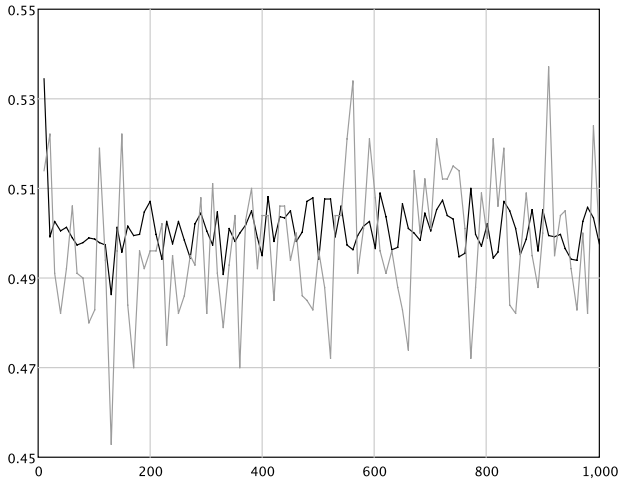

Figure 7.20: Interval estimate (black curve) and pointwise estimate (gray curve) of the availability curve.

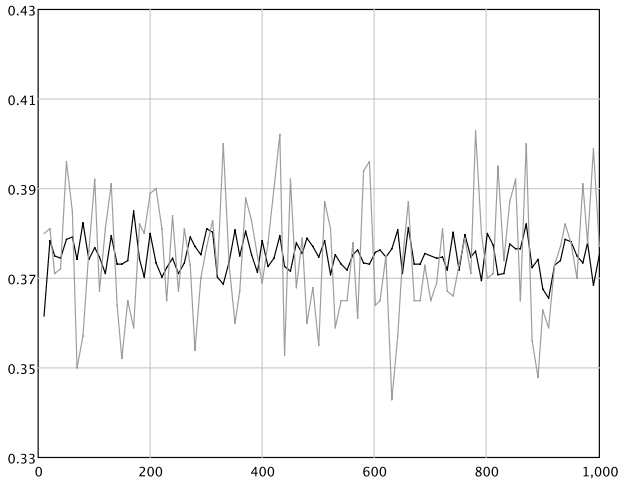

Figure 7.21: Interval estimate (black curve) and pointwise estimate (gray curve) of the importance curve.

Figure 7.20 and Figure 7.21 show respectively the availability curve and the criticality curve of component 1. The black curves are obtained using the interval estimates, while the gray curves show the corresponding pointwise estimate curves. In all cases we have used $M = 1000$ simulations and $N =$

100 sample points.

The plots clearly show the difference between the two methods. The black interval estimate curves are much more stable, and thus much closer to the true curve values, compared to the gray pointwise estimates.

One may think that increasing the number of sampling points would make the pointwise curve estimate better as more information is sampled. However, it turns out that the main effect of this is that the curve jumps more and more up and down. In fact with shorter intervals between sampling points the interval estimate becomes more unstable as well, and in the limit where the interval lengths go to zero, the two methods become equivalent. The only effective way of stabilizing the results for the pointwise curve estimate is to increase the number of simulations, i.e., M.

Table 7.1: Standard deviations for the pointwise curve estimates

M	2000	4000	6000	8000
St.dev.	0.0121	0.0076	0.0062	0.0054

In Table 7.1 we have listed estimated standard deviations for pointwise curve estimates for different values of M. We see that the standard deviation shows a steady decline as M increases. The corresponding numbers for $M = 1000$ are 0.0055 for the interval curve estimate and 0.0148 for the pointwise estimate. Thus, in this particular case we see that to obtain a pointwise curve estimate with a comparable stability to the interval curve estimate, one needs about eight times as many simulations.

For the interval curve estimate it is possible to obtain an even smoother curve simply by increasing Δ. Still, in general Δ should not be made too large, as this could produce a curve where important effects are obscured. Thus, in order to obtain optimal results, one should try out different values for Δ, and balance smoothness against the need of capturing significant oscillation properties of the curve.

Now, if smoothness is important, it is of course possible to apply some standard smoothing technique, such as moving averages or exponential smoothing, to the pointwise curve estimate. While such post-smoothing would clearly make the curve smoother, this technique does not add any new information to the estimate. The main advantage with the interval curve estimates is that such estimates actually use information about all events. Especially in cases where events occur at a very high rate, this turns out to be a great advantage.

7.9 Exercises

Exercise 1. Assume that T is Weibull distributed with parameters $\alpha > 0$ and $\beta > 0$.

a) Show that:

$$E[T] = \beta \cdot \Gamma(\frac{1}{\alpha} + 1),$$

where:

$$\Gamma(a) = \int_0^\infty x^{a-1} e^{-x} dx, \quad a > 0.$$

Hint: Use Lemma 7.4.2 and that $a \cdot \Gamma(a) = \Gamma(a+1)$, for all $a > 0$.

b) Show that:

$$Med[T] = \beta \cdot [\ln(2)]^{1/\alpha}$$

Exercise 2. Let (C, ϕ) be the binary monotone system shown in Figure 7.22. The lifetime of component i, T_i is Weibull-distributed with parameters a_i and b_i, for all $i \in C$ where:

$$a_1 = 2.0, \quad a_2 = 2.5, \quad a_3 = 3.0, \quad a_4 = 1.0,$$
$$b_1 = 50.0, \quad b_2 = 60.0, \quad b_3 = 70.0, \quad b_4 = 40.0.$$

a) Find the structure function of the system.

b) Consider the python script B.3.1. Modify this script so that you can use it to calculate $h(\boldsymbol{p}(t))$ for the system shown in Figure 7.22. Save the resulting plot to a file.

c) Find the minimal path and cut sets of the system.

d) Consider the python script B.3.2. Modify this script so that you can use it to estimate $h(\boldsymbol{p}(t))$ for the system shown in Figure 7.22. Save the resulting plot to a file. Compare the resulting plot to the one you found in (b).

Exercise 3. Consider a binary monotone system (C, ϕ) with minimal path sets $P_1, \ldots, P_p$ and minimal cut sets $K_1, \ldots, K_k$. We assume that the components cannot be repaired, and let T_i denote the lifetime of component $i \in C$. Moreover, we let S denote the lifetime of the system. Prove that:

$$S = \min_{1 \le j \le k} \max_{i \in K_j} T_i = \max_{1 \le j \le p} \min_{i \in P_j} T_i.$$

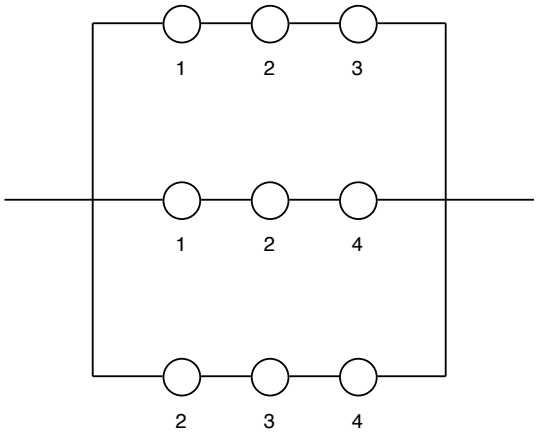

Figure 7.22: A binary monotone system (C, ϕ).

Exercise 4. Assume that the component lifetimes have so-called *proportional hazards*, that is:

$$\bar{F}_i(t) = \exp(-\lambda_i R(t)), \quad \lambda_i > 0, t \geq 0, \quad i = 1, \ldots, n,$$

where R is a strictly increasing, differentiable function such that $R(0) = 0$, and $\lim_{t\to\infty} R(t) = \infty$. Prove that for a series structure, we have:

$$I_{B-P}^{(i)} = I_N^{(i)} = \frac{\lambda_i}{\sum_{j=1}^n \lambda_j}.$$

Exercise 5. Assume that the i'th component is irrelevant for the system ϕ. Then, what is $I_{B-P}^{(i)}$ and $I_N^{(i)}$?

Chapter 8

Association and bounds for the system reliability

In this chapter, we begin by introducing a natural kind of positive dependency between random variables (for instance component states in a system). This positive dependency is called association. We show that associated random variables have some nice properties, which can be used in order to derive upper and lower bounds of the reliability of a system. As previously mentioned, it is not always possible to derive the exact reliability of a system due to e.g. computational time or complexity of the system in question. Nevertheless, we would like to find out something about how reliable a system is. This is the purpose of reliability bounds. In Section 8.2, we will derive several different such bounds, where some assume independence of components, while others just assume that the component states are associated.

8.1 Associated random variables

So far, we have usually assumed that the components in a system are independent, and hence that the corresponding random variables are independent. However, in many real-life situations, this is not the case. Some examples of such dependencies are components which are exposed to a common source of outer stress (for instance parts of a windmill) and structures where several components share a common load, so if one component fails, the load on the other components increases (for instance parts of a bridge or a house). In

this section, we define a concept of non-negative dependence between random variables called association.

A fairly natural definition of non-negative association is that two random variables S and T are associated if and only if $\mathrm{Cov}(S,T) \geq 0$. Though this definition is intuitive, the actual definition of association is stricter and applicable to any number of random variables.

Definition 8.1.1 *Let $T_1, \ldots, T_n$ be random variables, and let $\boldsymbol{T} = (T_1, \ldots, T_n)$. We say that $T_1, \ldots, T_n$ are* associated *if*

$$\mathrm{Cov}(\Gamma(\boldsymbol{T}), \Delta(\boldsymbol{T})) \geq 0,$$

for all binary non-decreasing functions Γ and Δ.

Note that according to this definition we only require $\mathrm{Cov}(\Gamma(\boldsymbol{T}), \Delta(\boldsymbol{T})) \geq 0$ for all *binary* non-decreasing functions. This property may seem somewhat strange. It turns out, however, that association implies the following apparently stronger result:

Theorem 8.1.2 *Let $T_1, \ldots, T_n$ be associated random variables, and f and g functions which are non-decreasing in each argument such that $\mathrm{Cov}(f(\boldsymbol{T}), g(\boldsymbol{T}))$ exists, i.e.,*

$$E[|f(\boldsymbol{T})|] < \infty, E[|g(\boldsymbol{T})|] < \infty, E[|f(\boldsymbol{T})g(\boldsymbol{T})|] < \infty.$$

Then we have:

$$\mathrm{Cov}(f(\boldsymbol{T}), g(\boldsymbol{T})) \geq 0.$$

We skip the proof here and refer to Barlow and Proschan [6] for details.

Associated random variables have some very useful properties. In order to establish these properties we need the following lemma:

Lemma 8.1.3 *Let X, Y and Z be random variables, and assume that $\mathrm{Cov}(X,Y) < \infty$. Then we have:*

$$\mathrm{Cov}(X,Y) = E[\mathrm{Cov}(X,Y|Z)] + \mathrm{Cov}[E(X|Z), E(Y|Z)].$$

Proof: By the definition of covariance we know that:

$$\mathrm{Cov}(X,Y) = E(XY) - E(X)E(Y).$$

We then apply the law of total expectation and condition with respect to the variable Z. This yields:

$$\mathrm{Cov}[X,Y] = E[E(XY|Z)] - E[E(X|Z)]E[E(Y|Z)]$$

Now we rewrite the first term using the definition of covariance once again:

$$\begin{aligned} E[E(XY|Z)] &= E[E(XY|Z) - E(X|Z)E(Y|Z)] + E[E(X|Z)E(Y|Z)] \\ &= E[\mathrm{Cov}(X,Y|Z)] + E[E(X|Z)E(Y|Z)] \end{aligned}$$

Inserting this in the expression for $\mathrm{Cov}(X,Y)$ we get:

$$\begin{aligned} \mathrm{Cov}[X,Y] &= E[\mathrm{Cov}(X,Y|Z)] + E[E(X|Z)E(Y|Z)] - E[E(X|Z)]E[E(Y|Z)] \\ &= E[\mathrm{Cov}(X,Y|Z)] + \mathrm{Cov}[E(X|Z), E(Y|Z)], \end{aligned}$$

which concludes the proof of the lemma. □

Theorem 8.1.4 *Associated random variables have the following properties:*
(i) Any subset of a set of associated random variables also consists of associated random variables.
(ii) A single random variable is always associated.
(iii) Non-decreasing functions of associated random variables are associated.
(iv) If two sets of associated random variables are independent, then their union is a set of associated random variables.

Proof: We note that (i) follows from (iii). However, we can also prove this property directly. Thus, let $T_1, \ldots, T_n$ be a set of associated random variables, and let $A \subset \{1, \ldots, n\}$. We would like to prove that $\{T_i\}_{i \in A}$ is a set of associated random variables. To do so, let Γ_A, Δ_A be arbitrary, binary functions which are non-decreasing in all of their arguments T_i, $i \in A$. We then define:

$$\Gamma(\boldsymbol{T}) = \Gamma_A(\boldsymbol{T}^A), \quad \Delta(\boldsymbol{T}) = \Delta_A(\boldsymbol{T}^A).$$

From this it follows that:

$$\mathrm{Cov}(\Gamma_A(\boldsymbol{T}^A), \Delta_A(\boldsymbol{T}^A)) = \mathrm{Cov}(\Gamma(\boldsymbol{T}), \Delta(\boldsymbol{T})) \geq 0,$$

where the inequality follows from Definition 8.1.1 because we have assumed that $T_1, \ldots, T_n$ is a set of associated random variables. Hence, (i) is proved.

To prove (ii) we let T be a random variable, and let Γ, Δ be arbitrary, binary functions which are non-decreasing in T. Then, since Γ, Δ are binary and non-decreasing in T, there are two possible cases:

CASE 1. $\Gamma(T) \leq \Delta(T)$ for all T,

CASE 2. $\Gamma(T) \geq \Delta(T)$ for all T.

We consider Case 1 only, as Case 2 can be handled similarly. Then, we have:

$$\begin{aligned}\mathrm{Cov}(\Gamma(T), \Delta(T)) &= E[\Gamma(T)\Delta(T)] - E[\Gamma(T)]E[\Delta(T)] \\ &= E[\Gamma(T)] - E[\Gamma(T)]E[\Delta(T)] \\ &= E[\Gamma(T)](1 - E[\Delta(T)]) \geq 0,\end{aligned}$$

where the second equality follows from the fact that we are in case 1. This means that either $\Gamma(T) = \Delta(T)$, in which case (since Γ, Δ are binary) $E[\Gamma(T)\Delta(T)] = E[\Gamma(T)]$. Or, alternatively, $\Gamma(T) = 0$, $\Delta(T) = 1$, in which case $E[\Gamma(T)\Delta(T)] = 0 = E[\Gamma(T)]$. The inequality holds because $\Gamma(T), \Delta(T) \in \{0, 1\}$ for all T. Hence, (ii) is proved as well.

We now turn to the proof of (iii) and let $T_1, \ldots, T_n$ be associated, and let $\boldsymbol{T} = (T_1, \ldots, T_n)$. Moreover, we let $S_i = f_i(\boldsymbol{T})$, $i = 1, \ldots, m$, where $f_1, \ldots, f_m$ are non-decreasing functions, and let $\boldsymbol{S} = (S_1, \ldots, S_m)$. Finally, let $\Gamma = \Gamma(\boldsymbol{S})$ and $\Delta = \Delta(\boldsymbol{S})$ be binary non-decreasing functions. Then $\Gamma(\boldsymbol{S}) = \Gamma(f_1(\boldsymbol{T}), \ldots, f_m(\boldsymbol{T}))$ and $\Delta(\boldsymbol{S}) = \Delta(f_1(\boldsymbol{T}), \ldots, f_m(\boldsymbol{T}))$ are non-decreasing functions of $\boldsymbol{T}$ as well. Hence, by Definition 8.1.1 it follows that:

$$\mathrm{Cov}(\Gamma(\boldsymbol{S}), \Delta(\boldsymbol{S})) = \mathrm{Cov}(\Gamma(f_1(\boldsymbol{T}), \ldots, f_m(\boldsymbol{T})), \Delta(f_1(\boldsymbol{T}), \ldots, f_m(\boldsymbol{T}))) \geq 0.$$

Hence, we conclude that $S_1, \ldots, S_m$ are associated as well.

Finally, we prove (iv). Thus, we let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be two vectors of associated random variables, and assume that $\boldsymbol{X}$ and $\boldsymbol{Y}$ are independent of each other. Moreover, we assume that $\Gamma = \Gamma(\boldsymbol{X}, \boldsymbol{Y})$ and $\Delta = \Delta(\boldsymbol{X}, \boldsymbol{Y})$ are binary and non-decreasing functions in both $\boldsymbol{X}$ and $\boldsymbol{Y}$. Then by Lemma 8.1.3 we have:

$$\mathrm{Cov}(\Gamma, \Delta) = E[\mathrm{Cov}(\Gamma, \Delta|\boldsymbol{X})] + \mathrm{Cov}[E(\Gamma|\boldsymbol{X}), E(\Delta|\boldsymbol{X})]. \tag{8.1}$$

We then note that for any $\boldsymbol{x}$, $\Gamma(\boldsymbol{x}, \boldsymbol{Y})$ and $\Delta(\boldsymbol{x}, \boldsymbol{Y})$ are binary non-decreasing functions of $\boldsymbol{Y}$. Hence, we must have:

$$\mathrm{Cov}(\Gamma(\boldsymbol{X}, \boldsymbol{Y}), \Delta(\boldsymbol{X}, \boldsymbol{Y})|\boldsymbol{X} = \boldsymbol{x}) = \mathrm{Cov}(\Gamma(\boldsymbol{x}, \boldsymbol{Y}), \Delta(\boldsymbol{x}, \boldsymbol{Y})) \geq 0, \text{ for all } \boldsymbol{x},$$

where the equality follows since $\boldsymbol{Y}$ is independent of $\boldsymbol{X}$, while the inequality follows since $\boldsymbol{Y}$ is associated. This implies that:

$$E[\mathrm{Cov}(\Gamma, \Delta|\boldsymbol{X})] \geq 0. \tag{8.2}$$

Moreover, $E[\Gamma(\boldsymbol{x}, \boldsymbol{Y})]$ and $E[\Delta(\boldsymbol{x}, \boldsymbol{Y})]$ are non-decreasing (but not necessarily binary) functions of $\boldsymbol{x}$. Hence, since $\boldsymbol{X}$ is associated, it follows by Theorem 8.1.2 that:

$$\mathrm{Cov}[E(\Gamma|\boldsymbol{X}), E(\Delta|\boldsymbol{X})] \geq 0. \tag{8.3}$$

Note that since Γ and Δ are binary, we must have that $E(\Gamma|\boldsymbol{X}) \in [0, 1]$ and $E(\Delta|\boldsymbol{X}) \in [0, 1]$ with probability one. Thus, obviously $\mathrm{Cov}[E(\Gamma|\boldsymbol{X}), E(\Delta|\boldsymbol{X})]$ exists.

Combining (8.1), (8.2) and (8.3) implies that:

$$\mathrm{Cov}(\Gamma, \Delta) \geq 0.$$

Thus, we conclude that $(\boldsymbol{X}, \boldsymbol{Y})$ is associated. □

Theorem 8.1.5 *Let $T_1, \ldots, T_n$ be independent random variables. Then, they are also associated.*

Proof: The proof is by induction on n. The result obviously holds for $n = 1$ by Theorem 8.1.4 (ii). Assume that the theorem holds for $n = m - 1$. That is, $\{T_1, \ldots, T_{m-1}\}$ is a set of associated random variables. Moreover, by Theorem 8.1.4 (ii), $\{T_m\}$ is associated as well. By the assumption, these two sets are independent. Hence, it follows from Theorem 8.1.4 (iv) that their union $\{T_1, \ldots, T_{m-1}, T_m\}$ is a set of associated random variables. Thus, the result is proved by induction. □

At the completely opposite end of the scale, we have the following result:

Theorem 8.1.6 *Let $T_1, \ldots, T_n$ be completely positively dependent random variables, i.e.,*

$$P(T_1 = T_2 = \ldots = T_n) = 1.$$

Then they are associated.

Proof: Let Γ, Δ be binary functions which are non-decreasing in each argument and let $\boldsymbol{T} = (T_1, \ldots, T_n)$ and $\boldsymbol{T}_1 = (T_1, \ldots, T_1)$. By the assumption it follows that $\boldsymbol{T}$ and $\boldsymbol{T}_1$ must have the same distribution. Hence, we get that:

$$\mathrm{Cov}(\Gamma(\boldsymbol{T}), \Delta(\boldsymbol{T})) = \mathrm{Cov}(\Gamma(\boldsymbol{T}_1), \Delta(\boldsymbol{T}_1)) \geq 0$$

where the final inequality follows by Theorem 8.1.4 (ii) and Definition 8.1.1. □

In the next section, we will establish upper and lower bounds for the system reliability. In order to do this, the following theorem is useful:

Theorem 8.1.7 *Let $X_1, \ldots, X_n$ be the associated or independent component state variables of a monotone system (C, ϕ). Moreover, let the minimal path series structures of the system be $(P_1, \rho_1), \ldots, (P_p, \rho_p)$, where:*

$$\rho_j(\boldsymbol{X}^{P_j}) = \coprod_{i \in P_j} X_i, \quad j = 1, \ldots, p. \tag{8.4}$$

Then, $\rho_1, \ldots, \rho_p$ are associated. Similarly, let the minimal cut parallel structures of the system be $(K_1, \kappa_1), \ldots, (K_k, \kappa_k)$, where:

$$\kappa_j(\boldsymbol{X}^{K_j}) = \coprod_{i \in K_j} X_i, \quad j = 1, \ldots, k. \tag{8.5}$$

Then, $\kappa_1, \ldots, \kappa_k$ are associated.

Proof: The result follows since $\rho_1, \ldots, \rho_p$ in (8.4) and $\kappa_1, \ldots, \kappa_k$ in (8.5) are non-decreasing functions of associated random variables, and hence associated by Theorem 8.1.4, (iii). □

The following result extends Theorem 8.1.4, (iii) to non-increasing functions as well.

Theorem 8.1.8 *Let $\boldsymbol{T} = (T_1, \ldots, T_n)$ be associated, and let:*

$$U_i = g_i(\boldsymbol{T}), \quad i = 1, \ldots, m,$$

where g_i, $i = 1, \ldots, m$ are non-increasing functions. Then, $\boldsymbol{U} = (U_1, \ldots, U_m)$ is associated.

