Solutions and Heuristics for Troubleshooting with Dependent Actions and Conditional Costs

Ottosen, Thorsten Jørgen

Publication date:
2012

Document Version
Accepteret manuscript, peer-review version

Solutions and Heuristics for Troubleshooting with Dependent Actions and Conditional Costs

Thorsten Jørgen Ottosen

Ph.D. Dissertation

The Machine Intelligence Group
Department of Computer Science
Aalborg University
Denmark

To my mother for taking so good care of our family,
and to my father for being such an inspiration.

Preface

> Life can only be understood backwards; but it must be lived forwards.
> –*Søren Kirkegaard*

We cannot always predict how our lives turn out. Some call it fate. My life shifted in a somewhat different direction ten years ago. I was completing my second year in computer science and was looking for a summer time job to earn some money. I had contacted IBM and other big companies and it seemed like I would spend most of my vacation in one of these companies. After lunch we were walking back to our group room when my good friend and student collegue Dennis Kristensen said: "Why don't you just apply for this job?". He was referring to a small job poster from HP which had a small research-unit within our university. In fact, they were located in the same building as our group room. I got a job interview with Claus Skaaning who was leading the unit and they hired me even though the job poster was old and they had already found a person for that position. Half a year later, the research unit became a new independent company, Dezide, and Claus Skaaning took all us student programmers with him as partners in Dezide. Before that date I had no idea that I would be writing a Ph.D. in artificial intelligence. Here we are ten years later. Kierkegaard's words make more sense than ever.

This thesis is the main result of my three and a half years of Ph.D. study on decision theoretic troubleshooting at the Machine Intelligence Group here in Aalborg. The work began in August 2007 and ended in January 2011. It can also be seen as the culmination of no less than nine and a half years of study in computer science here at our Department of Computer Science in Aalborg, of which about four years have been mainly devoted to decision theoretic troubleshooting. When I started university in 1999, it was hard to imagine that it would end with this dissertation.

In this work I try to present decision theoretic troubleshooting as detailed as possible, incorporating elements of previous, current and future research. The new results stem from a series of articles that I have written together with my supervisor Professor Finn V. Jensen:

1. Better Safe than Sorry—Optimal Troubleshooting through A* Search with Efficiency-based Pruning (Ottosen and Jensen, 2008b)

2. A* Wars: The Fight for Improving A* Search for Troubleshooting with Dependent Actions (Ottosen and Jensen, 2008a)
3. The Cost of Troubleshooting Cost Clusters with Inside Information (Ottosen and Jensen, 2010)
4. When to Test? Troubleshooting with Postponed System Test (Ottosen and Jensen, 2011)

These papers contain a mix of emperical and theoretical results: (1) is purely experimental, (3) is purely theoretical, whereas (2) and (4) falls in both categories. With reference to Pasteur's Quadrant, I believe we have found a good balance between theoretical results and algorithms of practical value. Articles (1) and (2) form the basis of Chapter 5, whereas articles (3) and (4) correponds to Chapter 6 and 7, respectively.

In this thesis you will also find a few results and discussions not published before because they were either not too relevant to the above papers or because they constitute work in progress. However, I believe these results and discussions still carry some insights that are of theoretical and practical value to other people in research as well as in industry.

Acknowledgements

This work was funded by the Faculties of Engineering, Science and Medicine at Aalborg University (position # 562/06-FS-27867) and for the last half year by the Machine Intelligence Group. Therefore I give a big thanks to these two institutions and not least to the hard working tax payers that fund them. I am very happy to have been part of a group with such a strong history in Bayesian networks and decision theory. I would like to thank all my collegues for providing an interesting research-environment. People like Manfred Jaeger, Thomas D. Nielsen, Yifeng Zeng, Uffe Kjærulff and Kristian G. Olesen have been more than kind to answer my stupid questions and discuss various issues with me. Thanks also to Sven Skyum. A special thanks goes to my long-time friend and Ph.D.-collegue Nicolaj Søndberg-Jeppesen who had the ungrateful task of reviewing drafts of this thesis.