Proof: Let Γ, Δ be binary non-decreasing functions, and introduce $\boldsymbol{g}(\boldsymbol{T}) = (g_1(\boldsymbol{T}), \ldots, g_m(\boldsymbol{T}))$. Thus, $\boldsymbol{U} = \boldsymbol{g}(\boldsymbol{T})$. We then introduce

$$\bar{\Gamma}(\boldsymbol{T}) = 1 - \Gamma(\boldsymbol{g}(\boldsymbol{T})) = 1 - \Gamma(\boldsymbol{U})$$
$$\bar{\Delta}(\boldsymbol{T}) = 1 - \Delta(\boldsymbol{g}(\boldsymbol{T})) = 1 - \Delta(\boldsymbol{U})$$

It follows directly that $\bar{\Gamma}$ and $\bar{\Delta}$ are binary and non-decreasing in T_i, $i = 1, \ldots, n$. Hence, by the assumption that $\boldsymbol{T}$ is associated, it follows that:

$$\begin{aligned}\mathrm{Cov}(\Gamma(\boldsymbol{U}), \Delta(\boldsymbol{U}))) &= \mathrm{Cov}(1 - \bar{\Gamma}(\boldsymbol{T}), 1 - \bar{\Delta}(\boldsymbol{T})) \\ &= \mathrm{Cov}(1,1) + \mathrm{Cov}(1, -\bar{\Delta}(\boldsymbol{T})) + \mathrm{Cov}(-\bar{\Gamma}(\boldsymbol{T}), 1) \\ &\quad + \mathrm{Cov}(-\bar{\Gamma}(\boldsymbol{T}), -\bar{\Delta}(\boldsymbol{T})) \\ &= \mathrm{Cov}(\bar{\Gamma}(\boldsymbol{T}), \bar{\Delta}(\boldsymbol{T})) \geq 0.\end{aligned}$$

Hence, we conclude that $\boldsymbol{U}$ is associated. □

In order to check whether two random variables are associated, the following result can sometimes be used. It states that for two binary random variables, association and non-negative covariance are equivalent.

Theorem 8.1.9 *Let X and Y be two binary random variables. Then, X and Y are associated if and only if*

$$\mathrm{Cov}(X, Y) \geq 0.$$

Proof: Assume first that X and Y are associated. We may then choose $\Gamma(X, Y) = X$ and $\Delta(X, Y) = Y$. Since obviously Γ and Δ are binary and non-decreasing functions, it follows from Definition 8.1.1 that $\mathrm{Cov}(X, Y) = \mathrm{Cov}(\Gamma, \Delta) \geq 0$.

We then assume conversely that $\mathrm{Cov}(X, Y) \geq 0$. If can prove that this implies that $\mathrm{Cov}(\Gamma, \Delta) \geq 0$ for all binary non-decreasing functions, the result is proved. Since there are only two variables in this case, X and Y, there are only a very limited number of choices for Γ and Δ. In fact, the only choices are:

$$\Gamma_1 \equiv 0, \quad \Gamma_2 = X \cdot Y, \quad \Gamma_3 = X, \quad \Gamma_4 = Y, \quad \Gamma_5 = X \amalg Y, \quad \Gamma_6 \equiv 1.$$

Moreover, we have the following partial ordering:

$$\Gamma_1 \leq \Gamma_2 \leq \left\{ \begin{array}{c} \Gamma_3 \\ \Gamma_4 \end{array} \right\} \leq \Gamma_5 \leq \Gamma_6.$$

Assume first that Γ and Δ from the set $\{\Gamma_1, \ldots, \Gamma_6\}$ such that $\Gamma(X,Y) \leq \Delta(X,Y)$. We then have:

$$\begin{aligned} \mathrm{Cov}(\Gamma, \Delta) &= E(\Gamma \cdot \Delta) - E(\Gamma) \cdot E(\Delta) \\ &= E(\Gamma) - E(\Gamma) \cdot E(\Delta) = E(\Gamma)[1 - E(\Delta)] \geq 0. \end{aligned}$$

The only possibility left is $\Gamma = \Gamma_3 = X$ and $\Delta = \Gamma_4 = Y$. However, in this case we get that:

$$\mathrm{Cov}(\Gamma, \Delta) = \mathrm{Cov}(X, Y) \geq 0,$$

where the last inequality follows by the assumption. Hence, we conclude that $\mathrm{Cov}(\Gamma, \Delta) \geq 0$ for all binary non-decreasing functions, and thus the result is proved. □

8.2 Upper and lower bounds for the reliability of monotone systems

As mentioned, it is often the case that we are unable to compute the exact reliability of a system. This may be the case for large, complex systems where the computations simply take too much time, but also for systems where the components are not independent (recall that we assumed independence of the components throughout Section 4). In this section, we derive several different upper and lower bounds for the system reliability. These bounds can be useful whenever computing the exact reliability is impossible or too computationally costly.

Note that in this section, we will not always assume that the component state variables are independent. However, to get useful bounds, we still need to assume something about the dependencies between the components. In most of the results of this section, this assumption will be that the component state variables are associated (see Section 8.1).

The following theorem is a key result for deriving reliability bounds.

Theorem 8.2.1 *Let $T_1, \ldots, T_n$ be associated random variables such that $0 \leq T_i \leq 1$, $i = 1, \ldots, n$. Then,*

$$E[\prod_{i=1}^{n} T_i] \geq \prod_{i=1}^{n} E[T_i] \tag{8.6}$$

$$E[\coprod_{i=1}^{n} T_i] \leq \coprod_{i=1}^{n} E[T_i] \tag{8.7}$$

Proof: We note that since $0 \leq T_i \leq 1$, both T_i and $S_i = 1 - T_i$ are non-negative random variables, $i = 1, \ldots, n$. Hence, the product functions $\prod_{i=1}^{n} T_i$ and $\prod_{i=1}^{n} S_i$ are both non-decreasing in each argument.

By using Theorem 8.1.2, we find:

$$E[\prod_{i=1}^{n} T_i] - E[T_1]E[\prod_{i=2}^{n} T_i] = \mathrm{Cov}(T_1, \prod_{i=2}^{n} T_i) \geq 0,$$

since the product function is non-decreasing in each argument because $T_i \geq 0$, $i = 2, \ldots, n$. This implies that:

$$E[\prod_{i=1}^{n} T_i] \geq E[T_1]E[\prod_{i=2}^{n} T_i].$$

By repeated use of this inequality, we get (8.6).

From Theorem 8.1.8, $S_1, \ldots, S_n$ are associated random variables. Moreover, $0 \leq S_i \leq 1$, $i = 1, \ldots, n$, so we can apply (8.6) to these variables. From this it follows that:

$$\begin{aligned} E[\coprod_{i=1}^{n} T_i] &= 1 - E[\prod_{i=1}^{n}(1 - T_i)] = 1 - E[\prod_{i=1}^{n} S_i] \\ &\leq 1 - \prod_{i=1}^{n} E(S_i) = 1 - \prod_{i=1}^{n}(1 - E[T_i]) \\ &= \coprod_{i=1}^{n} E[T_i], \end{aligned}$$

so (8.7) is proved as well.

□

Theorem 8.2.1 has an important interpretation: If we apply the theorem to the binary component state variables $X_1, \ldots, X_n$, (8.6) says that for a series structure of associated components, an incorrect assumption of independence will lead to an *underestimation* of the system reliability. Correspondingly, (8.7) says that for a parallel structure, an incorrect assumption of independence between the components will lead to an *overestimation* of the system reliability. Clearly, overestimating the system reliability can have serious consequences in applications, and should be avoided. Since most systems are not purely series or purely parallel, we conclude that for an arbitrary structure, we *cannot say for certain what the consequences of an incorrect assumption of independence will be.* This means that in any application where we do not know for sure that the components are independent, simply assuming independence in order to use one of the methods in Section 4 to compute the exact system reliability can be dangerous. Fortunately, it is still possible to obtain bounds on the system reliability.

In the next results we interpret T_i, $i = 1, \ldots, n$ as the lifetimes of components in a binary monotone system.

Theorem 8.2.2 *Let $T_1, \ldots, T_n$ be non-negative associated random variables. Then, for all t_i, $i = 1, \ldots, n$:*

$$P[\cap_{i=1}^{n}(T_i > t_i)] \geq \prod_{i=1}^{n} P[T_i > t_i], \tag{8.8}$$

$$P[\cap_{i=1}^{n}(T_i \leq t_i)] \geq \prod_{i=1}^{n} P[T_i \leq t_i]. \tag{8.9}$$

Proof: Interpreting T_i as the lifetime of the ith component in a binary monotone system, we may introduce the corresponding binary component state process $\{X_i(t)\}$, $i = 1, \ldots, n$, where:

$$X_i(t) = \begin{cases} 1, & t < T_i \\ 0, & t \geq T_i. \end{cases}$$

In Figure 8.1 we have plotted $X_i(t)$ as a function of $t \geq 0$ for a given value of T_i. The plot shows that $X_i(t)$ is a non-increasing function of t. For a fixed point of time t we may alternatively view $X_i(t)$ as a function of the lifetime T_i. The resulting plot is shown in Figure 8.2. From this plot we

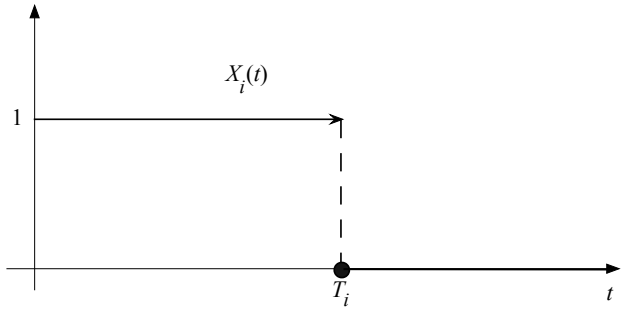

Figure 8.1: The binary component state process $X_i(t)$ plotted as a function of the time $t \geq 0$.

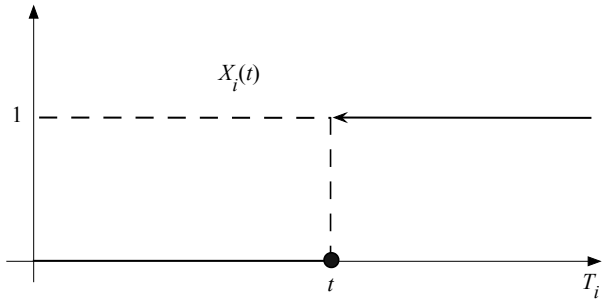

Figure 8.2: The binary component state process $X_i(t)$ plotted as a function of the lifetime T_i for a given $t \geq 0$.

see that $X_i(t)$ is a non-decreasing function of T_i. Hence, by Theorem 8.1.4, $X_1(t), \ldots, X_n(t)$ are associated for all $t \geq 0$.

Hence, by applying (8.6):

$$\begin{aligned} P[\cap_{i=1}^{n}(T_i > t_i)] &= P[\cap_{i=1}^{n}(X_i(t_i) = 1)] \\ &= P[\prod_{i=1}^{n} X_i(t_i) = 1] = E[\prod_{i=1}^{n} X_i(t_i)] \\ &\geq \prod_{i=1}^{n} E[X_i(t_i)] = \prod_{i=1}^{n} P[X_i(t_i) = 1] \\ &= \prod_{i=1}^{n} P[T_i > t_i], \end{aligned}$$

so (8.8) is proved. The proof of (8.9) is similar by using (8.7) instead of (8.6) and left as an exercise to the reader.

□

Note that the left hand side of (8.9) is the joint cumulative distribution of $T_1, \ldots, T_n$, while the right hand side is the corresponding cumulative distribution in the case where $T_1, \ldots, T_n$ are independent. As a corollary to Theorem 8.2.2, we have:

Corollary 8.2.3 *Let $T_1, \ldots, T_n$ be non-negative associated random variables. Then:*

$$P(\min_{1\leq i\leq n} T_i > t) \geq \prod_{i=1}^{n} P(T_i > t), \tag{8.10}$$

$$P(\max_{1\leq i\leq n} T_i > t) \leq \coprod_{i=1}^{n} P(T_i > t). \tag{8.11}$$

Proof: This follows from Theorem 8.2.2 by letting $t_i = t$ for $i = 1, \ldots, n$. Alternatively, this can also be proved by applying Theorem 8.2.1 to the binary component states at time t, since $\min_{1\leq i\leq n} T_i$ is the lifetime of a series system and $\max_{1\leq i\leq n} T_i$ is the lifetime of a parallel system. □

Now, we are ready to derive our first (crude) bounds of the reliability of a monotone system. In words, these bounds say that for any monotone system, the reliability is lower bounded by the reliability of a series system and upper bounded by the reliability of a parallel system. This corresponds

to our previous observation (see the comments after Theorem 2.2.4) that the series system is the *worst* non-trivial monotone system, while the parallel system is the *best*.

Theorem 8.2.4 *Let $X_1, \ldots, X_n$ be the component state variables of a non-trivial binary monotone structure (C, ϕ) with component reliabilities $p_1, \ldots, p_n$, and assume that $X_1, \ldots, X_n$ are associated. Then we have:*

$$\prod_{i=1}^{n} p_i \leq h \leq \coprod_{i=1}^{n} p_i.$$

Proof: We get:

$$\begin{aligned}
\prod_{i=1}^{n} p_i &= \prod_{i=1}^{n} E[X_i] \\
&\leq E[\prod_{i=1}^{n} X_i] \leq E[\phi(\boldsymbol{X})] \\
&\leq E[\coprod_{i=1}^{n} X_i] \leq \coprod_{i=1}^{n} E[X_i] \\
&= \coprod_{i=1}^{n} p_i.
\end{aligned}$$

Here, the first inequality follows from (8.6), the second and third inequalities from Theorem 2.2.4 and the fourth inequality follows from (8.7). □

As mentioned, these bounds are very crude, and usually not of much practical use. Luckily, we are always able to establish better bounds than those of Theorem 8.2.4 by including information about the minimal path and cut sets of the system. These improved bounds are a corollary to the following result:

Theorem 8.2.5 *Consider a monotone structure (C, ϕ) with minimal path sets $P_1, \ldots, P_p$, and minimal cut sets $K_1, \ldots, K_k$. Then we have:*

$$\max_{1 \leq j \leq p} P[\min_{i \in P_j} X_i = 1] \leq h \leq \min_{1 \leq j \leq k} P[\max_{i \in K_j} X_i = 1].$$

Proof: From (3.8) and (3.10), we get

$$\min_{i\in P_r} X_i \leq \max_{1\leq r\leq p} \min_{i\in P_r} X_i = \phi(\boldsymbol{X}) = \min_{1\leq s\leq k} \max_{i\in K_s} X_i \leq \max_{i\in K_s} X_i,$$

for all $r = 1, \ldots, p$ and all $s = 1, \ldots, k$. This implies that:

$$P(\min_{i\in P_r} X_i = 1) \leq h \leq P(\max_{i\in K_s} X_i = 1)$$

for all $r = 1, \ldots, p$ and all $s = 1, \ldots, k$. This proves the theorem. □

In Theorem 8.2.5, we made no assumptions about the dependence between the components. In order to obtain an explicit expression for the lower bound, we need to compute $P(\min_{i\in P_r} X_i = 1)$, $j = 1, \ldots, p$. Similarly, to obtain an explicit expression for the upper bound, we need to compute $P(\max_{i\in K_r} X_i = 1)$, $j = 1, \ldots, k$. This is difficult without making further assumptions about the joint distribution of the component state variables. In the following corollary we make the assumption that the component state variables are associated. This enables us to obtain explicit bounds.

Corollary 8.2.6 *Consider the same monotone system (C, ϕ) as in Theorem 8.2.5. Moreover, assume that the component state variables are associated with component reliabilities $p_1, \ldots, p_n$. Then we have:*

$$\max_{1\leq j\leq p} \prod_{i\in P_j} p_i \leq h \leq \min_{1\leq j\leq k} \coprod_{i\in K_j} p_i.$$

Proof: The result follows immediately from Theorem 8.2.5 by using (8.6) and (8.7) because:

$$\prod_{i\in P_j} p_i \leq E[\prod_{i\in P_j} X_i] = P[\min_{i\in P_j} X_i = 1]$$

and

$$\coprod_{i\in K_j} p_i \geq E[\coprod_{i\in K_j} X_i] = P[\max_{i\in K_j} X_i = 1].$$

□

The assumptions in Theorem 8.2.4 and Corollary 8.2.6 are the same. However, the bounds in Corollary 8.2.6 are obviously better than those in Theorem 8.2.4 because:

$$\prod_{i=1}^{n} p_i = \prod_{i\in P_j} p_i \prod_{i\notin P_j} p_i \leq \prod_{i\in P_j} p_i \leq \max_{1\leq j\leq p} \prod_{i\in P_j} p_i$$

since $p_i \in [0,1]$ for $i = 1, \ldots, n$. A similar argument proves that the upper bound in Corollary 8.2.3 is better than the one in Theorem 8.2.4.

If we in addition to the assumptions in Corollary 8.2.6 assume that the components are independent, we get bounds which are sometimes better than those in Corollary 8.2.6 and sometimes worse. These bounds are a corollary to the following theorem:

Theorem 8.2.7 *Let $X_1, \ldots, X_n$ be the associated component states of a binary monotone system (C, ϕ) with minimal path series structures $(P_1, \rho_1), \ldots, (P_p, \rho_p))$ and minimal cut parallel structures $(K_1, \kappa_1), \ldots, (K_k, \kappa_k))$. Then,*

$$\prod_{j=1}^{k} P(\kappa_j(\boldsymbol{X}^{K_j}) = 1) \leq h \leq \coprod_{j=1}^{p} P(\rho_j(\boldsymbol{X}^{P_j}) = 1). \tag{8.12}$$

Proof: From the comments after Theorem 8.1.4, it follows from Theorem 8.1.4 (iii) that the minimal path series structures, and the minimal cut parallel structures, are associated. Hence, we get that:

$$\begin{aligned}
\prod_{j=1}^{k} P(\kappa_j(\boldsymbol{X}^{K_j}) = 1) &\leq E[\prod_{j=1}^{k} \kappa_j(\boldsymbol{X}^{K_j}] \\
&= h \\
&= E[\coprod_{j=1}^{p} \rho_j(\boldsymbol{X}^{P_j})] \\
&\leq \coprod_{j=1}^{p} P(\rho_j(\boldsymbol{X}^{P_j}) = 1),
\end{aligned}$$

where the first inequality follows from (8.6), the first and second equalities follow from (3.10) and (3.8) respectively and the final inequality follows from (8.7). □

The problem with Theorem 8.2.7 is the same as the problem with Theorem 8.2.5. Without having explicit expressions for the reliabilities of the path series structures and cut parallel structures the bounds are difficult to use. However, by assuming independent component state variables, we can derive the following corollary.

Corollary 8.2.8 *Let (C, ϕ) be a binary monotone system of independent component states and where the component reliabilities are $p_1, \ldots, p_n$. Let $P_1, \ldots, P_n$ and $K_1, \ldots, K_n$ be respectively the minimal path and cut sets of the system. Then we have:*

$$\prod_{j=1}^{k} \coprod_{i \in K_j} p_i \leq h(\boldsymbol{p}) \leq \coprod_{j=1}^{p} \prod_{i \in P_j} p_i. \tag{8.13}$$

Proof: For independent components, the lower bound in (8.13) is equal to the one in (8.12) because

$$P(\kappa_j(\boldsymbol{X}^{K_j}) = 1) = E[\coprod_{i \in K_j} X_i] = \coprod_{i \in K_j} p_i.$$

The upper bound in (8.13) is proved in the same way. □

We conclude this section with an example.

Example 8.2.9 *Consider a 3-out-of-4 system with $p_i = p$, $i = 1, 2, 3, 4$. We assume that all the component state variables are independent. We shall compare the bounds from Corollary 8.2.6 to those from Corollary 8.2.8.*

The minimal path sets for a 3-out-of-4 system are:

$$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\},$$

and the minimal cut sets are:

$$\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}.$$

In the calculations below we will repeatedly use the formula for the reliability of a parallel system of two components with independent component state variables. Denoting the reliability of this system by h_2, and assuming that the components have reliability q, we have:

$$h_2(q) = q \amalg q = 1 - (1 - q)(1 - q) = 2q - q^2.$$

We denote the lower and upper bounds in Corollary 8.2.6 by $l_1(p)$ and $u_1(p)$ respectively. Since all the components have the same reliability, these bounds

are given by:

$$l_1(p) = \max_{1\leq j\leq 4} \prod_{i\in P_j} p = \max_{1\leq j\leq 4} p^3 = p^3,$$
$$u_1(p) = \min_{1\leq j\leq 6} \coprod_{i\in K_j} p = \min_{1\leq j\leq 6} h_2(p) = 2p - p^2.$$

Similarly, we denote the bounds in Corollary 8.2.8 by $l_2(p)$ and $u_2(p)$ respectively. Again since all the components have the same reliability, and since we have assumed that the component state variables are independent, these bounds are given by:

$$l_2(p) = \prod_{j=1}^{6} \coprod_{i\in K_j} p = \prod_{j=1}^{6} h_2(p) = (2p - p^2)^6,$$
$$u_2(p) = \coprod_{j=1}^{4} \prod_{i\in P_j} p = h_2(p^3)\amalg h_2(p^3) = 2(2p^3 - p^6) - (2p^3 - p^6)^2.$$

From Section 2.5 we know that the reliability of the 3-out-of-4 system is given by:

$$h(p) = \sum_{i=3}^{4} \binom{4}{i} p^i (1-p)^{n-i} = 4p^3(1-p) + p^4.$$

The bounds $l_1(p)$, $u_1(p)$, $l_2(p)$, $u_2(p)$, as well as the reliability function $h(p)$, are plotted in Figure 8.3.

From Figure 8.3, we see that in this case, the bounds from Corollary 8.2.8 are better than those from Corollary 8.2.6. However, note that the lower bound from Corollary 8.2.6 is actually best for small values of p, while the opposite is true for larger p-values. Correspondingly, the upper bound from Corollary 8.2.6 is best for large p-values, while the upper bound from Corollary 8.2.8 is best for smaller values of p.

By defining a new lower bound as the largest of the two lower bounds from Corollary 8.2.6 and Corollary 8.2.8, and a new upper bound as the smallest of the two upper bounds from the same corollaries, we get a new set of bounds which is better than the previous ones. It is also possible to improve these bounds further by using a modular decomposition of the system at hand. We will not go into details on this, but refer to Natvig [50] for more on this topic.

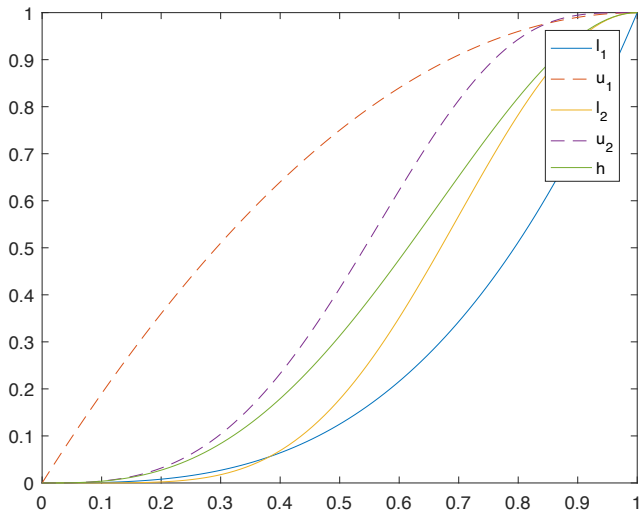

Figure 8.3: Illustration of the bounds in Example 8.2.9.

8.3 Exercises

Exercise 1. Prove (8.9) of Theorem 8.2.2 by using equation (8.7).

Exercise 2. Prove that the upper bound in Corollary 8.2.6 is better than the one in Theorem 8.2.4.

Exercise 3. Prove the upper bound of Corollary 8.2.8.

Exercise 4. Prove the upper bound in Corollary 8.2.6 by applying the lower bound on the dual structure function ϕ^D.

Exercise 5. Prove the upper bound in Corollary 8.2.8 by applying the lower bound on the dual structure function ϕ^D.

Exercise 6. Consider the network in Figure 8.4 consisting of independent components with component reliabilities p.

a) What is the reliability $h(p)$ for this system?

b) For $p \in [0, 1]$, write a program to compute the reliability $h(p)$. Plot $h(p)$.

c) In the same plot, illustrate the bounds from Corollary 8.2.6 and 8.2.8. Comment on the result.

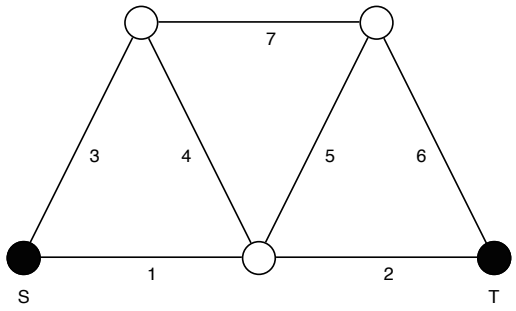

Figure 8.4: Illustration of the network for Exercise 8.2.6.

d) Is it possible to improve these bounds further?

Chapter 9

Applications

In this chapter, we study two different applications of the theory presented in the previous chapters. In Section 9.1, we show how to perform a reliability analysis of two different kinds of fishing boat engines, while in Section 9.2, we compute the reliability of a network for transmission of electronic pulses.

9.1 Case study: Reliability analysis and comparison of two fishing boat engines

This section is based on a real-life example provided by a Norwegian ship certification agency.

We will perform a reliability analysis and comparison of two different engines for fishing boats. In the following, we say that the boat engine is functioning if and only if the propeller and the power supply are functioning.

In 1977 the certification agency only approved one of the two engines we will study. The critical parts of this engine are a propulsion engine, an auxiliary engine and a hydraulic operated clutch. In Natvig [50], you can find a detailed illustration of the engine. We omit the details, and instead focus on the parts which are essential to our reliability analysis.

We define the following fault-events which may happen in this engine:

$$\begin{aligned} F1 &: \text{The propulsion engine fails} \\ K1 &: \text{The hydraulic clutch fails} \\ H1 &: \text{The auxiliary engine fails} \end{aligned} \tag{9.1}$$

Corresponding to the engine, we can make a fault tree to illustrate what

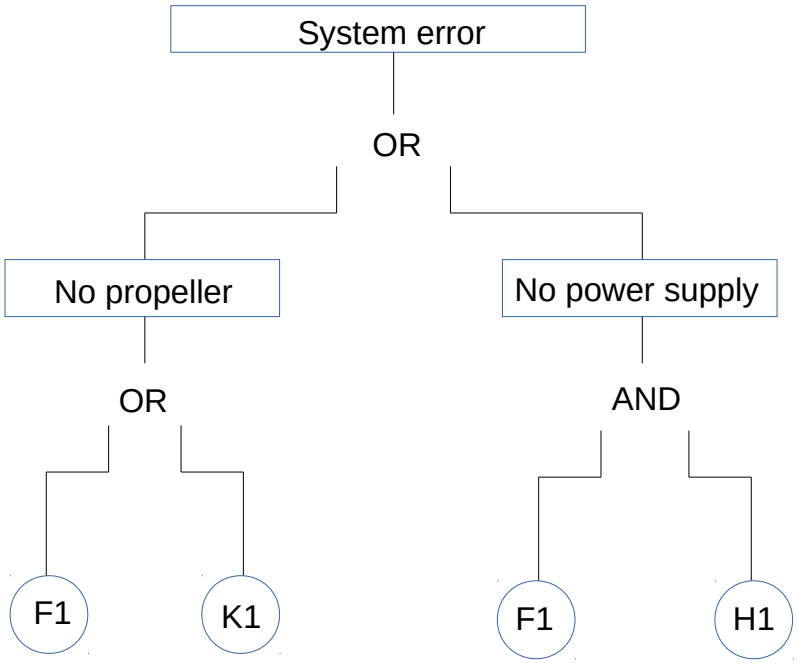

Figure 9.1: Fault tree for the first boat engine.

The "OR" part of the fault tree means that at least one of the faults under the "OR" must occur for for the above fault to occur. For instance, in order for there to be a system failure, either the propeller is not functioning or there is no power supply. Correspondingly, the "AND" means that both of the faults under the "AND" must occur for the fault to occur. That is, in order for there to be no power supply, both $F1$ and $H1$ must happen. A more detailed description of fault trees can be found in Barlow and Proschan [6].

The auxiliary engine is only used when the boat is at harbour, while the propulsion engine and the clutch are used while the boat is running at sea. The time at sea per year is 3000 hours, while the harbour time is 2000 hours. Hence, the run time for the propulsion engine and the clutch is 3000 hours, while the run time for the auxiliary engine is 2000 hours (per year). We

assume that the lifetime distributions of the components are exponential. Due to the memoryless property of the exponential distribution and the fact that we are considering mechanical components, this may be unrealistic. The Weibull distribution (which takes aging of components into account) would be preferable. However, to simplify, we assume exponential lifetimes.

Det Norske Veritas supplied point estimates for the failure rates, measured in failures per hour run. For a diesel engine (such as the propulsion engine and the auxiliary engine), this failure rate is estimated to be $2.96 \cdot 10^{-4}$. For a magnetic clutch, the estimate is $3.88 \cdot 10^{-5}$, while for a mechanic clutch it is $6.0 \cdot 10^{-6}$ (we will use this later in the section). Due to lack of data for hydraulic clutches, the error estimate for the magnetic clutch was used as an approximation.

Based on this, we can compute the following estimates for the probability of the basic fault events in (9.1) happening in one year:

$$\begin{aligned} P(F1) &= 1 - \exp(-2.96 \cdot 10^{-4} \cdot 3000) = 0.58852, \\ P(K1) &= 1 - \exp(-3.88 \cdot 10^{-5} \cdot 3000) = 0.10988, \\ P(H1) &= 1 - \exp(-2.96 \cdot 10^{-4} \cdot 2000) = 0.44678. \end{aligned} \tag{9.2}$$

This gives the following estimates:

$$\begin{aligned} P(\text{propeller failure in a year}) &= 1 - (1 - P(F1)) \cdot (1 - P(K1)) = 0.63373, \\ P(\text{no power supply in a year}) &= P(H1) \cdot P(F1) = 0.26294. \end{aligned}$$

From this, the certification agency concluded that:

$$P(\text{system failure in a year}) = 1-(1-0.63373)(1-0.26294) = 0.73004. \tag{9.3}$$

However, this method is incorrect because it does not take into account the dependence which the basic fault $F1$ creates in the fault tree because it occurs twice. In order to derive the correct reliability (or equivalently, the probability of system failure within a year), we introduce the following binary

variables:

$$X_1 = \begin{cases} 1 & \text{if } F1 \text{ does } \textit{not} \text{ occur within a year} \\ 0 & \text{otherwise} \end{cases} \tag{9.4}$$

$$X_2 = \begin{cases} 1 & \text{if } K1 \text{ does } \textit{not} \text{ occur within a year} \\ 0 & \text{otherwise} \end{cases}$$

$$X_3 = \begin{cases} 1 & \text{if } H1 \text{ does } \textit{not} \text{ occur within a year} \\ 0 & \text{otherwise} \end{cases}$$

The fault tree 9.1 is equivalent to reliability block diagram shown in Figure 9.2. In the reliability block diagram component 1 with state variable X_1 corresponds to the propulsion engine, component 2 with state variable X_2 corresponds to the clutch, while component 3 with state variable X_3 corresponds to the auxiliary engine.

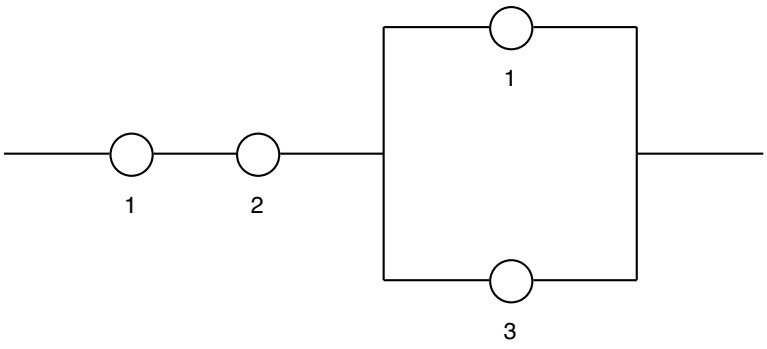

Figure 9.2: System equivalent to the fault tree.

From Figure 9.2, we see that component 3, i.e., the auxiliary engine, is an irrelevant component. It does not matter whether the auxiliary engine is working for the system to be working. This can also be seen by deriving the

structure function of the system in Figure 9.2:

$$\begin{aligned}\phi_1(\boldsymbol{X}) &= X_1X_2(1-(1-X_1)(1-X_3))\\ &= X_1X_2 + X_1X_2X_3 - X_1X_2X_3\\ &= X_1X_2.\end{aligned}$$

Hence, we get the following correct estimate of the probability of system error within a year.

$$\begin{aligned}&P(\text{System failure within a year})\\ &\quad = P(\phi_1(\boldsymbol{X}) = 0) = 1 - P(\phi_1(\boldsymbol{X}) = 1)\\ &\quad = 1 - (1 - P(F1))(1 - P(K1)) = 0.63373\end{aligned} \tag{9.5}$$

By comparing this to the estimate in equation (9.3), we see that the certification agency ended up with a failure probability which is larger than the actual one.

As mentioned in the beginning of the section, we would like to perform a reliability analysis for two different engines, and compare them. Let us now turn to the second engine, which is a two-engine system consisting of right and left propulsion engines, right and left hydraulic clutches and a mechanical clutch. Again, we will omit a detailed illustration of the system (this can be found in Natvig [50], Figure 5.1.4). Instead, we focus on the corresponding fault tree shown in Figure 9.3 consisting of the following basic fault-events:

$$\begin{aligned}F2:&\ \text{Error in the left propulsion engine}\\ K2:&\ \text{Error on left hydraulic clutch}\\ F3:&\ \text{Error in the right propulsion engine}\\ K3:&\ \text{Error on right hydraulic clutch}\\ K4:&\ \text{Error on mechanical clutch}\end{aligned} \tag{9.6}$$

For this system, both left and right engines are used at sea. At the harbour, only the right engine is used (this runs the generator which produces power when docking). Hence, the run time per year for the right engine is 5000 hours, and for the mechanical clutch it is 2000 hours (this is only used when docking). For the left propulsion engine and the two hydraulic clutches, the run time per year is 3000 hours, as they are only used at sea.

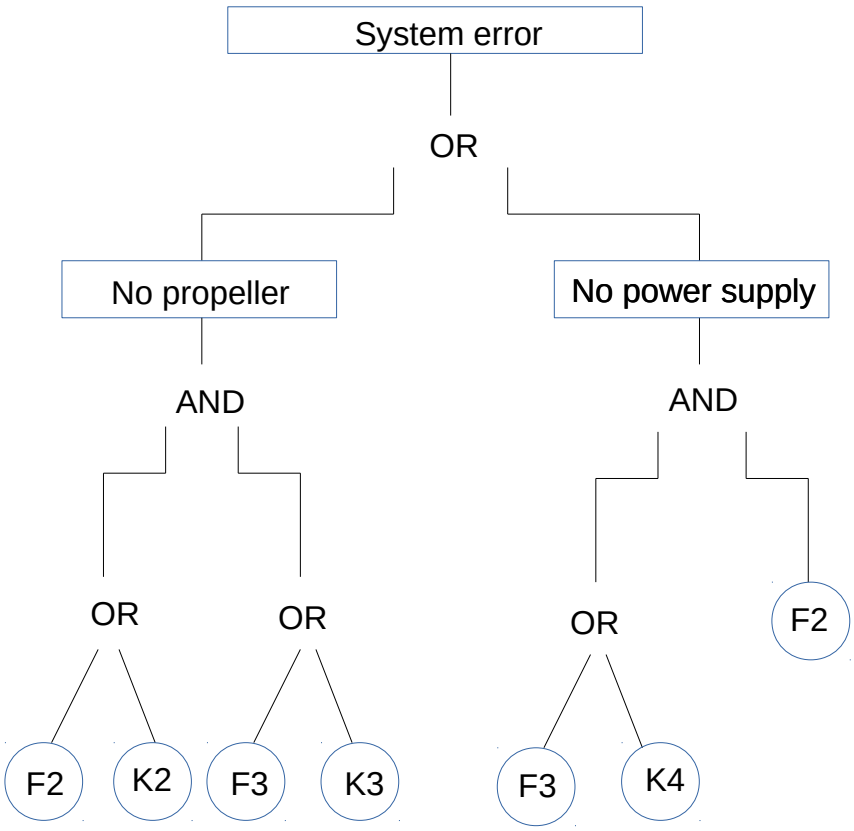

Figure 9.3: Fault tree for the second boat engine.

Based on the point estimates from Det Norske Veritas (see the comments before equation (9.2)), we compute the following estimates for the probabilities of the basic fault-events:

$$\begin{aligned} P(F2) &= P(F1) = 0.58852, \\ P(K2) &= P(K3) = P(K1) = 0.10988, \\ P(F3) &= 1 - \exp(-2.96 \cdot 10^{-4} \cdot 5000) = 0.77236, \\ P(K4) &= 1 - \exp(-6 \cdot 10^{-6} \cdot 2000) = 0.01192. \end{aligned} \tag{9.7}$$

This leads to the following estimates:

$$\begin{aligned}
&P(\text{propeller failure in a year}) \\
&\quad = [1-(1-P(F2))(1-P(K2))][1-(1-P(F3))(1-P(K3))] \\
&\quad = 0.63372 \cdot 0.79737 \\
&\quad = 0.50532,
\end{aligned}$$

$$\begin{aligned}
&P(\text{no power supply in a year}) \\
&\quad = [1-(1-P(F3))(1-P(K4))]P(F2) \\
&\quad = 0.77508 \cdot 0.58852 \\
&\quad = 0.45615.
\end{aligned}$$

If we make the same kind of error as was done for the one-engine system in equation (9.3), we find that:

$$\begin{aligned}
&P(\text{system error in a year}) \\
&\quad = 1-(1-0.50532)(1-0.45615) = 0.73097.
\end{aligned} \tag{9.8}$$

In order to correct this mistake, we introduce some binary random variables:

$$X_1 = \begin{cases} 1 & \text{if } F2 \text{ does not occur within a year} \\ 0 & \text{if } F2 \text{ occurs within a year} \end{cases} \tag{9.9}$$

$$X_2 = \begin{cases} 1 & \text{if } K2 \text{ does not occur within a year} \\ 0 & \text{if } K2 \text{ occurs within a year} \end{cases}$$

$$X_3 = \begin{cases} 1 & \text{if } F3 \text{ does not occur within a year} \\ 0 & \text{if } F3 \text{ occurs within a year.} \end{cases}$$

$$X_4 = \begin{cases} 1 & \text{if } K3 \text{ does not occur within a year} \\ 0 & \text{if } K3 \text{ occurs within a year.} \end{cases}$$

$$X_5 = \begin{cases} 1 & \text{if } K4 \text{ does not occur within a year} \\ 0 & \text{if } K4 \text{ occurs within a year.} \end{cases}$$

Again, by letting component 1 with state variable X_1, correspond to the left propulsion engine, component 2 with state variable X_2 correspond to the left hydraulic clutch and so on, we see that the fault tree for the two-engine system is equivalent to the reliability block diagram shown in Figure 9.4.

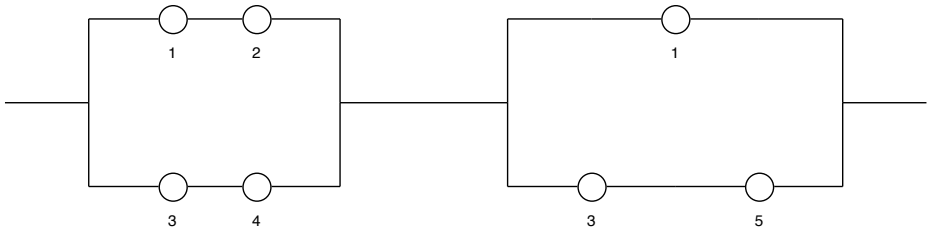

Figure 9.4: System equivalent to the fault tree for the two-engine system.

From this, we get that the structure function for the two-engine system is:

$$\begin{aligned}
\phi_2(\boldsymbol{X}) &= [1-(1-X_1X_2)(1-X_3X_4)][1-(1-X_1)(1-X_3X_5)] \\
&= (X_1X_2 + X_3X_4 - X_1X_2X_3X_4)(X_1 + X_3X_5 - X_1X_3X_5) \\
&= X_1X_2 + X_1X_3X_4 - X_1X_2X_3X_4 + X_1X_2X_3X_5 + X_3X_4X_5 \\
&\quad - X_1X_2X_3X_4X_5 - X_1X_2X_3X_5 - X_1X_3X_4X_5 + X_1X_2X_3X_4X_5 \\
&= X_1X_2 + X_1X_3X_4 - X_1X_2X_3X_4 + X_3X_4X_5 - X_1X_3X_4X_5 \\
&= X_1X_2 + X_3X_4[X_1(1-X_2) + X_5(1-X_1)].
\end{aligned}$$

Hence, all of the components are relevant since they are all part of the structure function. From this, we derive the correct estimate for the probability of the two-engine system failing:

$$\begin{aligned}
&P(\text{System error within a year}) \qquad (9.10)\\
&\quad = 1 - P(\phi_2(\boldsymbol{X}) = 1) \\
&\quad = 1 - E[\phi_2(\boldsymbol{X})] \\
&\quad = 1 - \big((1-P(F2)) \cdot (1-P(K2)) + (1-P(F3)) \cdot (1-P(K3)) \\
&\quad \cdot [(1-P(F2))P(K2) + (1-P(K4))P(F2)]\big) \\
&\quad = 0.50666.
\end{aligned}$$

To conclude, we see from equations (9.3) and (9.8) that when we are not careful with taking the system structure into account in the correct way, the two systems are (almost) equally reliable. However, if we do the reliability analysis in the correct way, considering the actual structure of the system, we find from equations (9.5) and (9.10) that the two-engine system is actually more reliable than the one-engine system (which was the system that was approved by the certification agency). Intuitively, this is actually quite obvious, since in the two-engine system, both engines can be used for both running the propeller and supplying power. In the one-engine system, the auxiliary engine is just an irrelevant component.

9.2 Case study: Reliability analysis of a network for transmission of electronic pulses

In this case study, we will perform a reliability analysis of a system for transmission of electronic pulses. This system consists of several identical parts. As we shall see, this property can be utilized in order to simplify the calculations.

In this case study twe consider a *multistate* system. This means that the system is not just functioning or failed. Instead the set of possible system states is a set of non-negative integers. This approach provides a more detailed description of the system. As before, we let:

$$X_i(t) = \text{ The state of component } i \text{ at time } t,$$

and assume that the stochastic processes $\{X_i(t), t \geq 0\}_{i=1}^n$ are independent. The structure function is given by:

$$\phi(t) = \phi(\boldsymbol{X}(t)) = \text{ The state of the system at time } t.$$

We assume that $\phi(t) \in \{\phi_1, \ldots, \phi_k\}$. In principle, one can find the distribution of the state of a multistate system by using the approach introduced in Section 4.1, i.e., by enumerating all possible component states:

$$P(\phi(t) = \phi_j) = \sum_{\boldsymbol{x}} I(\phi(\boldsymbol{x}) = \phi_j) \cdot P(\boldsymbol{X}(t) = \boldsymbol{x}),\ j = 1, \ldots, k. \tag{9.11}$$

where the indicator function $I(\phi(\boldsymbol{x}) = \phi_j)$ is 1 if $\phi(\boldsymbol{x}) = \phi_j$ and 0 otherwise. However, as in Section 4.1, we typically need to reduce the number of terms

in this sum because the calculations quickly become too time-consuming. We now assume that there exists variables $Y_1 = Y_1(\boldsymbol{X}), \ldots, Y_m = Y_m(\boldsymbol{X})$ such that $\phi(t)$ can be written:

$$\phi(t) = \phi(\boldsymbol{X}(t)) = \phi(\boldsymbol{Y}(\boldsymbol{X}(t))),$$

where $\boldsymbol{Y}(\boldsymbol{X}(t)) = (Y_1(\boldsymbol{X}(t)), \ldots, Y_m(\boldsymbol{X}(t)))$. In this case, the probability distribution of ϕ can be found from the formula:

$$P(\phi(t) = \phi_j) = \sum_{\boldsymbol{y}} I(\phi(\boldsymbol{y}) = \phi_j) P(\boldsymbol{Y}(\boldsymbol{X}(t)) = \boldsymbol{y}),\ j = 1, \ldots, k. \quad (9.12)$$

In order to use (9.12) to compute the system reliability, we have to compute $P(\boldsymbol{Y}(\boldsymbol{X}(t)) = \boldsymbol{y})$ for all $\boldsymbol{y}$. Hence, if the number of possible values of $\boldsymbol{Y}$ is large, little is gained by using (9.12) instead of (9.11). In some cases, however, we can achieve a large reduction in the computational load by a clever choice of $Y_1, \ldots, Y_m$. In particular, this turns out to be the case for the network in Figure 9.5.

The purpose of the system shown in Figure 9.5 is to ensure communication between a control room and 5 production units. In Figure 9.5 the components in the system are represented by blue circles. In addition to these components, the system consists of a control room, 5 production units and two connection units called A and B. To simplify the example somewhat, we will not consider potential errors in the control room, the production units and the connection units. Hence, the system consists of $n = 52$ components.

Each of the connection units, A and B, receive signals from the control room via 6 input wires. The number of production units which can be controlled by a connection unit is bounded by the number of functioning input wires to the respective connection unit. That is, if only 3 of the input wires to connection unit A are functioning, A can control at most 3 production units. If all 6 of the input wires to A are functioning, then all of the 5 production units can be controlled via connection unit A.

We assume that all of the components are stochastically independent, and we also assume that all of the 12 input wires to the connection units A and B have the same lifetime distributions. In particular, we assume that for any wire between the control room and a connection unit we have:

$$P(\text{The wire is functioning at time } t) = w(t).$$

We observe that the part of the system which ensures communication between the connection units A and B and the production units consists of 5 identical

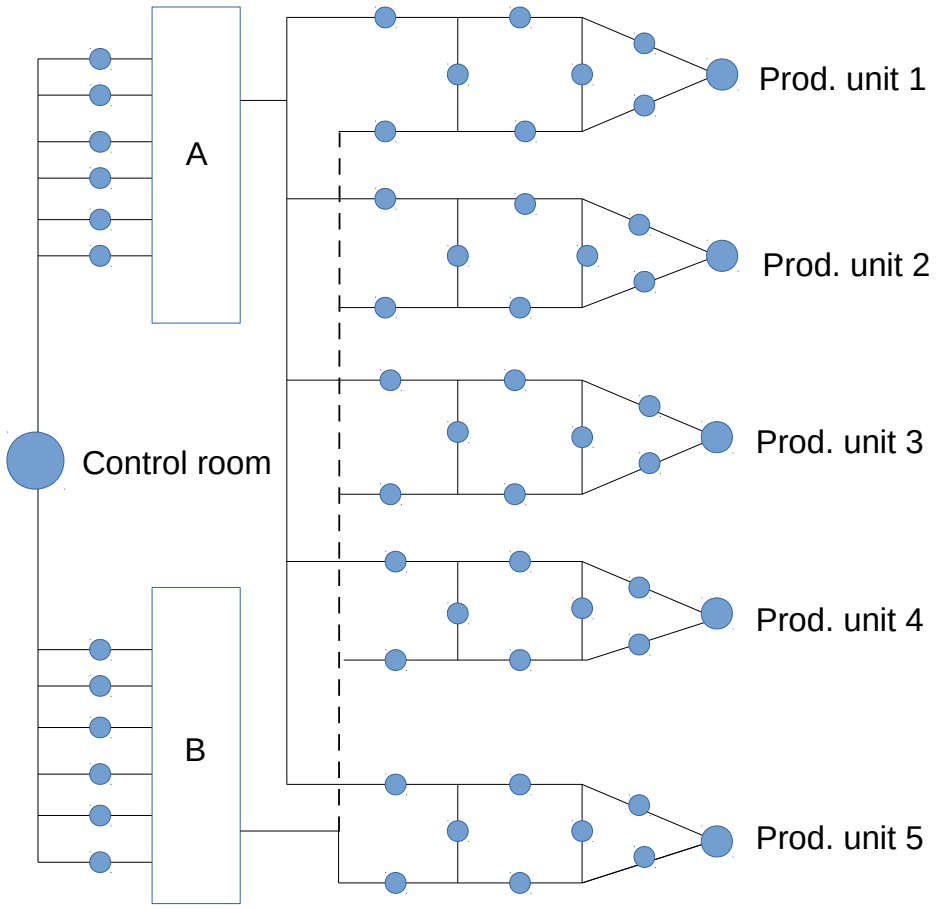

Figure 9.5: A network for transmission of electronic pulses.

subsystems. Each of these subsystems consists of 8 components. We assume that all of these subsystems have the same stochastic properties, in the sense that the corresponding components in the different subsystems have the same lifetime distributions.

Now, we define the state of the system, represented by the function ϕ, as:

$\phi(t) =$ The number of production units which can be controlled from the control room at time t.

Thus, the set of possible values for ϕ is $\{0, 1, \ldots, 5\}$. The components are assumed to be either functioning or failed, so the component states are binary. Hence, the system in Figure 9.5 is a multistate system of binary components. The number of terms in the sum (9.11) for computing the distribution of ϕ is $2^{52} = 4.504 \cdot 10^{15}$, so finding this by enumerating all of the states is very time-consuming. However, by introducing appropriate variables, we can obtain a significant reduction of terms in (9.12) compared to (9.11). More specifically, we introduce the following variables:

$Y_1(t) =$ The number of intact input wires to connection unit A at time t
$Y_2(t) =$ The number of intact input wires to connection unit B at time t
$Y_3(t) =$ The number of production units connected to A and B at time t
$Y_4(t) =$ The number of production units only connected to A at time t
$Y_5(t) =$ The number of production units only connected to B at time t.

The state of the system can now be expressed as:

$$\phi(t) = W_1(t) + W_2(t) + W_3(t), \tag{9.13}$$

where:

$$W_1(t) = \min\{Y_1(t), Y_4(t)\}, \quad W_2(t) = \min\{Y_2(t), Y_5(t)\}, \tag{9.14}$$
$$W_3(t) = \min\{(Y_1(t) - W_1(t)) + (Y_2(t) - W_2(t)), Y_3(t)\}.$$

In order to derive equation (9.13), we distribute the $Y_1(t)$ functioning input wires to A and the $Y_2(t)$ functioning input wires to B between the 5 production units in such a way that as many production units as possible are running. To do this, we first distribute input wires to the production units which are only connected to one connection unit, i.e., the $Y_4(t)$ production units only connected to A and the $Y_5(t)$ production units only connected to B. In this way, we establish connection with $W_1(t)$ production units via A and $W_2(t)$ connection units via B. When this is done, there are $Y_1(t) - W_1(t)$ input wires available via connection unit A, and $Y_2(t) - W_2(t)$ input wires available

via B. These are then used to establish connection to as many as possible of the $Y_3(t)$ remaining production units. The number of production units which can be reached in this way is then $W_3(t)$. Note that both $Y_1(t) - W_1(t)$ an $Y_2(t) - W_2(t)$ may be 0. We conclude that $W_1(t) + W_2(t) + W_3(t)$ is the number of production units which can be controlled from the control room. Thus, equation (9.13) is indeed correct.

Next, we need to find the probability distributions of $Y_1(t), \ldots, Y_5(t)$. Note that $Y_1(t)$ is a function of the state variables of the wires into connection unit A, $Y_2(t)$ is a function of the state variables of the wires into connection unit B, while the vector $(Y_3(t), Y_4(t), Y_5(t))$ is a function of state variables of the other 40 components. Since we have assumed that the components are independent, this implies that Y_1, Y_2 and the vector $(Y_3(t), Y_4(t), Y_5(t))$ are independent. Note, however that $Y_3(t)$, $Y_4(t)$ and $Y_5(t)$ are dependent. In fact we have $0 \leq Y_3(t) + Y_4(t) + Y_5(t) \leq 5$ for all $t \geq 0$.

Both $Y_1(t)$ and $Y_2(t)$ are sums of 6 independent, identically distributed binary random variables. Hence, from standard probability theory, $Y_1(t)$ and $Y_2(t)$ are binomially distributed:

$$P(Y_i(t) = y) = \binom{6}{y}[w(t)]^y[1 - w(t)]^{6-y}, \quad y = 0, 1, \ldots, 6, \quad i = 1, 2, \tag{9.15}$$

We then consider the probability distribution of $(Y_3(t), Y_4(t), Y_5(t))$. This depends on the 5 subsystems which ensure communication between the connection units and the production units. Each of these subsystems can be in one out of four possible states, denoted S_{AB}, S_A, S_B and $S_\emptyset$, where we define:

$$\begin{aligned} S_{AB} &= \text{The prod. unit can communicate with both connection units} \\ S_A &= \text{The prod. unit can only communicate with connection unit } A \\ S_B &= \text{The prod. unit can only communicate with connection unit } B \\ S_\emptyset &= \text{The prod. unit cannot communicate with any connection unit.} \end{aligned} \tag{9.16}$$

Furthermore, we have assumed that the subsystems are independent and that each of the subsystems have the same stochastic properties. This implies that the probability of being in either of the four states S_{AB}, S_A, S_B or $S_\emptyset$ at a given time t is the same for all the subsystems.

We observe that at time t the variables $Y_3(t), Y_4(t)$ and $Y_5(t)$ are the number of production units in states S_{AB}, S_A and S_B respectively. It follows

from standard probability theory that for $t \geq 0$ the vector $(Y_3(t), Y_4(t), Y_5(t))$ is multinomially distributed,. That is, we have:

$$P(Y_3(t) = y_3 \cap Y_4(t) = y_4 \cap Y_5(t) = y_5) = \frac{5!}{y_3!y_4!y_5!(5-y_3-y_4-y_5)!} \cdot [p_3(t)]^{y_3}[p_4(t)]^{y_4}[p_5(t)]^{y_5}[1-p_3(t)-p_4(t)-p_5(t)]^{5-y_3-y_4-y_5} \tag{9.17}$$

where $y_3 = 0, 1, \ldots, 5$, $y_4 = 0, 1, \ldots, 5 - y_3$, $y_5 = 0, 1, \ldots, 5 - y_3 - y_4$ and $p_3(t), p_4(t), p_5(t)$ are the probabilities of a subsystem being in states S_{AB}, S_A and S_B respectively.

In order to compute the distribution of ϕ, all that remains is to find $p_3(t), p_4(t)$ and $p_5(t)$. To do so, we study the structure of the subsystems, as shown in Figure 9.6, more closely.

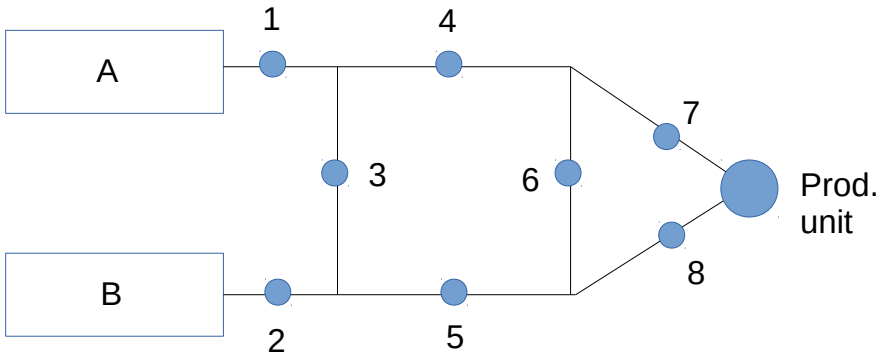

Figure 9.6: Subsystem of a network for transmission of electronic pulses.

In Figure 9.6, we have numbered the components of the subsystem under consideration from 1 through 8. We denote the corresponding component state variables by $X_1, \ldots, X_8$, and their respective reliabilities by $q_1, \ldots, q_8$, where we have simplified the notation by omitting the time t. Moreover, we

introduce the events:

$$E_A = \{\text{The production unit communicates with connection unit } A\}$$
$$E_B = \{\text{The production unit communicates with connection unit } B\},$$

and observe that

$$\begin{aligned} p_3(t) &= P(E_A \cap E_B), \\ p_4(t) &= P(E_A) - P(E_A \cap E_B), \\ p_5(t) &= P(E_B) - P(E_A \cap E_B). \end{aligned} \tag{9.18}$$

Computing $P(E_A \cap E_B)$

To find this probability, we condition with respect to the two bridges in the structure, i.e., components 3 and 6, and get:

$$\begin{aligned} P(E_A \cap E_B) = {} & P(E_A \cap E_B | X_3 = 1, X_6 = 1) q_3 q_6 \\ & + P(E_A \cap E_B | X_3 = 1, X_6 = 0) q_3 (1 - q_6) \\ & + P(E_A \cap E_B | X_3 = 0, X_6 = 1) (1 - q_3) q_6 \\ & + P(E_A \cap E_B | X_3 = 0, X_6 = 0) (1 - q_3)(1 - q_6). \end{aligned} \tag{9.19}$$

The four conditional probabilities in (9.19) can be computed by series- and parallel reductions (this is left as an exercise). They are given by:

$$\begin{aligned} P(E_A \cap E_B | X_3 = 1, X_6 = 1) &= q_1 q_2 [q_4 + q_5 - q_4 q_5][q_7 + q_8 - q_7 q_8], \\ P(E_A \cap E_B | X_3 = 1, X_6 = 0) &= q_1 q_2 [q_4 q_7 + q_5 q_8 - q_4 q_5 q_7 q_8], \\ P(E_A \cap E_B | X_3 = 0, X_6 = 1) &= q_1 q_2 q_4 q_5 [q_7 + q_8 - q_7 q_8], \\ P(E_A \cap E_B | X_3 = 0, X_6 = 0) &= q_1 q_2 q_4 q_5 q_7 q_8. \end{aligned} \tag{9.20}$$

Computing $P(E_A)$ and $P(E_B)$

Note that in order to compute $P(E_A)$, component 2 is irrelevant (see Figure 9.6). If we remove this component, we see that components 3 and 5 are in series. By series redulction, we end up with a bridge structure (see Example 3.1.2) connected in series with component 1. Hence, $P(E_A)$ is given by the following (again, see Example 3.1.2):

$$\begin{aligned} P(E_A) = {} & q_1 q_6 [q_4 + q_3 q_5 - q_3 q_4 q_5][q_7 + q_8 - q_7 q_8] \\ & + q_1 (1 - q_6)[q_4 q_7 + q_3 q_5 q_8 - q_3 q_4 q_5 q_7 q_8]. \end{aligned} \tag{9.21}$$

Similarly, $P(E_B)$ is given by:

$$\begin{aligned} P(E_B) = {} & q_2 q_6 [q_5 + q_3 q_4 - q_3 q_4 q_5][q_7 + q_8 - q_7 q_8] \\ & + q_2(1 - q_6)[q_5 q_8 + q_3 q_4 q_7 - q_3 q_4 q_5 q_7 q_8]. \end{aligned} \tag{9.22}$$

By inserting the expressions (9.19), (9.21) and (9.22) into (9.18), we have all the probabilities we need in order to calculate the distribution of the system state ϕ using equation (9.12).

What have we gained by this approach? Instead of enumerating all possible states of the 52 binary state variables of the components in Figure 9.5, we now have to enumerate the possible states of just the five multinary variables $Y_1, \ldots Y_5$. We observe that Y_1 and Y_2 each attain 7 values in the set $\{0, 1, \ldots 6\}$. The vector (Y_3, Y_4, Y_5) can attain any value in the set $\{(Y_3, Y_4, Y_5) : 0 \leq Y_3 + Y_4 + Y_5 \leq 5, 0 \leq Y_3, Y_4, Y_5\}$. In order to count the number of values in this set, we consider 6 different cases corresponding to the possible values of Y_3, and count the number of possible values for each case, noting that if $Y_3 = y$, then $Y_4 + Y_5 \leq 5 - y$, $y = 0, 1, \ldots, 5$. Hence, we get:

$$\begin{array}{lllll} \text{Case 0.} & Y_3 = 0, & Y_4 + Y_5 \leq 5 - 0 = 5: & \text{21 possible values,} \\ \text{Case 1.} & Y_3 = 1, & Y_4 + Y_5 \leq 5 - 1 = 4: & \text{15 possible values,} \\ \text{Case 2.} & Y_3 = 2, & Y_4 + Y_5 \leq 5 - 2 = 3: & \text{10 possible values,} \\ \text{Case 3.} & Y_3 = 3, & Y_4 + Y_5 \leq 5 - 3 = 2: & \text{6 possible values,} \\ \text{Case 4.} & Y_3 = 4, & Y_4 + Y_5 \leq 5 - 4 = 1: & \text{3 possible values,} \\ \text{Case 5.} & Y_3 = 5, & Y_4 + Y_5 \leq 5 - 5 = 0: & \text{1 possible value.} \end{array} \tag{9.23}$$

Adding these counts up, we see that there are 56 values in total. Hence, the sum in (9.12) for computing $P(\phi(t) = \phi_j)$ contains $7 \cdot 7 \cdot 56 = 2744$ terms (since there are 7 possible values for Y_1 and Y_2, as well as 56 possible values for (Y_3, Y_4, Y_5)). This is very easy to compute, as opposed to the $4.504 \cdot 10^{15}$ terms of the original method by simply enumerating the states. We see that by introducing the new variables $Y_1, \ldots, Y_5$, the computational load has been significantly reduced. The same approach is useful in other systems where the structure consists of many identical components and where there is a strong symmetry. A difficulty with the method is that in practice, it may not be possible to find efficient ways to introduce new variables. However, as there is so much to be gained from the method when it works, it may be worth spending some time trying to figure out which variables to introduce.

A python implementation of the above methods can be found in Script B.4.1 in Appendix B.4.

9.3 Exercises

Exercise 1. Use series- and parallel reductions to show that the conditional probabilities in (9.19) are given by the expressions in (9.20).

Exercise 2. Explain why the 6 different values in (9.23) hold true (for example, the only way $Y_4 + Y_5 \leq 0$ for non-negative integers is for $Y_4 = 0, Y_5 = 0$).

Appendix A

Notations

A^C The complement of a set. See Section 3.3.

$\equiv$ Identically equal.

$\phi(\cdot)$ The structure function of a binary system. See equation (2.2).

$h(\cdot)$ The reliability function of a binary system. See equation (2.8).

$\prod_{i=1}^{n} a_i = a_1 \cdot a_2 \ldots \cdot a_n$ The product operator.

$\coprod_{i=1}^{n} a_i = 1 - \prod_{i=1}^{n}(1 - a_i)$ The coproduct operator.

$\sum_{\mathbf{y}\in\{0,1\}^n} \ldots$ Sum over all possible n-dimensional binary vectors.

ϕ^D The dual structure function. See Definition 2.3.1.

0 A vector where all components are 0.

1 A vector where all components are 1.

Appendix B

Python scripts

This appendix contains various python scripts.

B.1 k-out-of-n systems and threshold systems

Script B.1.1

```
#########################################################################
#                                                                       #
#  This program calculates the reliability of a k-out-of-n system       #
#  by finding the distribution of:                                      #
#                                                                       #
#     S = X_{1} + ... + X_{n}.                                          #
#                                                                       #
#########################################################################

import numpy as np

# Component reliabilities (p[0] is not in use)
p = [0.0, 0.75, 0.80, 0.82, 0.65, 0.88, 0.91, 0.92, 0.86]

# Threshold
b = 6

# Number of components
n = len(p) - 1
```

```
# After each iteration ps[s] = P(X_{1} + ... + X_{i+1} = s)
ps = np.zeros(n + 1)

# Temporary storage of ps[s]
ph = np.zeros(n + 1)

# Calculate ps[0] = P(X_{1} = 0) and ps[1] = P(X_{1} = 1):
ps[0] = 1 - p[1]
ps[1] = p[1]

# Save these values for the next iteration
ph[0] = ps[0]
ph[1] = ps[1]


for j in range(1, n):
    # Calculate P(X_{1} + ... + X_{j+1} = 0):
    ps[0] = ph[0] * (1 - p[j+1])

    # Calculate P(X_{1} + ... + X_{j+1} = s), s = 1, ..., j:
    for s in range(1, j+1):
        ps[s] = ph[s] * (1 - p[j+1]) + ph[s-1] * p[j+1]

    # Calculate P(X_{1} + ... + X_{j+1} = j+1):
    ps[j+1] = ph[j] * p[j+1]

    # Save these values for the next iteration
    for s in range(j+2):
        ph[s] = ps[s]


# At this stage ps[s] = P(X_{1} + ... + X_{n} = s), s = 0, 1, ..., n

for s in range(n+1):
    print("P(S = ", s, ") = ", ps[s])

# Calculate the system reliability h = P(S = b) + ... + P(S = n)
h = 0
```

```
for s in range(b, n+1):
    h += ps[s]

print("h = ", h)
```

Script B.1.2

```
#########################################################################
#                                                                       #
#  This program calculates the reliability of a threshold system        #
#  by finding the distribution of:                                      #
#                                                                       #
#     S = a_{1} X_{1} + ... + a_{n} X_{n}.                              #
#                                                                       #
#########################################################################

import numpy as np

# Component reliabilities (p[0] is not in use)
p = [0.0, 0.75, 0.80, 0.82, 0.65, 0.88, 0.91, 0.92, 0.86]

# Component weights (a[0] is not in use)
a = [0, 2, 3, 5, 6, 7, 8, 8, 11]

# Threshold
b = 32

# Number of components
n = len(p) - 1

# Cumulative weights (d[0] is not in use)
d = np.zeros(n+1, dtype=int)

for j in range(1, n + 1):
    d[j] = a[j] + d[j-1]

# After each iteration ps[s] = a_{1} X_{1} + ... + a_{n} X_{n} = s)
ps = np.zeros(d[n] + 1)

# Temporary storage of ps[s]
ph = np.zeros(d[n] + 1)

# Calculate ps[0] = P(X_{1} = 0) and ps[1] = P(X_{1} = 1):
ps[0] = 1 - p[1]
ps[a[1]] = p[1]
```

```
# Save these values for the next iteration
ph[0] = ps[0]
ph[a[1]] = ps[a[1]]


for j in range(1, n):
    # Calculate P(a_{1} X_{1} + ... + a_{j} X_{j+1} = s), s < a[j+1]
    for s in range(a[j+1]):
        ps[s] = ph[s] * (1 - p[j+1])

    # Calculate P(a_{1} X_{1} + ... + a_{j} X_{j+1} = s), a[j+1] <= s <= d[j]
    for s in range(a[j+1], d[j] + 1):
        ps[s] = ph[s] * (1 - p[j+1]) + ph[s-a[j+1]] * p[j+1]

    # Calculate P(a_{1} X_{1} + ... + a_{j} X_{j+1} = s), s > d[j]
    for s in range(d[j] + 1, d[j+1] + 1):
        ps[s] = ph[s-a[j+1]] * p[j+1]

    # Save these values for the next iteration
    for s in range(d[j+1] + 1):
        ph[s] = ps[s]

# At this stage ps[s] = P(a_{1} X_{1} + ... + a_{j} X_{j+1} = s),
# s = 0, ... , d[n]

for s in range(d[n] + 1):
    print("P(S = ", s, ") = ", ps[s])

# Calculate the system reliability h = P(S = b) + ... + P(S = d[n])
h = 0

for s in range(d[n] + 1):
    if s >= b:
        h += ps[s]

print("h = ", h)
```

Script B.1.3

```
#########################################################################
#                                                                       #
#  This program calculates the reliability of a threshold system        #
#  by finding the distribution of:                                      #
#                                                                       #
#     S = (a_{1} X_{1} + ... + a_{n} X_{n}) / gcf.                      #
#                                                                       #
#  where gcf denotes the greatest common factor of the                  #
#  weights a_{1}, ..., a_{n}.                                           #
#                                                                       #
#########################################################################

import numpy as np

# Find the greatest common factor of a set of integers using
# the Euclidean algorithm.
# See:  https://en.wikipedia.org/wiki/Greatest_common_divisor
def greatest_common_factor(x):
    if len(x) == 1:
        for i in x:
            return i
    m = np.inf
    for i in x:
        if i < m and i > 0:
            m = i
    y = {m}
    for i in x:
        if i > m:
            y.add(i-m)
    return greatest_common_factor(y)

# Component reliabilities (p[0] is not in use)
p = [0.0, 0.75, 0.80, 0.82, 0.65, 0.88, 0.91, 0.92, 0.86]

# Component weights (a[0] is not in use)
a = [0, 26, 39, 65, 78, 91, 104, 104, 143]

# Threshold
b = 410
```

```
gcf = greatest_common_factor(set(a))
for i in range(len(a)):
    a[i] = int(a[i] / gcf)

# Number of components
n = len(p) - 1

# Cumulative weights (d[0] is not in use)
d = np.zeros(n+1, dtype=int)

for j in range(1, n + 1):
    d[j] = a[j] + d[j-1]

# After each iteration ps[s] = a_{1} X_{1} + ... + a_{n} X_{n} = s)
ps = np.zeros(d[n] + 1)

# Temporary storage of ps[s]
ph = np.zeros(d[n] + 1)

# Calculate ps[0] = P(X_{1} = 0) and ps[1] = P(X_{1} = 1):
ps[0] = 1 - p[1]
ps[a[1]] = p[1]

# Save these values for the next iteration
ph[0] = ps[0]
ph[a[1]] = ps[a[1]]

for j in range(1, n):
    # Calculate P(a_{1} X_{1} + ... + a_{j} X_{j+1} = s),
    # s = 0, 1, ... , (a[j+1]-1)
    for s in range(a[j+1]):
        ps[s] = ph[s] * (1 - p[j+1])

    # Calculate P(a_{1} X_{1} + ... + a_{j} X_{j+1} = s),
    # s = a[j+1], ..., d[j]:
    for s in range(a[j+1], d[j] + 1):
        ps[s] = ph[s] * (1 - p[j+1]) + ph[s-a[j+1]] * p[j+1]

    # Calculate P(a_{1} X_{1} + ... + a_{j} X_{j+1} = s),
```

```
    # s = d[j]+1 , ... , d[j+1]:
    for s in range(d[j] + 1, d[j+1] + 1):
        ps[s] = ph[s-a[j+1]] * p[j+1]

    # Save these values for the next iteration
    for s in range(d[j+1] + 1):
        ph[s] = ps[s]

# At this stage ps[s] = P(a_{1} X_{1} + ... + a_{j} X_{j+1} = s),
# s = 0, 1, ... , d[n]

for s in range(d[n] + 1):
    print("P(S = ", s, ") = ", ps[s])

# Calculate the system reliability:
# h = P(S = b) + ... + P(S = d[n])
h = 0

for s in range(d[n] + 1):
    if gcf * s >= b:
        h += ps[s]

print("h = ", h)
```

B.2 Binary decision diagrams

Script B.2.1

```
############################################################
#                                                          #
#    This file contains various classes and methods        #
#    needed to handle BDDs [Filename: c_bdd.py]            #
#                                                          #
############################################################

# Constants
LEVL_PREFIX = 'L'
COMP_PREFIX = 'C'
NODE_PREFIX = 'N'

FAILED = 0
FUNCTIONING = 1

PRINT_PROBABILITIES = True

class BDDLevel:
    def __init__(self, sys, i):
        self.system = sys
        self.index = i
        self.nodes = []

    def getName(self):
        return LEVL_PREFIX + str(self.index)

    def getNextLevel(self):
        return self.system.getLevel(self.index + 1)

    def getPrevLevel(self):
        return self.system.getLevel(self.index - 1)

    def getFailedNode(self):
        return self.system.getFailedNode()

    def getFunctioningNode(self):
        return self.system.getFunctioningNode()
```

```
    def addNode(self, nd):
        self.nodes.append(nd)

    def getNodeIndex(self, nd):
        return self.nodes.index(nd)

    def createLeaves(self):
        k = len(self.nodes)
        for j in range(k):
            self.nodes[j].createLeaves()

    def initializeProbability(self):
        k = len(self.nodes)
        for j in range(k):
            self.nodes[j].initializeProbability()

    def propagateProbability(self):
        k = len(self.nodes)
        for j in range(k):
            self.nodes[j].propagateProbability()

    def printLevel(self):
        print(self.getName(), "[", sep = "", end = "")
        k = len(self.nodes)
        for j in range(k):
            self.nodes[j].printNode()
            if j < k-1:
                print(", ", sep = "", end = "")
        print("]")


class BDDStructure:
    def __init__(self, n):
        self.order = n
        self.pivotIndex = 0
        self.components = []

    def addComponent(self, comp):
        self.components.append(comp)
```

```
    def getPivotComponent(self):
        self.selectPivotIndex()
        return self.components[self.pivotIndex]

    def selectPivotIndex(self):
        pass

    def restriction(self):
        pass

    def contraction(self):
        pass

    def isFunctioning(self):
        pass

    def isFailed(self):
        pass


class BDDSystem:
    def __init__(self, st):
        self.structure = st
        self.order = self.structure.order
        self.components = []
        for i in range(self.order):
            comp = BDDComponent(self, i)
            self.components.append(comp)
            self.structure.addComponent(comp)
        self.levels = []
        for i in range(self.order):
            levl = BDDLevel(self, i)
            self.levels.append(levl)
        self.root = BDDNode(self.levels[0], st)
        self.levels[0].addNode(self.root)
        self.failedNode = BDDNode(None, None, nm = "D0")
        self.functioningNode = BDDNode(None, None, nm = "D1")
        for i in range(self.order):
            self.levels[i].createLeaves()
```

```
    def getLevel(self, i):
        if i < 0:
            return None
        elif i < self.order:
            return self.levels[i]
        else:
            return None

    def getRoot(self):
        return self.root

    def getFailedNode(self):
        return self.failedNode

    def getFunctioningNode(self):
        return self.functioningNode

    # Calculate reliability when all components have equal reliability.
    # rel = The common reliability
    def calculateReliability0(self, rel):
        for i in range(self.order):
            self.components[i].setProbability(rel)
        self.root.initializeProbability(1)
        for i in range(1, self.order):
            self.levels[i].initializeProbability()
        self.failedNode.initializeProbability()
        self.functioningNode.initializeProbability()
        for i in range(self.order):
            self.levels[i].propagateProbability()
        return self.failedNode.getProbability(), \
            self.functioningNode.getProbability()

    # Calculate reliability for a given vector of component reliabilities.
    # rv = The vector of component reliabilities
    def calculateReliability(self, rv):
        for i in range(self.order):
            self.components[i].setProbability(rv[i])
        self.root.initializeProbability(1)
        for i in range(1, self.order):
```

```
            self.levels[i].initializeProbability()
        self.failedNode.initializeProbability()
        self.functioningNode.initializeProbability()
        for i in range(self.order):
            self.levels[i].propagateProbability()
        return self.failedNode.getProbability(), \
            self.functioningNode.getProbability()

    def printSystem(self):
        for i in range(self.order):
            self.levels[i].printLevel()


class BDDComponent:
    def __init__(self, sys, i):
        self.system = sys
        self.index = i
        self.probability = 0

    def getName(self):
        return COMP_PREFIX + str(self.index)

    def setProbability(self, p):
        self.probability = p

    def getProbability(self):
        return self.probability


class BDDNode:
    def __init__(self, lv, st, rt = None, nm = ""):
        self.level = lv
        self.structure = st
        self.root = rt
        if self.structure != None:
            self.component = self.structure.getPivotComponent()
        else:
            self.component = None
        self.left = None
        self.right = None
```

```
        self.probability = 0
        self.name = nm

    def getName(self):
        if self.level != None:
            return NODE_PREFIX + str(self.level.getNodeIndex(self))
        else:
            return self.name

    def isRoot(self):
        return (self.root == None)

    def isLeaf(self):
        return (self.level == None)

    def initializeProbability(self, p = 0):
        self.probability = p

    def addProbability(self, p):
        self.probability += p

    def getProbability(self):
        return self.probability

    def createLeaves(self):
        if not self.isLeaf():
            nextLevel = self.level.getNextLevel()
            leftStructure = self.structure.restriction()
            if leftStructure.isFailed():
                self.left = self.level.getFailedNode()
            elif nextLevel != None:
                self.left = BDDNode(nextLevel, leftStructure, self)
                nextLevel.addNode(self.left)
            else:
                self.left = self.level.getFailedNode()
            rightStructure = self.structure.contraction()
            if rightStructure.isFunctioning():
                self.right = self.level.getFunctioningNode()
            elif nextLevel != None:
                self.right = BDDNode(nextLevel, rightStructure, self)
```

```
            nextLevel.addNode(self.right)
        else:
            self.right = self.level.getFunctioningNode()

    def propagateProbability(self):
        if self.component != None:
            p = self.component.getProbability()
            self.left.addProbability(self.probability * (1-p))
            self.right.addProbability(self.probability * p)

    def printNode(self):
        if PRINT_PROBABILITIES:
            print(self.getName(), "(", \
                self.getProbability(), ")", sep = "", end = "")
        else:
            print(self.getName(), "()", sep = "", end = "")
```

Script B.2.2

```
############################################################
#                                                          #
#    This file contains various classes and methods        #
#    needed to handle ROBDDs [Filename: c_robdd.py]        #
#                                                          #
############################################################

# Constants
COMP_PREFIX = 'C'
NODE_PREFIX = 'N'

FAILED = 0
FUNCTIONING = 1

DO_SIMPLIFY = True
PRINT_PROBABILITIES = True

def SET_DO_SIMPLIFY(flag):
    global DO_SIMPLIFY
    DO_SIMPLIFY = flag

def SET_PRINT_PROBABILITIES(flag):
    global PRINT_PROBABILITIES
    PRINT_PROBABILITIES = flag

class ROBDDComponent:
    def __init__(self, sys, i):
        self.system = sys
        self.index = i
        self.nodes = []

    def getName(self):
        return COMP_PREFIX + str(self.index)

    def getNextComponent(self):
        return self.system.getComponent(self.index + 1)

    def getPrevComponent(self):
        return self.system.getComponent(self.index - 1)
```

```
    def getFailedNode(self):
        return self.system.getFailedNode()

    def getFunctioningNode(self):
        return self.system.getFunctioningNode()

    def addNode(self, nd):
        self.nodes.append(nd)

    def removeNode(self, nd):
        self.nodes.remove(nd)

    def getNodeIndex(self, nd):
        return self.nodes.index(nd)

    def createLeaves(self):
        k = len(self.nodes)
        for j in range(k):
            self.nodes[j].createLeaves()

    def simplify(self):
        if DO_SIMPLIFY:
            k1 = len(self.nodes)
            for j1 in range(k1-1):
                i1 = k1-2-j1
                n1 = self.nodes[i1]
                k2 = len(self.nodes)
                for j2 in range(k2-i1-1):
                    i2 = k2-1-j2
                    n2 = self.nodes[i2]
                    if n1.isEquivalentTo(n2):
                        n1.mergeNode(n2)

    def initializeProbability(self):
        k = len(self.nodes)
        for j in range(k):
            self.nodes[j].initializeProbability()

    def propagateProbability(self, rel):
```

```
        k = len(self.nodes)
        for j in range(k):
            self.nodes[j].propagateProbability(rel)

    def printComponent(self):
        print(self.getName(), "[", sep = "", end = "")
        k = len(self.nodes)
        for j in range(k):
            self.nodes[j].printNode()
            if j < k-1:
                print(", ", sep = "", end = "")
        print("]")


class ROBDDStructure:
    def __init__(self, n):
        self.order = n

    def getOrder(self):
        return self.order

    def restriction(self):
        pass

    def contraction(self):
        pass

    def isFunctioning(self):
        pass

    def isFailed(self):
        pass

    def isEquivalentTo(self, s):
        pass

    def mergeStructure(self, s):
        pass
```

```
class ROBDDSystem:
    def __init__(self, st):
        self.structure = st
        self.order = self.structure.getOrder()
        self.components = []
        for i in range(self.order):
            comp = ROBDDComponent(self, i)
            self.components.append(comp)
        self.root = ROBDDNode(self.components[0], self.structure)
        self.failedNode = ROBDDNode(None, None, nm = "D0")
        self.functioningNode = ROBDDNode(None, None, nm = "D1")
        for i in range(self.order):
            self.components[i].createLeaves()
            if (i+1) < self.order:
                self.components[i+1].simplify()

    def getComponent(self, i):
        if i < 0:
            return None
        elif i < self.order:
            return self.components[i]
        else:
            return None

    def getRoot(self):
        return self.root

    def getFailedNode(self):
        return self.failedNode

    def getFunctioningNode(self):
        return self.functioningNode

    # Calculate reliability when all components have equal reliability.
    # rel = The common reliability
    def calculateReliability0(self, rel):
        self.root.initializeProbability(1)
        for i in range(1, self.order):
            self.components[i].initializeProbability()
        self.failedNode.initializeProbability()
```

```
        self.functioningNode.initializeProbability()
        for i in range(self.order):
            self.components[i].propagateProbability(rel)
        return self.failedNode.getProbability(), \
            self.functioningNode.getProbability()

    # Calculate reliability for a given vector of component reliabilities.
    # rv = The vector of component reliabilities
    def calculateReliability(self, rv):
        self.root.initializeProbability(1)
        for i in range(1, self.order):
            self.components[i].initializeProbability()
        self.failedNode.initializeProbability()
        self.functioningNode.initializeProbability()
        for i in range(self.order):
            self.components[i].propagateProbability(rv[i])
        return self.failedNode.getProbability(), \
            self.functioningNode.getProbability()

    def printSystem(self):
        for i in range(self.order):
            self.components[i].printComponent()


class ROBDDNode:
    def __init__(self, cp, st, rt = None, nm = ""):
        self.component = cp
        if cp != None:
            self.component.addNode(self)
        self.structure = st
        if rt == None:
            self.roots = None
        else:
            self.roots = [rt]
        self.left = None
        self.right = None
        self.probability = 0
        self.name = nm

    def getName(self):
```

```
        if self.component != None:
            return NODE_PREFIX + str(self.component.getNodeIndex(self))
        else:
            return self.name

    def addRoot(self, rt):
        if self.roots == None:
            self.roots = [rt]
        else:
            self.roots.append(rt)

    def replaceSucc(self, old_nd, new_nd):
        if self.left == old_nd:
            self.left = new_nd
        if self.right == old_nd:
            self.right = new_nd

    def isEquivalentTo(self, nd):
        return self.structure.isEquivalentTo(nd.structure)

    def mergeNode(self, nd):
        m = len(nd.roots)
        for j in range(m):
            self.addRoot(nd.roots[j])
            nd.roots[j].replaceSucc(nd, self)
        self.structure.mergeStructure(nd.structure)
        self.component.removeNode(nd)

    def isRoot(self):
        return (self.roots == None)

    def isLeaf(self):
        return (self.component == None)

    def initializeProbability(self, p = 0):
        self.probability = p

    def addProbability(self, p):
        self.probability += p
```

```
    def getProbability(self):
        return self.probability

    def createLeaves(self):
        if not self.isLeaf():
            nextComponent = self.component.getNextComponent()
            leftStructure = self.structure.restriction()
            if leftStructure.isFailed():
                self.left = self.component.getFailedNode()
            elif nextComponent != None:
                self.left = ROBDDNode(nextComponent, leftStructure, self)
            else:
                self.left = self.component.getFailedNode()
            rightStructure = self.structure.contraction()
            if rightStructure.isFunctioning():
                self.right = self.component.getFunctioningNode()
            elif nextComponent != None:
                self.right = ROBDDNode(nextComponent, rightStructure, self)
            else:
                self.right = self.component.getFunctioningNode()

    def propagateProbability(self, p):
        self.left.addProbability(self.probability * (1-p))
        self.right.addProbability(self.probability * p)

    def printNode(self):
        if PRINT_PROBABILITIES:
            print(self.getName(), "(", \
                self.getProbability(), ")", sep = "", end = "")
        else:
            print(self.getName(), "()", sep = "", end = "")
```

Script B.2.3

```
############################################################
#                                                          #
#    This file contains various classes and methods        #
#    needed to handle threshold systems using either       #
#    BDD or ROBDD methods [Filename: c_threshold.py]       #
#                                                          #
############################################################

from c_bdd import BDDStructure
from c_robdd import ROBDDStructure


class BDDThreshold(BDDStructure):
    def __init__(self, w, wsum, th):
        super().__init__(len(w))
        self.weights = w
        self.weightsum = wsum
        self.threshold = th

    def selectPivotIndex(self):
        max_w = -1
        max_i = -1
        for i in range(self.order):
            if self.weights[i] > max_w:
                max_w = self.weights[i]
                max_i = i
        self.pivotIndex = max_i

    def restriction(self):
        w = []
        for i in range(self.order):
            if i != self.pivotIndex:
                w.append(self.weights[i])
        wsum = self.weightsum - self.weights[self.pivotIndex]
        struct = BDDThreshold(w, wsum, self.threshold)
        for i in range(self.order):
            if i != self.pivotIndex:
                struct.addComponent(self.components[i])
        return struct
```

```
    def contraction(self):
        w = []
        for i in range(self.order):
            if i != self.pivotIndex:
                w.append(self.weights[i])
        wsum = self.weightsum - self.weights[self.pivotIndex]
        struct = BDDThreshold(w, wsum, \
            self.threshold - self.weights[self.pivotIndex])
        for i in range(self.order):
            if i != self.pivotIndex:
                struct.addComponent(self.components[i])
        return struct

    def isFunctioning(self):
        return self.threshold <= 0

    def isFailed(self):
        return self.threshold > self.weightsum



class ROBDDThreshold(ROBDDStructure):
    def __init__(self, w, wsum, th):
        super().__init__(len(w))
        self.weights = w
        self.weightsum = wsum
        self.threshold = th

    def restriction(self):
        w = []
        for i in range(1, self.order):
            w.append(self.weights[i])
        wsum = self.weightsum - self.weights[0]
        return ROBDDThreshold(w, wsum, self.threshold)

    def contraction(self):
        w = []
        for i in range(1, self.order):
            w.append(self.weights[i])
```

```
        wsum = self.weightsum - self.weights[0]
        return ROBDDThreshold(w, wsum, \
            self.threshold - self.weights[0])

    def isFunctioning(self):
        return self.threshold <= 0

    def isFailed(self):
        return self.threshold > self.weightsum

    def isEquivalentTo(self, s):
        return self.threshold == s.threshold
```

Script B.2.4

```
############################################################
#                                                          #
#    Construct a BDD and a ROBDD for a given threshold     #
#    system and evaluate its reliability                   #
#                                                          #
############################################################

from c_bdd import BDDSystem
from c_robdd import ROBDDSystem
from c_threshold import BDDThreshold, ROBDDThreshold

# Component reliabilities
p = 0.5

# Component weights
a = [8, 7, 6, 5, 3, 2]

# Threshold
b = 20

# Number of components
n = len(a)


# Sum of weights
wsum = 0
for wg in a:
    wsum += wg

sys = BDDSystem(BDDThreshold(a, wsum, b))

result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("BDD-method:")
print("---------------")
print("System unreliability = ", result[0])
```

```
print("System reliability = ", result[1])

print("")
print("------------------------")
print("")

sys = ROBDDSystem(ROBDDThreshold(a, wsum, b))

result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("ROBDD-method:")
print("---------------")
print("System unreliability = ", result[0])
print("System reliability = ", result[1])
```

Script B.2.5

```
############################################################
#                                                          #
#    Construct a BDD and a ROBDD for a given threshold     #
#    system and evaluate its reliability                   #
#                                                          #
############################################################

from c_bdd import BDDSystem
from c_robdd import ROBDDSystem
from c_threshold import BDDThreshold, ROBDDThreshold

# Component reliabilities
p = 0.5

# Component weights
a = [11, 8, 8, 7, 6, 5, 3, 2]

# Threshold
b = 32

# Number of components
n = len(a)


# Sum of weights
wsum = 0
for wg in a:
    wsum += wg

sys = BDDSystem(BDDThreshold(a, wsum, b))

result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("BDD-method:")
print("---------------")
print("System unreliability = ", result[0])
```

```
print("System reliability = ", result[1])

print("")
print("------------------------")
print("")

sys = ROBDDSystem(ROBDDThreshold(a, wsum, b))

result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("ROBDD-method:")
print("---------------")
print("System unreliability = ", result[0])
print("System reliability = ", result[1])
```

Script B.2.6

```
############################################################
#                                                          #
#    This file contains various classes and methods        #
#    needed to handle consecutive threshold systems        #
#    using either BDD or ROBDD methods                     #
#    [Filename: c_conthreshold.py]                         #
#                                                          #
############################################################

from c_bdd import BDDStructure
from c_robdd import ROBDDStructure

class BDDConsecutiveThreshold(BDDStructure):
    def __init__(self, w, wsum, s0, th):
        super().__init__(len(w))
        self.weights = w
        self.weightsum = wsum
        self.initialvalue = s0
        self.threshold = th

    def selectPivotIndex(self):
        return 0

    def restriction(self):
        w = []
        for i in range(1, self.order):
            w.append(self.weights[i])
        struct = BDDConsecutiveThreshold(w, self.weightsum - self.weights[0], \
                    0, self.threshold)
        for i in range(1, self.order):
            struct.addComponent(self.components[i])
        return struct

    def contraction(self):
        w = []
        for i in range(1, self.order):
            w.append(self.weights[i])
        struct = BDDConsecutiveThreshold(w, self.weightsum - self.weights[0], \
                    self.initialvalue + self.weights[0], self.threshold)
```

```
        for i in range(1, self.order):
            struct.addComponent(self.components[i])
        return struct

    def isFunctioning(self):
        return self.threshold <= self.initialvalue

    def isFailed(self):
        return self.threshold > self.weightsum + self.initialvalue


class ROBDDConsecutiveThreshold(ROBDDStructure):
    def __init__(self, w, wsum, s0, th):
        super().__init__(len(w))
        self.weights = w
        self.weightsum = wsum
        self.initialvalue = s0
        self.threshold = th

    def restriction(self):
        w = []
        for i in range(1, self.order):
            w.append(self.weights[i])
        return ROBDDConsecutiveThreshold(w, self.weightsum - self.weights[0], \
                    0, self.threshold)

    def contraction(self):
        w = []
        for i in range(1, self.order):
            w.append(self.weights[i])
        return ROBDDConsecutiveThreshold(w, self.weightsum - self.weights[0], \
                    self.initialvalue + self.weights[0], self.threshold)

    def isFunctioning(self):
        return self.threshold <= self.initialvalue

    def isFailed(self):
        return self.threshold > self.weightsum + self.initialvalue

    def isEquivalentTo(self, s):
```

```
        return self.initialvalue == s.initialvalue
```

Script B.2.7

```
############################################################
#                                                          #
#    Construct a BDD and a ROBDD for a given consecutive   #
#    threshold system and evaluate its reliability         #
#                                                          #
############################################################

from c_bdd import BDDSystem
from c_robdd import ROBDDSystem
from c_conthreshold import \
    BDDConsecutiveThreshold, ROBDDConsecutiveThreshold

# Component reliabilities
p = 0.5

# Component weights
a = [8, 7, 6, 5, 3, 2]


# Threshold
b = 10

# Number of components
n = len(a)


# Sum of weights
wsum = 0
for wg in a:
    wsum += wg

sys = BDDSystem(BDDConsecutiveThreshold(a, wsum, 0, b))

result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("BDD-method:")
```

```
print("---------------")
print("System unreliability = ", result[0])
print("System reliability = ", result[1])

print("")
print("-------------------------")
print("")

sys = ROBDDSystem(ROBDDConsecutiveThreshold(a, wsum, 0, b))

result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("ROBDD-method:")
print("---------------")
print("System unreliability = ", result[0])
print("System reliability = ", result[1])
```

Script B.2.8

```
############################################################
#                                                          #
#    This file contains various classes and methods        #
#    needed to handle network systems                      #
#    [Filename: c_graph.py]                                #
#                                                          #
############################################################

import numpy as np

from c_bdd import BDDStructure
from c_robdd import ROBDDStructure


class GraphNode:
    def __init__(self):
        self.edges = []
        self.marked = False
        self.tempID = -1
        self.state = 1          # 1 = FUNCTIONING, 0 = FAILED

    def addEdge(self, e):
        if (self.edges.count(e) == 0):
            self.edges.append(e)

    def removeEdge(self, e):
        if (self.edges.count(e) > 0):
            self.edges.remove(e)

    def removeAllEgdes(self):
        self.edges.clear()

    def edgeCount(self):
        return len(self.edges)

    def mergeNode(self, n):
        c = n.edgeCount()
        for j in range(c):
            e = n.edges[j]
```

```
            self.addEdge(e)
            e.move(n, self)
        n.removeAllEgdes()

    def isMarked(self):
        return self.marked

    def mark(self):
        self.marked = True

    def unmark(self):
        self.marked = False

    def setTempID(self, i):
        self.tempID = i

    def getTempID(self):
        return self.tempID

    def setState(self, s):
        self.state = s

    def getState(self):
        return self.state

    def isFunctioning(self):
        return (self.state == 1)

    def isFailed(self):
        return (self.state == 0)

    def markByState(self):
        if self.isFailed() or self.isMarked():
            return
        self.mark()
        for edge in self.edges:
            edge.markByState()
```

```
class GraphEdge:
    def __init__(self, n1, n2):
        self.node1 = n1
        self.node2 = n2    # We allow loops, i.e., n1 may be equal to n2
        self.marked = False
        self.tempID = -1
        self.state = 1     # 1 = FUNCTIONING, 0 = FAILED

    def move(self, fromNode, toNode):
        if fromNode == self.node1:
            self.node1 = toNode
        if fromNode == self.node2:
            self.node2 = toNode

    def isMarked(self):
        return self.marked

    def mark(self):
        self.marked = True

    def unmark(self):
        self.marked = False

    def setTempID(self, i):
        self.tempID = i

    def getTempID(self):
        return self.tempID

    def setState(self, s):
        self.state = s

    def getState(self):
        return self.state

    def isFunctioning(self):
        return (self.state == 1)

    def isFailed(self):
        return (self.state == 0)
```

```
    def markByState(self):
        if self.isFailed() or self.isMarked():
            return
        self.mark()
        self.node1.markByState()
        self.node2.markByState()


class Graph:
    def __init__(self):
        self.nodes = []
        self.edges = []
        self.terminals = []

    def addNode(self):
        node = GraphNode()
        self.nodes.append(node)

    def addEdge(self, i1, i2):
        node1 = self.nodes[i1]
        node2 = self.nodes[i2]
        edge = GraphEdge(node1, node2)
        node1.addEdge(edge)
        node2.addEdge(edge)
        self.edges.append(edge)

    def makeGraph(self, matrix):
        m,n = matrix.shape
        for _ in range(m):
            self.addNode()
        for j in range(n):
            i1 = -1
            i2 = -1
            for ii in range(m):
                if matrix[ii,j] == 1:
                    if i1 == -1:
                        i1 = ii
                    else:
                        i2 = ii
```

```
            if i2 == -1:
                i2 = i1
            self.addEdge(i1, i2)

    def makeTerminals(self, tlist):
        for i in tlist:
            terminal = self.nodes[i]
            self.terminals.append(terminal)

    def isTerminal(self, node):
        return (self.terminals.count(node) > 0)

    def getIncidenceMatrix(self):
        m = len(self.nodes)
        for i in range(m):
            self.nodes[i].setTempID(i)
        n = len(self.edges)
        matrix = np.zeros((m,n), dtype=int)
        for j in range(n):
            edge = self.edges[j]
            r1 = edge.node1.getTempID()
            r2 = edge.node2.getTempID()
            matrix[r1, j] = 1
            matrix[r2, j] = 1
        return matrix

    def getTerminalList(self):
        m = len(self.nodes)
        for i in range(m):
            self.nodes[i].setTempID(i)
        t = len(self.terminals)
        tlist = []
        for i in range(t):
            terminal = self.terminals[i]
            tlist.append(terminal.getTempID())
        return tlist

    def restriction(self, i = 0):
        edge = self.edges[i]
        self.edges.remove(edge)
```

```
        node1 = edge.node1
        node2 = edge.node2
        node1.removeEdge(edge)
        node2.removeEdge(edge)

    def contraction(self, i = 0):
        edge = self.edges[i]
        self.edges.remove(edge)
        node1 = edge.node1
        node2 = edge.node2
        if (node1 == node2):
            node1.removeEdge(edge)
        else:
            node1.removeEdge(edge)
            node2.removeEdge(edge)
            node1.mergeNode(node2)
            self.nodes.remove(node2)
            if self.isTerminal(node2):
                self.terminals.remove(node2)
                if not self.isTerminal(node1):
                    self.terminals.append(node1)

    def unmarkAll(self):
        for node in self.nodes:
            node.unmark()
        for edge in self.edges:
            edge.unmark()

    def setNodeState(self, i, s):
        self.nodes[i].setState(s)

    def getNodeState(self, i):
        return self.nodes[i].getState()

    def setEdgeState(self, i, s):
        self.edges[i].setState(s)

    def getEdgeState(self, i):
        return self.edges[i].getState()
```

```
    def computeState(self):
        self.unmarkAll()
        self.terminals[0].markByState()
        state = 1
        for terminal in self.terminals:
            if not terminal.isMarked():
                state = 0     # If at least one terminal is not marked,
                break         # the system is failed.
        return state          # If all terminals are marked,
                              # the system is functioning


class BDDGraph(BDDStructure):
    def __init__(self, matrix, tlist):
        super().__init__(matrix.shape[1])
        self.incidenceMatrix = matrix
        self.terminalList = tlist
        self.graph = Graph()
        self.graph.makeGraph(self.incidenceMatrix)
        self.graph.makeTerminals(tlist)

    def selectPivotIndex(self):
        return 0

    def restriction(self):
        g = Graph()
        g.makeGraph(self.incidenceMatrix)
        g.makeTerminals(self.terminalList)
        g.restriction(self.pivotIndex)
        matrix = g.getIncidenceMatrix()
        tlist = g.getTerminalList()
        struct = BDDGraph(matrix, tlist)
        for i in range(self.order):
            if i != self.pivotIndex:
                struct.addComponent(self.components[i])
        return struct

    def contraction(self):
        g = Graph()
        g.makeGraph(self.incidenceMatrix)
```

```
        g.makeTerminals(self.terminalList)
        g.contraction(self.pivotIndex)
        matrix = g.getIncidenceMatrix()
        tlist = g.getTerminalList()
        struct = BDDGraph(matrix, tlist)
        for i in range(self.order):
            if i != self.pivotIndex:
                struct.addComponent(self.components[i])
        return struct

    def isFunctioning(self):
        t = len(self.terminalList)
        return (t <= 1)

    def isFailed(self):
        g = Graph()
        g.makeGraph(self.incidenceMatrix)
        g.makeTerminals(self.terminalList)
        return (g.computeState() == 0)


class ROBDDGraph(ROBDDStructure):
    def __init__(self, matrix, tlist):
        super().__init__(matrix.shape[1])
        self.incidenceMatrix = matrix
        self.terminalList = tlist
        self.graph = Graph()
        self.graph.makeGraph(self.incidenceMatrix)
        self.graph.makeTerminals(tlist)

    def restriction(self):
        g = Graph()
        g.makeGraph(self.incidenceMatrix)
        g.makeTerminals(self.terminalList)
        g.restriction()
        matrix = g.getIncidenceMatrix()
        tlist = g.getTerminalList()
        return ROBDDGraph(matrix, tlist)

    def contraction(self):
```

```
        g = Graph()
        g.makeGraph(self.incidenceMatrix)
        g.makeTerminals(self.terminalList)
        g.contraction()
        matrix = g.getIncidenceMatrix()
        tlist = g.getTerminalList()
        return ROBDDGraph(matrix, tlist)

    def isFunctioning(self):
        t = len(self.terminalList)
        return (t <= 1)

    def isFailed(self):
        g = Graph()
        g.makeGraph(self.incidenceMatrix)
        g.makeTerminals(self.terminalList)
        return (g.computeState() == 0)

    def isEquivalentTo(self, s):
        m1,n1 = self.incidenceMatrix.shape
        m2,n2 = s.incidenceMatrix.shape
        t1 = len(self.terminalList)
        t2 = len(s.terminalList)
        # First, we check if the dimensions are equal
        if (m1 != m2) or (n1 != n2) or (t1 != t2):
            return False
        # Next, we check if the incidenceMatrices are equal
        matrixFlag = True
        for i in range(m1):
            for j in range(n1):
                if self.incidenceMatrix[i,j] != s.incidenceMatrix[i,j]:
                    matrixFlag = False
                    break
            if not matrixFlag:
                break
        if not matrixFlag:
            return False
        # Finally, we check if the terminalLists are equal
        terminalFlag = True
        for i in range(t1):
```

```
            if self.terminalList[i] != s.terminalList[i]:
                terminalFlag = False
                break
        return terminalFlag
```

Script B.2.9

```
############################################################
#                                                          #
#    Construct a BDD and a ROBDD for a given network       #
#    system, and evaluate its reliability                  #
#                                                          #
############################################################

from c_bdd import BDDSystem
from c_robdd import ROBDDSystem
from c_graph import BDDGraph, ROBDDGraph

import numpy as np

p = 0.5

matrix = np.array([[1, 1, 0, 0, 0], \
                   [1, 0, 1, 1, 0], \
                   [0, 1, 1, 0, 1], \
                   [0, 0, 0, 1, 1]])
tlist = [0,3]

sys = BDDSystem(BDDGraph(matrix, tlist))

result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("BDD-method:")
print("---------------")
print("System unreliability = ", result[0])
print("System reliability = ", result[1])

print("")
print("------------------------")
print("")

sys = ROBDDSystem(ROBDDGraph(matrix, tlist))
```

```
result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("ROBDD-method:")
print("---------------")
print("System unreliability = ", result[0])
print("System reliability = ", result[1])
```

Script B.2.10

```
############################################################
#                                                          #
#    Construct a BDD and a ROBDD for a given network       #
#    system, and evaluate its reliability                  #
#                                                          #
############################################################

from c_bdd import BDDSystem
from c_robdd import ROBDDSystem
from c_graph import BDDGraph, ROBDDGraph

import numpy as np

# Component reliabilities
p = 0.5

matrix = np.array([[1, 1, 0, 0, 0, 0, 0, 0], \
                   [1, 0, 1, 1, 0, 0, 0, 0], \
                   [0, 1, 1, 0, 1, 0, 0, 0], \
                   [0, 0, 0, 1, 0, 1, 1, 0], \
                   [0, 0, 0, 0, 1, 1, 0, 1], \
                   [0, 0, 0, 0, 0, 0, 1, 1]])
tlist = [0,5]


sys = BDDSystem(BDDGraph(matrix, tlist))

result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("BDD-method:")
print("---------------")
print("System unreliability = ", result[0])
print("System reliability = ", result[1])

print("")
print("------------------------")
```

```
print("")

sys = ROBDDSystem(ROBDDGraph(matrix, tlist))

result = sys.calculateReliability0(p)
sys.printSystem()

print("")
print("---------------")
print("ROBDD-method:")
print("---------------")
print("System unreliability = ", result[0])
print("System reliability = ", result[1])
```

B.3 Dynamic System reliability

Script B.3.1

```
#########################################################################
#                                                                       #
#  This script plots h(p(t)) as a function of t for a simple            #
#  2-out-of-3 system with Weibull-distributed component lifetimes.      #
#                                                                       #
#########################################################################

from math import *
import matplotlib.pyplot as plt
import numpy as np

# Time parameters
max_t = 120
steps = 120

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0]
beta = [50.0, 60.0, 70.0]

# Component reliability vector
num_comps = 3
p = np.zeros(num_comps)

# System reliabilities
h = np.zeros(steps)

def reliability(pp):
    return pp[0] * pp[1] + pp[0] * pp[2] + pp[1] * pp[2] \
        - 2 * pp[0] * pp[1] * pp[2]

# The survival function of a Weibull-distribution
def weibull(aa, bb, t):
    return exp(- (t / bb) ** aa)

time = np.linspace(0.0, max_t, steps)

for step in range(steps):
```

```
    for i in range(num_comps):
        p[i] = weibull(alpha[i], beta[i], time[step])
    h[step] = reliability(p)

fig = plt.figure(figsize = (7, 4))
plt.plot(time, h, label='h(p(t))')
plt.xlabel('Time')
plt.ylabel('Reliability')
plt.title("System reliability as a function of time")
plt.legend()
plt.show()
```

Script B.3.2

```
#######################################################################
#                                                                     #
#  This program plots h(p(t)) as a function of t for a system         #
#  with given minimal path sets using Monte Carlo simulations of      #
#  the component lifetimes and calculating the resulting system       #
#  lifetime.                                                          #
#                                                                     #
#######################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Number of components and paths
num_comps = 5
num_paths = 4

# Minimal path sets
paths = [{1,4}, {1,3,5}, {2,3,4}, {2,5}]

# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    sys_life = 0.0
    for j in range(num_paths):
        path_life = np.inf
        for i in paths[j]:
            if tt[i-1] < path_life:
                path_life = tt[i-1]
        if path_life > sys_life:
            sys_life = path_life
    return sys_life
```

```
# The upper percentile levels
q = np.zeros(num_sims)

# The simulated system lifetimes
s = np.zeros(num_sims)

# The simulated lifetimes of the components
t = np.zeros(num_comps)

for k in range(num_sims):
    q[n] = 1.0 - n / num_sims
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
    s[k] = sys_lifetime(t)

s.sort()

fig = plt.figure(figsize = (7, 4))
plt.plot(s, q, label='h(p(t))')
plt.xlabel('Time')
plt.ylabel('Reliability')
plt.title("System reliability as a function of time")
plt.legend()
plt.show()
```

Script B.3.3

```
#######################################################################
#                                                                     #
#  This program plots h(p(t)) as a function of t for a threshold      #
#  system using Monte Carlo simulations of the component              #
#  lifetimes and calculating the resulting system lifetime.           #
#                                                                     #
#######################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Threshold system weights and threshold
w = [2.0, 4.0, 2.3, 3.5, 1.9]
threshold = 5.0

# Number of components
num_comps = 5

# Component state vector
x = np.zeros(num_comps)

# The structure function of the threshold system
def phi(xx):
    wsum = 0.0
    for i in range(num_comps):
        wsum += w[i] * xx[i]
    if wsum >= threshold:
        return 1
    else:
        return 0
```

```
# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    # Initialize component states
    for i in range(num_comps):
        x[i] = 1
    # Initialize system state and lifetime
    sys_state = phi(x)
    sys_life = 0.0
    # Determine the ordering of the failure times
    order = tt.argsort()
    # Initialize failure counter
    c = 0
    # Switch off one component at a time in the order of the failure times
    while (sys_state == 1) and (c < num_comps):
        x[order[c]] = 0
        sys_life = tt[order[c]]
        sys_state = phi(x)
        c += 1
    if sys_state == 0:
        return sys_life
    else: # This happens only for trivial systems where phi(0,...,0) = 1.
        return np.inf

# The upper percentile levels
q = np.zeros(num_sims)

# The simulated system lifetimes
s = np.zeros(num_sims)

# The simulated lifetimes of the components
t = np.zeros(num_comps)

for k in range(num_sims):
    q[k] = 1.0 - k / num_sims
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
    s[k] = sys_lifetime(t)

s.sort()
```

```
fig = plt.figure(figsize = (7, 4))
plt.plot(s, q, label='h(p(t))')
plt.xlabel('Time')
plt.ylabel('Reliability')
plt.title("System reliability as a function of time")
plt.legend()
plt.show()
```

Script B.3.4

```
#########################################################################
#                                                                       #
#  This program plots h(p(t)) as a function of t for a threshold        #
#  system using a ROBDD representation of the system.                   #
#                                                                       #
#  Contrary to the previous script which uses MC simulation,            #
#  this script calculates reliability analytically. Thus, the           #
#  result is an exact h(p(t))-curve.                                    #
#                                                                       #
#########################################################################

from math import exp
import matplotlib.pyplot as plt
import numpy as np
from c_robdd import ROBDDSystem
from c_threshold import ROBDDThreshold

# Time parameters
max_t = 175
steps = 175

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Number of components in the system
num_comps = len(alpha)


# Threshold system weights and threshold
a = [2.0, 4.0, 2.3, 3.5, 1.9]
b = 5.0

# Calculate the weight sum
asum = 0
for i in range(num_comps):
    asum += a[i]

# The ROBDD representation of the threshold system
```

```
sys = ROBDDSystem(ROBDDThreshold(a, asum, b))

# The reliability function of the system
def reliability(pp):
    result = sys.calculateReliability(pp)
    return result[1]

# The survival function of a Weibull-distribution
def weibull(aa, bb, t):
    if t >= 0:
        return exp(- (t / bb) ** aa)
    else:
        return 1.0

# Component reliability vector
p = np.zeros(num_comps)

# Time values and Reliability function
time = np.linspace(0.0, max_t, steps)
h = np.zeros(steps)

# Calculate the Reliability function
for step in range(steps):
    for i in range(num_comps):
        p[i] = weibull(alpha[i], beta[i], time[step])
    h[step] = reliability(p)

# Plot the results
fig = plt.figure(figsize = (7, 4))
plt.plot(time, h, label='h(p(t))')
plt.xlabel('Time')
plt.ylabel('Reliability')
plt.title("System reliability as a function of time")
plt.legend()
plt.show()
```

Script B.3.5

```
######################################################################
#                                                                    #
#  This program plots h(p(t)) as a function of t for a network       #
#  system using Monte Carlo simulations of the component             #
#  lifetimes and calculating the resulting system lifetime.          #
#                                                                    #
######################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5, 2.0, 2.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0, 55.0, 65.0]

# Number of components
num_comps = 7

# Component state vector
x = np.zeros(num_comps)

# The coproduct function
def coprod(x1, x2):
    return x1 + x2 - x1 * x2

# The structure function given that component 3 is functioning
def phi_3_1(xx):
    return coprod(xx[2],xx[4])*coprod(xx[0],xx[1])*coprod(xx[5],xx[6]) \
        + (1 - coprod(xx[2], xx[4]))*coprod(xx[0]*xx[5], xx[1]*xx[6])

# The structure function given that component 3 is failed
def phi_3_0(xx):
    return coprod(xx[0] * xx[2], xx[1]) * coprod(xx[4] * xx[5], xx[6])
```

```
# The structure function of the network system (using pivotal decomposition)
def phi(xx):
    return xx[3] * phi_3_1(xx) + (1 - xx[3]) * phi_3_0(xx)

# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    # Initialize component states
    for i in range(num_comps):
        x[i] = 1
    # Initialize system state and lifetime
    sys_state = phi(x)
    sys_life = 0.0
    # Determine the ordering of the failure times
    order = tt.argsort()
    # Initialize failure counter
    c = 0
    # Switch off one component at a time in the order of the failure times
    while (sys_state == 1) and (c < num_comps):
        x[order[c]] = 0
        sys_life = tt[order[c]]
        sys_state = phi(x)
        c += 1
    if sys_state == 0:
        return sys_life
    else: # This happens only for trivial systems where phi(0,...,0) = 1.
        return np.inf

# The upper percentile levels
q = np.zeros(num_sims)

# The simulated system lifetimes
s = np.zeros(num_sims)

# The simulated lifetimes of the components
t = np.zeros(num_comps)

for k in range(num_sims):
    q[k] = 1.0 - k / num_sims
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
```

```
    s[k] = sys_lifetime(t)

s.sort()

fig = plt.figure(figsize = (7, 4))
plt.plot(s, q, label='h(p(t))')
plt.xlabel('Time')
plt.ylabel('Reliability')
plt.title("System reliability as a function of time")
plt.legend()
plt.show()
```

Script B.3.6

```
##############################################################################
#                                                                            #
#    This program plots I_B^{(i)}(t) as a function of t for a simple         #
#    2-out-of-3 system with Weibull-distributed component lifetimes.         #
#                                                                            #
##############################################################################

from math import exp
import matplotlib.pyplot as plt
import numpy as np

# Time parameters
max_t = 120
steps = 120

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0]
beta = [50.0, 60.0, 70.0]

# Number of components in the system
num_comps = len(alpha)

# The reliability function of the system
def reliability(pp):
    return pp[0] * pp[1] + pp[0] * pp[2] + pp[1] * pp[2] - 2 * pp[0] * pp[1] * pp[2]

# The survival function of a Weibull-distribution
def weibull(aa, bb, t):
    if t >= 0:
        return exp(- (t / bb) ** aa)
    else:
        return 1.0

# Component reliability vector
p = np.zeros(num_comps)

# Time values and Birnbaum measures
time = np.linspace(0.0, max_t, steps)
IB = np.zeros((num_comps, steps))
```

```
# Calculate the Birnbaum measures
for step in range(steps):
    for i in range(num_comps):
        p[i] = weibull(alpha[i], beta[i], time[step])
    for i in range(num_comps):
        temp = p[i]
        p[i] = 1
        h1 = reliability(p)
        p[i] = 0
        h0 = reliability(p)
        p[i] = temp
        IB[i][step] = h1 - h0

# Plot the results
fig = plt.figure(figsize = (7, 4))
for i in range(num_comps):
    plt.plot(time, IB[i], label='IB(' + str(i+1) + ", t)")
plt.xlabel('Time')
plt.ylabel('Importance')
plt.title("Importance as a function of time")
plt.legend()
plt.show()
```

Script B.3.7

```
########################################################################
#                                                                      #
#  This program plots I_B^{(i)}(t) as a function of t for a            #
#  system  with given minimal path sets using Monte Carlo              #
#  simulations of the component lifetimes and calculating the          #
#  resulting system lifetime.                                          #
#                                                                      #
########################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Time parameters
max_t = 120
steps = 120

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Number of components and minimal paths
num_comps = 5
num_paths = 4

# Birnbaum measures
IB = np.zeros((num_comps, steps))

# A component is identified as critical when t is in [L, U)
L = np.zeros(num_comps)
U = np.zeros(num_comps)

# Minimal path sets
paths = [{1,4}, {1,3,5}, {2,3,4}, {2,5}]
```

```
# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    sys_life = 0.0
    for j in range(num_paths):
        path_life = np.inf
        for i in paths[j]:
            if tt[i-1] < path_life:
                path_life = tt[i-1]
        if path_life > sys_life:
            sys_life = path_life
    return sys_life

# The simulated lifetimes of the components
t = np.zeros(num_comps)

# Time steps
time = np.linspace(0.0, max_t, steps)

# Run simulations
for k in range(num_sims):
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
    for i in range(num_comps):
        temp = t[i]
        t[i] = np.inf
        U[i] = sys_lifetime(t)
        t[i] = 0
        L[i] = sys_lifetime(t)
        t[i] = temp
    for step in range(steps):
        for i in range(num_comps):
            if L[i] <= time[step] and time[step] < U[i]:
                IB[i][step] += 1

# Calculate average importance values
for step in range(steps):
    for i in range(num_comps):
        IB[i][step] = IB[i][step] / num_sims

fig = plt.figure(figsize = (7, 4))
```

```
for i in range(num_comps):
    plt.plot(time, IB[i], label='IB(' + str(i+1) + ", t)")
plt.xlabel('Time')
plt.ylabel('Importance')
plt.title("Importance as a function of time")
plt.legend()
plt.show()
```

Script B.3.8

```
#########################################################################
#                                                                       #
#  This program plots I_B^{(i)}(t) as a function of t for a             #
#  threshold system using Monte Carlo simulations of the                #
#  component lifetimes and calculating the resulting system             #
#  lifetime.                                                            #
#                                                                       #
#########################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Time parameters
max_t = 120
steps = 120

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Threshold system weights and threshold
w = [2.0, 4.0, 2.3, 3.5, 1.9]
threshold = 5.0

# Number of components and minimal paths
num_comps = 5

# Birnbaum measures
IB = np.zeros((num_comps, steps))

# A component is identified as critical when t is in [L, U)
L = np.zeros(num_comps)
U = np.zeros(num_comps)
```

```
# Component state vector
x = np.zeros(num_comps)

# The structure function of the threshold system
def phi(xx):
    wsum = 0.0
    for i in range(num_comps):
        wsum += w[i] * xx[i]
    if wsum >= threshold:
        return 1
    else:
        return 0

# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    # Initialize component states
    for i in range(num_comps):
        x[i] = 1
    # Initialize system state and lifetime
    sys_state = phi(x)
    sys_life = 0.0
    # Determine the ordering of the failure times
    order = tt.argsort()
    # Initialize failure counter
    c = 0
    # Switch off one component at a time in the order of the failure times
    while (sys_state == 1) and (c < num_comps):
        x[order[c]] = 0
        sys_life = tt[order[c]]
        sys_state = phi(x)
        c += 1
    if sys_state == 0:
        return sys_life
    else: # This happens only for trivial systems where phi(0,...,0) = 1.
        return np.inf

# The simulated lifetimes of the components
t = np.zeros(num_comps)

# Time steps
```

```
time = np.linspace(0.0, max_t, steps)

# Run simulations
for k in range(num_sims):
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
    for i in range(num_comps):
        temp = t[i]
        t[i] = np.inf
        U[i] = sys_lifetime(t)
        t[i] = 0
        L[i] = sys_lifetime(t)
        t[i] = temp
    for step in range(steps):
        for i in range(num_comps):
            if L[i] <= time[step] and time[step] < U[i]:
                IB[i][step] += 1

# Calculate average importance values
for step in range(steps):
    for i in range(num_comps):
        IB[i][step] = IB[i][step] / num_sims

fig = plt.figure(figsize = (7, 4))
for i in range(num_comps):
    plt.plot(time, IB[i], label='IB(' + str(i+1) + ", t)")
plt.xlabel('Time')
plt.ylabel('Importance')
plt.title("Importance as a function of time")
plt.legend()
plt.show()
```

Script B.3.9

```
########################################################################
#                                                                      #
#  This program plots I_B^{(i)}(t), i = 1, ..., 5 as a function        #
#  of t for a threshold system using a ROBDD representation of         #
#  the system.                                                         #
#                                                                      #
#  Contrary to the previous script which uses MC simulation,           #
#  this script calculates importance analytically. Thus, the           #
#  results are exact I_B^{(i)}(t)-curves.                              #
#                                                                      #
########################################################################

from math import exp
import matplotlib.pyplot as plt
import numpy as np
from c_robdd import ROBDDSystem
from c_threshold import ROBDDThreshold

# Time parameters
max_t = 120
steps = 120

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Number of components in the system
num_comps = len(alpha)

# Threshold system weights and threshold
a = [2.0, 4.0, 2.3, 3.5, 1.9]
b = 5.0

# Calculate the weight sum
asum = 0
for i in range(num_comps):
    asum += a[i]

# The ROBDD representation of the threshold system
```

```
sys = ROBDDSystem(ROBDDThreshold(a, asum, b))

# The reliability function of the system
def reliability(pp):
    result = sys.calculateReliability(pp)
    return result[1]

# The survival function of a Weibull-distribution
def weibull(aa, bb, t):
    if t >= 0:
        return exp(- (t / bb) ** aa)
    else:
        return 1.0

# Component reliability vector
p = np.zeros(num_comps)

# Time values and Birnbaum measures
time = np.linspace(0.0, max_t, steps)
IB = np.zeros((num_comps, steps))

# Calculate the Birnbaum measures
for step in range(steps):
    for i in range(num_comps):
        p[i] = weibull(alpha[i], beta[i], time[step])
    for i in range(num_comps):
        temp = p[i]
        p[i] = 1
        h1 = reliability(p)
        p[i] = 0
        h0 = reliability(p)
        p[i] = temp
        IB[i][step] = h1 - h0

# Plot the results
fig = plt.figure(figsize = (7, 4))
for i in range(num_comps):
    plt.plot(time, IB[i], label='IB(' + str(i+1) + ", t)")
plt.xlabel('Time')
plt.ylabel('Importance')
```

```
plt.title("Importance as a function of time")
plt.legend()
plt.show()
```

Script B.3.10

```
#########################################################################
#                                                                       #
#  This program plots I_B^{(i)}(t) as a function of t for an            #
#  undirected network system using Monte Carlo simulations of           #
#  the component lifetimes and calculating the resulting system         #
#  lifetime.                                                            #
#                                                                       #
#########################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Time parameters
max_t = 120
steps = 120

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5, 2.0, 2.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0, 55.0, 65.0]

# Number of components
num_comps = 7

# Birnbaum measures
IB = np.zeros((num_comps, steps))

# A component is identified as critical when t is in [L, U)
L = np.zeros(num_comps)
U = np.zeros(num_comps)

# Component state vector
x = np.zeros(num_comps)

# The coproduct function
```

```
def coprod(x1, x2):
    return x1 + x2 - x1 * x2

# The structure function given that component 3 is functioning
def phi_3_1(xx):
    return coprod(xx[2],xx[4])*coprod(xx[0],xx[1])*coprod(xx[5],xx[6]) \
        + (1 - coprod(xx[2], xx[4]))*coprod(xx[0]*xx[5], xx[1]*xx[6])

# The structure function given that component 3 is failed
def phi_3_0(xx):
    return coprod(xx[0] * xx[2], xx[1])*coprod(xx[4] * xx[5], xx[6])

# The structure function of the network system (using pivotal decomposition)
def phi(xx):
    return xx[3] * phi_3_1(xx) + (1 - xx[3]) * phi_3_0(xx)

# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    # Initialize component states
    for i in range(num_comps):
        x[i] = 1
    # Initialize system state and lifetime
    sys_state = phi(x)
    sys_life = 0.0
    # Determine the ordering of the failure times
    order = tt.argsort()
    # Initialize failure counter
    c = 0
    # Switch off one component at a time in the order of the failure times
    while (sys_state == 1) and (c < num_comps):
        x[order[c]] = 0
        sys_life = tt[order[c]]
        sys_state = phi(x)
        c += 1
    if sys_state == 0:
        return sys_life
    else: # This happens only for trivial systems where phi(0,...,0) = 1.
        return np.inf

# The simulated lifetimes of the components
```

```
t = np.zeros(num_comps)

# Time steps
time = np.linspace(0.0, max_t, steps)

# Run simulations
for k in range(num_sims):
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
    for i in range(num_comps):
        temp = t[i]
        t[i] = np.inf
        U[i] = sys_lifetime(t)
        t[i] = 0
        L[i] = sys_lifetime(t)
        t[i] = temp
    for step in range(steps):
        for i in range(num_comps):
            if L[i] <= time[step] and time[step] < U[i]:
                IB[i][step] += 1

# Calculate average importance values
for step in range(steps):
    for i in range(num_comps):
        IB[i][step] = IB[i][step] / num_sims

fig = plt.figure(figsize = (7, 4))
for i in range(num_comps):
    plt.plot(time, IB[i], label='IB(' + str(i+1) + ", t)")
plt.xlabel('Time')
plt.ylabel('Importance')
plt.title("Importance as a function of time")
plt.legend()
plt.show()
```

Script B.3.11

```
#########################################################################
#                                                                       #
#  This program estimates the Barlow-Proschan importance measure        #
#  for components in a system with given minimal path sets using        #
#  Monte Carlo simulations.                                             #
#                                                                       #
#########################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Number of components and minimal paths
num_comps = 5
num_paths = 4

# Minimal path sets
paths = [{1,4}, {1,3,5}, {2,3,4}, {2,5}]

# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    sys_life = 0.0
    for j in range(num_paths):
        path_life = np.inf
        for i in paths[j]:
            if tt[i-1] < path_life:
                path_life = tt[i-1]
        if path_life > sys_life:
            sys_life = path_life
    return sys_life
```

```
# The simulated lifetimes of the components
t = np.zeros(num_comps)

# The number of times the components are critical when failing
c = np.zeros(num_comps)

for k in range(num_sims):
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
    s = sys_lifetime(t)
    for i in range(num_comps):
        if t[i] == s:
            c[i] += 1

bp_imp = np.zeros(num_comps)
for i in range(num_comps):
    bp_imp[i] = c[i] / num_sims

comp = np.linspace(1, num_comps, num_comps)

fig = plt.figure(figsize = (7, 4))
plt.bar(comp, bp_imp, color ='gray', width = 0.4)
plt.xlabel("Components")
plt.ylabel("Importance")
plt.title("Barlow-Proschan importance")
plt.show()
```

Script B.3.12

```
#######################################################################
#                                                                     #
#  This program estimates the Barlow-Proschan importance measure      #
#  for components in a threshold system using Monte Carlo             #
#  simulations.                                                       #
#                                                                     #
#######################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Threshold system weights and threshold
w = [2.0, 4.0, 2.3, 3.5, 1.9]
threshold = 5.0

# Number of components and minimal paths
num_comps = 5

# Component state vector
x = np.zeros(num_comps)

# The structure function of the threshold system
def phi(xx):
    wsum = 0.0
    for i in range(num_comps):
        wsum += w[i] * xx[i]
    if wsum >= threshold:
        return 1
    else:
        return 0
```

```
# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    # Initialize component states
    for i in range(num_comps):
        x[i] = 1
    # Initialize system state and lifetime
    sys_state = phi(x)
    sys_life = 0.0
    # Determine the ordering of the failure times
    order = tt.argsort()
    # Initialize failure counter
    c = 0
    # Switch off one component at a time in the order of the failure times
    while (sys_state == 1) and (c < num_comps):
        x[order[c]] = 0
        sys_life = tt[order[c]]
        sys_state = phi(x)
        c += 1
    if sys_state == 0:
        return sys_life
    else: # This happens only for trivial systems where phi(0,...,0) = 1.
        return np.inf

# The simulated lifetimes of the components
t = np.zeros(num_comps)

# The number of times the components are critical when failing
c = np.zeros(num_comps)

for k in range(num_sims):
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
    s = sys_lifetime(t)
    for i in range(num_comps):
        if t[i] == s:
            c[i] += 1

bp_imp = np.zeros(num_comps)
for i in range(num_comps):
```

```
    bp_imp[i] = c[i] / num_sims

comp = np.linspace(1, num_comps, num_comps)

fig = plt.figure(figsize = (7, 4))
plt.bar(comp, bp_imp, color ='gray', width = 0.4)
plt.xlabel("Components")
plt.ylabel("Importance")
plt.title("Barlow-Proschan importance")
plt.show()
```

Script B.3.13

```
######################################################################
#                                                                    #
#  This program estimates the Barlow-Proschan importance measure     #
#  for components in an undirected network system using Monte        #
#  Carlo simulations.                                                #
#                                                                    #
######################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5, 2.0, 2.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0, 55.0, 65.0]

# Number of components
num_comps = 7

# Component state vector
x = np.zeros(num_comps)

# The coproduct function
def coprod(x1, x2):
    return x1 + x2 - x1 * x2

# The structure function given that component 3 is functioning
def phi_3_1(xx):
    return coprod(xx[2], xx[4]) * coprod(xx[0], xx[1]) * coprod(xx[5], xx[6]) \
        + (1 - coprod(xx[2], xx[4])) * coprod(xx[0] * xx[5], xx[1] * xx[6])

# The structure function given that component 3 is failed
def phi_3_0(xx):
    return coprod(xx[0] * xx[2], xx[1]) * coprod(xx[4] * xx[5], xx[6])
```

```
# The structure function of the network system (using pivotal decomposition)
def phi(xx):
    return xx[3] * phi_3_1(xx) + (1 - xx[3]) * phi_3_0(xx)

# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    # Initialize component states
    for i in range(num_comps):
        x[i] = 1
    # Initialize system state and lifetime
    sys_state = phi(x)
    sys_life = 0.0
    # Determine the ordering of the failure times
    order = tt.argsort()
    # Initialize failure counter
    c = 0
    # Switch off one component at a time in the order of the failure times
    while (sys_state == 1) and (c < num_comps):
        x[order[c]] = 0
        sys_life = tt[order[c]]
        sys_state = phi(x)
        c += 1
    if sys_state == 0:
        return sys_life
    else: # This happens only for trivial systems where phi(0,...,0) = 1.
        return np.inf

# The simulated lifetimes of the components
t = np.zeros(num_comps)

# The number of times the components are critical when failing
c = np.zeros(num_comps)

for k in range(num_sims):
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
    s = sys_lifetime(t)
    for i in range(num_comps):
        if t[i] == s:
            c[i] += 1
```

```
bp_imp = np.zeros(num_comps)
for i in range(num_comps):
    bp_imp[i] = c[i] / num_sims

comp = np.linspace(1, num_comps, num_comps)

fig = plt.figure(figsize = (7, 4))
plt.bar(comp, bp_imp, color ='gray', width = 0.4)
plt.xlabel("Components")
plt.ylabel("Importance")
plt.title("Barlow-Proschan importance")
plt.show()
```

Script B.3.14

```
#######################################################################
#                                                                     #
#  This program estimates the Natvig importance measure for           #
#  components in a system with given minimal path sets using          #
#  Monte Carlo simulations.                                           #
#                                                                     #
#######################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Number of components and minimal paths
num_comps = 5
num_paths = 4

# Minimal path sets
paths = [{1,4}, {1,3,5}, {2,3,4}, {2,5}]

# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    sys_life = 0.0
    for j in range(num_paths):
        path_life = np.inf
        for i in paths[j]:
            if tt[i-1] < path_life:
                path_life = tt[i-1]
        if path_life > sys_life:
            sys_life = path_life
    return sys_life
```

```
# The improvement in system lifetimes
# due to minimal repairs
z = np.zeros(num_comps)

# The simulated lifetimes of the components
t = np.zeros(num_comps)

# The simulated lifetimes of the components
# sampled from the conditional distribution
# given that they exceed the corresponding t
v = np.zeros(num_comps)

for k in range(num_sims):
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
        v[i] = 0.0
        while v[i] < t[i]:
            v[i] = weibullvariate(beta[i], alpha[i])
    s = sys_lifetime(t)
    for i in range(num_comps):
        temp = t[i]
        t[i] = v[i]
        z[i] += (sys_lifetime(t) - s)
        t[i] = temp

# Calculate the unstandardized Natvig measure
z_sum = 0.0
z_imp = np.zeros(num_comps)
for i in range(num_comps):
    z_imp[i] = z[i] / num_sims
    z_sum += z_imp[i]

# Standardize the measure so that it adds up to 1
nat_imp = np.zeros(num_comps)
for i in range(num_comps):
    nat_imp[i] = z_imp[i] / z_sum

comp = np.linspace(1, num_comps, num_comps)

fig = plt.figure(figsize = (7, 4))
```

```
plt.bar(comp, nat_imp, color ='gray', width = 0.4)
plt.xlabel("Components")
plt.ylabel("Importance")
plt.title("Natvig importance")
plt.show()
```

Script B.3.15

```
#######################################################################
#                                                                     #
#  This program estimates the Natvig importance measure              #
#  for components in a threshold system using Monte Carlo             #
#  simulations.                                                       #
#                                                                     #
#######################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0]

# Threshold system weights and threshold
w = [2.0, 4.0, 2.3, 3.5, 1.9]
threshold = 5.0

# Number of components and minimal paths
num_comps = 5

# Component state vector
x = np.zeros(num_comps)

# The structure function of the threshold system
def phi(xx):
    wsum = 0.0
    for i in range(num_comps):
        wsum += w[i] * xx[i]
    if wsum >= threshold:
        return 1
    else:
        return 0
```

```
# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    # Initialize component states
    for i in range(num_comps):
        x[i] = 1
    # Initialize system state and lifetime
    sys_state = phi(x)
    sys_life = 0.0
    # Determine the ordering of the failure times
    order = tt.argsort()
    # Initialize failure counter
    c = 0
    # Switch off one component at a time in the order of the failure times
    while (sys_state == 1) and (c < num_comps):
        x[order[c]] = 0
        sys_life = tt[order[c]]
        sys_state = phi(x)
        c += 1
    if sys_state == 0:
        return sys_life
    else: # This happens only for trivial systems where phi(0,...,0) = 1.
        return np.inf

# The improvement in system lifetimes
# due to minimal repairs
z = np.zeros(num_comps)

# The simulated lifetimes of the components
t = np.zeros(num_comps)

# The simulated lifetimes of the components
# sampled from the conditional distribution
# given that they exceed the corresponding t
v = np.zeros(num_comps)

for k in range(num_sims):
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
        v[i] = 0.0
```

```
        while v[i] < t[i]:
            v[i] = weibullvariate(beta[i], alpha[i])
    s = sys_lifetime(t)
    for i in range(num_comps):
        temp = t[i]
        t[i] = v[i]
        z[i] += (sys_lifetime(t) - s)
        t[i] = temp

# Calculate the unstandardized Natvig measure
z_sum = 0.0
z_imp = np.zeros(num_comps)
for i in range(num_comps):
    z_imp[i] = z[i] / num_sims
    z_sum += z_imp[i]

# Standardize the measure so that it adds up to 1
nat_imp = np.zeros(num_comps)
for i in range(num_comps):
    nat_imp[i] = z_imp[i] / z_sum

comp = np.linspace(1, num_comps, num_comps)

fig = plt.figure(figsize = (7, 4))
plt.bar(comp, nat_imp, color ='gray', width = 0.4)
plt.xlabel("Components")
plt.ylabel("Importance")
plt.title("Natvig importance")
plt.show()
```

Script B.3.16

```
#######################################################################
#                                                                     #
#  This program estimates the Natvig importance measure              #
#  for components in an undirected network system using Monte         #
#  Carlo simulations.                                                 #
#                                                                     #
#######################################################################

from math import *
from random import *
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 100000

# Weibull-parameters for the components
alpha = [2.0, 2.5, 3.0, 1.0, 1.5, 2.0, 2.5]
beta = [50.0, 60.0, 70.0, 40.0, 50.0, 55.0, 65.0]

# Number of components
num_comps = 7

# Component state vector
x = np.zeros(num_comps)

# The coproduct function
def coprod(x1, x2):
    return x1 + x2 - x1 * x2

# The structure function given that component 3 is functioning
def phi_3_1(xx):
    return coprod(xx[2], xx[4]) * coprod(xx[0], xx[1]) * coprod(xx[5], xx[6]) \
        + (1 - coprod(xx[2], xx[4])) * coprod(xx[0] * xx[5], xx[1] * xx[6])

# The structure function given that component 3 is failed
def phi_3_0(xx):
    return coprod(xx[0] * xx[2], xx[1]) * coprod(xx[4] * xx[5], xx[6])
```

```
# The structure function of the network system (using pivotal decomposition)
def phi(xx):
    return xx[3] * phi_3_1(xx) + (1 - xx[3]) * phi_3_0(xx)

# Calculate the system lifetime given the component lifetimes
def sys_lifetime(tt):
    # Initialize component states
    for i in range(num_comps):
        x[i] = 1
    # Initialize system state and lifetime
    sys_state = phi(x)
    sys_life = 0.0
    # Determine the ordering of the failure times
    order = tt.argsort()
    # Initialize failure counter
    c = 0
    # Switch off one component at a time in the order of the failure times
    while (sys_state == 1) and (c < num_comps):
        x[order[c]] = 0
        sys_life = tt[order[c]]
        sys_state = phi(x)
        c += 1
    if sys_state == 0:
        return sys_life
    else: # This happens only for trivial systems where phi(0,...,0) = 1.
        return np.inf

# The improvement in system lifetimes
# due to minimal repairs
z = np.zeros(num_comps)

# The simulated lifetimes of the components
t = np.zeros(num_comps)

# The simulated lifetimes of the components
# sampled from the conditional distribution
# given that they exceed the corresponding t
v = np.zeros(num_comps)

for k in range(num_sims):
```

```
    for i in range(num_comps):
        t[i] = weibullvariate(beta[i], alpha[i])
        v[i] = 0.0
        while v[i] < t[i]:
            v[i] = weibullvariate(beta[i], alpha[i])
    s = sys_lifetime(t)
    for i in range(num_comps):
        temp = t[i]
        t[i] = v[i]
        z[i] += (sys_lifetime(t) - s)
        t[i] = temp

# Calculate the unstandardized Natvig measure
z_sum = 0.0
z_imp = np.zeros(num_comps)
for i in range(num_comps):
    z_imp[i] = z[i] / num_sims
    z_sum += z_imp[i]

# Standardize the measure so that it adds up to 1
nat_imp = np.zeros(num_comps)
for i in range(num_comps):
    nat_imp[i] = z_imp[i] / z_sum

comp = np.linspace(1, num_comps, num_comps)

fig = plt.figure(figsize = (7, 4))
plt.bar(comp, nat_imp, color ='gray', width = 0.4)
plt.xlabel("Components")
plt.ylabel("Importance")
plt.title("Natvig importance")
plt.show()
```

B.4 Transmission of electronic pulses

Script B.4.1

```
#######################################################################
#                                                                     #
#  Python script calculating reliability at a given point of          #
#  time of a subsea network for transmitting electronic pulses.       #
#  See Section 9.2 of Huseby and Dahl (2021)                          #
#                                                                     #
#######################################################################

from math import *

import matplotlib.pyplot as plt
import numpy as np

def mcomb(n, k1, k2, k3):
    return factorial(n) / (factorial(k1) * factorial(k2) \
                    * factorial(k3) * factorial(n-k1-k2-k3))

def binomialDist(n, k, p):
    return comb(n, k) * p**k * (1-p)**(n-k)

def multinomialDist(n, k1, k2, k3, p1, p2, p3):
    return mcomb(n, k1, k2, k3) * p1**k1 * p2**k2 * p3**k3 * \
        (1 - p1 - p2 - p3)**(n-k1-k2-k3)

minTime: float = 0.0
maxTime: float = 50.0

# r[0] :             Failure rate pr time unit for the wires.
# r[1], ... , r[8] : Failure rates of the communication links
r = [0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]


# p[0] :             Survival probability for the wires.
# p[1], ... , p[8] : Survival probabilities of the comm. links
q = np.zeros(9)

reliability = np.zeros(6)
```

```
def initReliabilities():
    for i in range(6):
        reliability[i] = 0.0


def calcLinkProbabilities(t):
    for i in range(9):
        q[i] = exp(-r[i] * t)


def calcConnectionProbabilities():
    q_A = q[1] * q[6] * (q[4] + q[3] * q[5] - q[3] \
            * q[4] * q[5]) * (q[7] + q[8] - q[7] * q[8])\
     + q[1] * (1 - q[6]) * (q[4] * q[7] + q[3] * q[5] \
            * q[8] - q[3] * q[4] * q[5] * q[7] * q[8])
    q_B = q[2] * q[6] * (q[5] + q[3] * q[4] - q[3] \
            * q[4] * q[5]) * (q[7] + q[8] - q[7] * q[8]) \
     + q[2] * (1 - q[6]) * (q[5] * q[8] + q[3] * q[4] \
            * q[7] - q[3] * q[4] * q[5] * q[7] * q[8])
    q_AB_1_1 = q[1] * q[2] * (q[4] + q[5] - q[4] * q[5]) \
            * (q[7] + q[8] - q[7] * q[8])
    q_AB_1_0 = q[1] * q[2] * (q[4] * q[7] + q[5] \
            * q[8] - q[4] * q[5] * q[7] * q[8])
    q_AB_0_1 = q[1] * q[2] * q[4] * q[5] \
            * (q[7] + q[8] - q[7] * q[8])
    q_AB_0_0 = q[1] * q[2] * q[4] * q[5] \
            * q[7] * q[8]
    q_AB = q_AB_1_1 * q[3] * q[6] \
            + q_AB_1_0 * q[3] * (1 - q[6])\
            + q_AB_0_1 * (1-q[3]) * q[6] \
            + q_AB_0_0 * (1-q[3]) * (1 - q[6])
    return q_AB, q_A, q_B


def calcReliabilities(t):
    initReliabilities()
    calcLinkProbabilities(t)
    result = calcConnectionProbabilities()   # q_AB, q_A, q_B
    p_AB = result[0]                         # p_AB = q_AB
```

```
    p_A = result[1] - result[0]             # p_A = q_A - q_AB
    p_B = result[2] - result[0]             # p_B = q_B - q_AB
    for y1 in range(7):
        for y2 in range(7):
            for y3 in range(6):
                for y4 in range(6 - y3):
                    for y5 in range(6 - y3 - y4):
                        py1 = binomialDist(6, y1, q[0])
                        py2 = binomialDist(6, y2, q[0])
                        py345 = multinomialDist(5, \
                                y3, y4, y5, p_AB, p_A, p_B)
                        w1 = min(y1, y4)
                        w2 = min(y2, y5)
                        w3 = min((y1-w1) + (y2-w2), y3)
                        phi = w1 + w2 + w3
                        reliability[phi] += (py1 * py2 * py345)


time = np.zeros(101)
rel1 = np.zeros(101)
rel2 = np.zeros(101)
rel3 = np.zeros(101)
rel4 = np.zeros(101)
rel5 = np.zeros(101)

for s in range(101):
    t = minTime + (maxTime - minTime) * s / 100
    time[s] = t
    calcReliabilities(t)
    rel5[s] = reliability[5]
    rel4[s] = rel5[s] + reliability[4]
    rel3[s] = rel4[s] + reliability[3]
    rel2[s] = rel3[s] + reliability[2]
    rel1[s] = rel2[s] + reliability[1]

plt.plot(time, rel1, label='P(phi >= 1)')
plt.plot(time, rel2, label='P(phi >= 2)')
plt.plot(time, rel3, label='P(phi >= 3)')
plt.plot(time, rel4, label='P(phi >= 4)')
plt.plot(time, rel5, label='P(phi >= 5)')
```

```
plt.xlabel('time')
plt.ylabel('Reliability')
plt.title("Reliabilities as functions of time")
plt.legend()
plt.show()
```

Bibliography

[1] Akers, S. B., Binary Decision Diagrams, *IEEE Transactions on Computers*, C-27-6, 509–516, 1978.

[2] Baarholm, G.S., Haver, S. and Økland, O.D. Combining contours of significant wave height and peak period with platform response distributions for predicting design response. *Marine Structures* 23, 147–163, 2010.

[3] Ball, M. O., The complexity of network reliability computations. *Networks*, 10, 253–278, 1977.

[4] Banks, J., Carson, J., Nelson, B.L., and Nicol, D. *Discrete-Event System Simulation (4th Edition)*. Prentice-Hall International Series in Industrial and Systems, 2004.

[5] Barlow, R. E. and Proschan, F., Importance of system components and fault tree events, *Stochastic Processes and their Applications*, 3, 153–173, 1975.

[6] Barlow, R. E. and Proschan, F., *Statistical Theory of Reliability and Life Testing*, To Begin With, Silver Spring, MD, 1981.

[7] Barlow, R. E. and Proschan, F., *Mathematical Theory of Reliability*, SIAM, Philadelphia, 1996.

[8] Birnbaum, Z. W., On the Importance of Different Components in a Multicomponent System, Multivariate Analysis - II, Edited by P. R. Krishnaiah, Academic Press, 581–592, 1969.

[9] Bobbio, A., Terruggia, R., Binary decision diagrams in network reliability analysis, *1st IFAC Workshop on Dependable Control of Discrete Systems*, 6 pages, 2007.

[10] Broström, G. and L. Nilsson, L., Acceptance/rejection sampling from the conditional distribution of independent discrete random variables, given their sum, *Statistics*, 34, 247, 2000.

[11] Brace, K. S., Rudell, R. L., Bryant, R. G., Efficient Implementation of a BDD Package, *27th ACM/IEEE Design Automation Conference*, 0738, 40–45, 1990.

[12] Bryant, R. G., Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, C-35-8, 677–691, 1986.

[13] Burch, J.R., E.M. Clarke, K.L. McMillan, D.L. Dill and J. Hwang, Symbolic model checking: 10^{20} states and beyond, *Information and Computation*, 98, 142–170, 1992.

[14] Calkin, N. J., Edds, J. D. and Shier, D. R. Cancellation in cyclic consecutive systems, *Journ. of Comp. and Appl. Math.*, 142, 13–26, 2002.

[15] Cancela, H. and El Khadari, M., A recursive variance-reduction algorithm for estimating communication-network reliability, *IEEE Transactions on Reliability*, R-44, 1995.

[16] Cancela, H. and El Khadari, M., Series-parallel reductions in Monte Carlo network reliability evaluation, *IEEE Transactions on Reliability*, R-47, 1998.

[17] Cancela, H. and El Khadari, M., The recursive variance-reduction simulation algorithm for network reliability evaluation, *IEEE Transactions on Reliability*, R-52, 2003.

[18] Chang, G. J., Wang, F. H., and Cui, L., Reliabilities of Consecutive-k Systems, In: *Handbook of Reliability Engineering*, Springer Verlag, London, 37–59, 2003.

[19] Chiang, D. T. and Niu, S. C., Reliability of Consecutive-k-out-of-n: F System, *IEEE Transactions on Reliability*, R-30 (1), 87–89, 1981.

[20] Coudert, O. and Madre, J., Fault tree analysis: 10^{20} prime implicants and beyond, In *Proceedings of the Annual Reliability and Maintainability Symposium, ARMS'93, Atlanta NC, USA*, 1993.

[21] Derman, C., Lieberman, G. J. and Ross, S. M., On the Consecutive-k-out-of-n: F System, *IEEE Transactions on Reliability*, R-31 (1), 57–63, 1982.

[22] Ditlevsen, O., Stochastic model for joint wave and wind loads on offshore structures, *Structural Safety*, 24, 139–163, 2002.

[23] Fontaine, E., Orsero, P., Ledoux, A., Nerzic, R., Prevesto, M. and Quiniou, V., Reliability analysis and response based design of a moored FPSO in West Africa, *Structural Safety*, 41, 82–96, 2013.

[24] Fishman, G. S., A Monte Carlo sampling plan for estimating network reliability, *Operations Research*, 34, 1986.

[25] Fishman, G. S., A comparison of four Monte Carlo methods for estimating the probability of $s-t$ connectedness, *IEEE Transactions on Reliability*, R-35, 1986.

[26] Gao, K., Liu, X., Wang, H., Ma, X., Peng, R., A New Iterative Approach to Evaluate the Reliability of a Combined Consecutive k and v-out-of-n System, *International Journal of Performability Engineering*, 14 (12), 2983–2993, 2018.

[27] Garey, M. R. and Johnson, D. S., Computers and Intractibility: A Guide to the theory of NP-Completeness, W. H. Freeman, San Francisco, 1979.

[28] Glasserman, P. and Yao, D. D., *Monotone Structure in Discrete-event Systems*, John Wiley and Sons, Inc., 1994.

[29] Glasserman, P., *Monte Carlo Methods in Financial Engineering*, Springer Verlag, 2004.

[30] Haver, S., On the joint distribution of heights and periods of sea waves., *Ocean Engineering* 14, 359–376, 1987.

[31] Haver, S. and Winterstein, S., Environmental contour lines: A method for estimating long term extremes by a short term analysis., *Transactions of the Society of Naval Architects and Marine Engineers* 116, 116–127, 2009.

[32] Huseby, A. B., A unified theory of domination and signed domination with application to exact reliability computations, Statistical Research Report No. 3, Department of Mathematics, University of Oslo, 1984.

[33] Huseby, A. B., Domination theory and the Crapo β-invariant, *Networks*, 19, 135–149, 1989.

[34] Huseby, A. B. , Naustdal, M. and Vårli, I. D.,System reliability evaluation using conditional Monte Carlo methods,Department of Mathematics, University of Oslo, 2004.

[35] Huseby, A. B., Oriented matroid systems, *Discrete Applied Mathematics*, 159, 1, 31–45, 2011.

[36] Huseby, A. B., Vanem, E., Natvig, B., A new approach to environmental contours for ocean engineering applications based on direct Monte Carlo simulations, *Ocean Engineering*, 60, 124–135, 2013.

[37] Huseby, A. B., Vanem, E., Natvig, B., A new Monte Carlo method for environmental contour estimation, In *Safety and Reliability : Methodology and Applications*, Proceedings of the European safety and reliability Conference, Taylor & Francis. Chapter 270, 2091–2098, 2015.

[38] Huseby, A. B., Vanem, E., Natvig, B., Alternative environmental contours for structural reliability analysis, *Structural Safety*, 54, 32–45, 2015.

[39] Huseby, A. B. and Vanem, E. and Eskeland, K., Evaluating properties of environmental contours, In *Safety and Reliability, Theory and Applications. Proceedings of the European safety and reliability Conference*, CRC Press. Chapter 893, 2101–2109, 2017.

[40] Jonathan, P. and Ewans, K. and Flynn, J., On the estimation of ocean engineering design contours, *Journal of Offshore Mech. Arct. Eng*, 136(4), 8 pages, 2011.

[41] Jonathan, P., Ewans, K. and Flynn, J. On the estimation of ocean engineering design contours. In: *Proc. 30th International Conference on Offshore Mechanics and Arctic Engineering (OMAE 2011)* American Society of Mechanical Engineers (ASME), 2011.

[42] Hardy, G., C. Lucet and N. Limnios, Computing all-terminal reliability of stochastic networks by bdds, In: *Proc Applied Stochastic Modeling and Data Analysis*, ASMDA, 2005.

[43] Klebaner, F. C., *Introduction to stochastic calculus with applications.* Imperial College Press, 2005.

[44] Knuth, D. E., The art of computer programming, Vol. 2, Seminumerical algorithms, Addison-Wesley, Reading, Mass. second edition, 1981.

[45] Lee, C.Y., Representation of Switching Circuits by Binary-Decision Programs, Bell System Technical Journal, 38, 985–999, 1959.

[46] Leira, B. J., A comparison of stochastic process models for definition of design contours, *Structural Safety*, 30, 493–505, 2008.

[47] Lindqvist, B. H. and Taraldsen, G., Monte Carlo Conditioning on a Sufficient Statistics, Preprint Statistics 9, Norwegian University of Science and Technology, 2001.

[48] Moan, T., Development of accidental collapse limit state criteria for offshore structures, *Structural Safety*, 31, 124–135, 2009.

[49] Natvig, B., A suggestion for a new measure of importance of system components, *Stochastic Processes and their Applications*, 9, 319–330, 1979.

[50] Natvig, B., Pålitelighetsanalyse med teknologiske anvendelser, lecture notes, Department of Mathematics, University of Oslo, 3rd ed. 1998.

[51] Natvig, B., *Multistate Systems Reliability Theory with Applications*, John Wiley and Sons Ltd., Chichester, 2011.

[52] Naustdal, M., Forbedrede betingede Monte Carlo metoder i pålitelighetsberegninger (in Norwegian), Cand. Scient. thesis, Department of Mathematics, University of Oslo, 2001.

[53] Newman, M., The structure and function of complex networks, *SIAM Rev*, 45, 167–256, 2003.

[54] Nilsson, L., On the simulation of conditional distributions in the Bernoulli case, Research report 14, Department of Mathematical Statistics, Umeå University, Sweden 1997.

[55] Pflug, G., Some remarks on the value-at-risk and the conditional value-at-risk, In Uryasev, S. P. (Ed.) *Probabilistic Constrained Optimization. Methodology and Applications.* Norwell, Kluwer Academic Publishers, 272–281, 2000.

[56] Rockafellar, R. T., Coherent Approaches to Risk in Optimization Under Uncertainty, In *OR Tools and Applications: Glimpses of Future Technologies*, INFORMS Tutorials in Operations Research, Ch. 3, 38–61, 2007.

[57] Rockafellar, R. T. and Royset, J. O., On buffered failure probability in design and optimization of structures, *Reliability Engineering and System Safety*, 95, 499–510, 2010.

[58] Rockafellar, R. T. and Uryasev, S. P., Optimization of conditional value-at-risk, *Journal of Risk*, 2, 21–42, 2000.

[59] Rosenblatt, M., Remarks on a Multivariate Transformation, *The Annals of Mathematical Statistics*, 23, No 3, 470–472, 1952.

[60] Rosenthal, A., Computing the reliability of complex networks, *SIAM, Journal of Applied Math.*, 32, 384–393, 1977.

[61] Rauzy, A., New algorithms for fault tree analysis, *Reliability Engineering and System Safety*, 40, 203–211, 1993.

[62] Rauzy, A., Binary Decision Diagrams for Reliability Studies, In: *Handbook of Performability Engineering, Ed. Mishra, K. B.*, Springer Verlag, London, Ch. 25, 381–396, 2008.

[63] Satyanarayana, A., A unified formula for analysis of some network reliability problems, *IEEE Trans. Reliability*, 31, 23–32, 1982.

[64] Satyanarayana, A. and Chang, M. K., Network Reliability and the Factoring Theorem, *Networks*, 13, 107–120, 1983.

[65] Prabhakar, A. and Satyanarayana, A., New topological formula and rapid algorithm for reliability analysis of complex networks, *IEEE Trans. Reliability*, 27, 82–100, 1978.

[66] Triantafyllou, I. S., Consecutive-Type Reliability Systems: An Overview and Some Applications, *Journal of Quality and Reliability Engineering*, 1–20, 2015.

[67] Vanem, E. and Bitner-Gregersen, E., Stochastic modelling of long-term trends in wave climate and its potential impact on ship structural loads, *Applied Ocean Research*, 37, 235–248, 2012.

[68] Vanem, E. and Bitner-Gregersen, E., Alternative Environmental Contours for Marine Structural Design – A Comparison Study, *Journal of Offshore Mechanics and Arctic Engineering*, 137, 051601-1–051601-8, 2015.

[69] Vårli, I. D. Forbedrede Monte Carlo metoder i pålitelighetsberegninger, (in Norwegian), Cand. Scient. thesis, Department of Mathematics, University of Oslo, 1996.

[70] Winterstein, S., Ude, T., Cornell, C., Bjerager, P. and Haver, S., Environmental parameters for extreme response: Inverse FORM with omission factors. In: *Proc. 6th International Conference on Structural Safety and Reliability*, 1993.

[71] Winther, R., Threshold systems, PhD-thesis, University of Oslo, 1996.

Index