My application was accompanied by letters of recommendation from Claus B. Madsen, Claus Skaaning, Beman Dawes, Matthew Wilson and Bjarne Stroustrup. I am very grateful for your support, and I am sure it was an important reason for my successful application. I'm also grateful to my old collegues in Dezide, including Claus Skaaning, Esben Rasmussen, Frank Hahn, and Lars Hammer. You have been very helpful in giving feedback on the directions of this research. Furthermore, Dezide have also provided real-world models on which to carry out experiments.

During my Ph.D., I had the pleasure of supervising a group of very talented 3rd year students in 2008. The group consisted of Ron Cohen, Peter Finderup, Lasse Jacobsen, Morten Jacobsen, Marting Vang Jørgensen, and Michael Harkjær Møller. I learned a lot from this supervison and was very happy that you could actually extend my experiments from paper (2) above.

My "study abroad" period was spend at the Institute of Information Theory and Automation (ÚTIA) at the Academy of Sciences of the Czech Republic in Prague during spring 2009. Special thanks goes to Jiří Vomlel from the Department of Decision-Making Theory for making my stay possible and enjoyable, and for being my supervisor for this half year. My skills matured substantially while being in Prague, and working in a different field than troubleshooting was just the breeze of fresh air that I needed. Our joint work is nowhere to be seen in this thesis, but the interested reader may refer to (Ottosen and Vomlel, 2010b) and (Ottosen and Vomlel, 2010a). Thanks also go to my other collegues at ÚTIA for making my stay so enjoyable, including Michal Cervinka, Václav Kratochvíl, Tomáš Kroupa, Martin Kružík, František Matúš, Jirina Vejnarova, Jirí Outrata, Milan Studený, Petr Lachout, and Radim Jiroušek. Special thanks go to František Matúš for giving a wonderful course on Markov distributions and to Petr Lachout for tutoring me on linear and non-linear optimization.

The last person I would like to thank is my supervisor Professor Finn V. Jensen. It has been a distinguished pleasure to work with you, and a great honour to have you as my mentor. Because of you, I now understand the true meaning of the famous quote:

> No, no, you're not thinking; you're just being logical.
> –*Niels Bohr*

Thorsten Jørgen Ottosen,
Aalborg, March 8, 2012.

Contents

Chapter 1

Introduction

> Detection is, or ought to be, an exact science, and should be treated in the same cold and unemotional manner.
> –*Sherlock Holmes*

1.1 What is Troubleshooting?

This thesis is about **decision theoretic troubleshooting** (or simply **troubleshooting**) which in many ways can be seen as an exact science for "detection". However, at the core of decision theoretic troubleshooting is not only the goal of arriving at the true diagnosis or conclusion of a given problem, but also a desire to do so as efficiently as possible. In layman's terms, the faster (or cheaper) we arrive at the proper conclusion, the better. But decision theoretic troubleshooting goes further than that because it establishes a principled way to solve the problem while concurrently narrowing down the potential causes of the problem. As such, its detection and problem solving mixed together in the (ideally) most optimal manner. In modern terms this is called **decision theory**.

When we build a computer system based on decision theory, it often takes the form of an **expert system** which gives *advice* to humans in certain contexts (e.g. to advice a doctor that it would be most beneficial to make a CT-scan of the patient) or which *semi-automates* some task that is normally carried out manually by humans (e.g. to provide a customer with technical support about his printer over the phone). We typically create the system as a synthesis between human expert knowledge and the enormous computational power of a modern computer. This fusion is enabled by an underlying mathematical model, and if this model is *sound*, the resulting system can potentially reason far better than any human.

An example of a very successful expert system is given in Figure 1.1 which shows an intelligent guide for printer troubleshooting. The aim was to reduce service costs as well as improve customer satisfaction. The printer

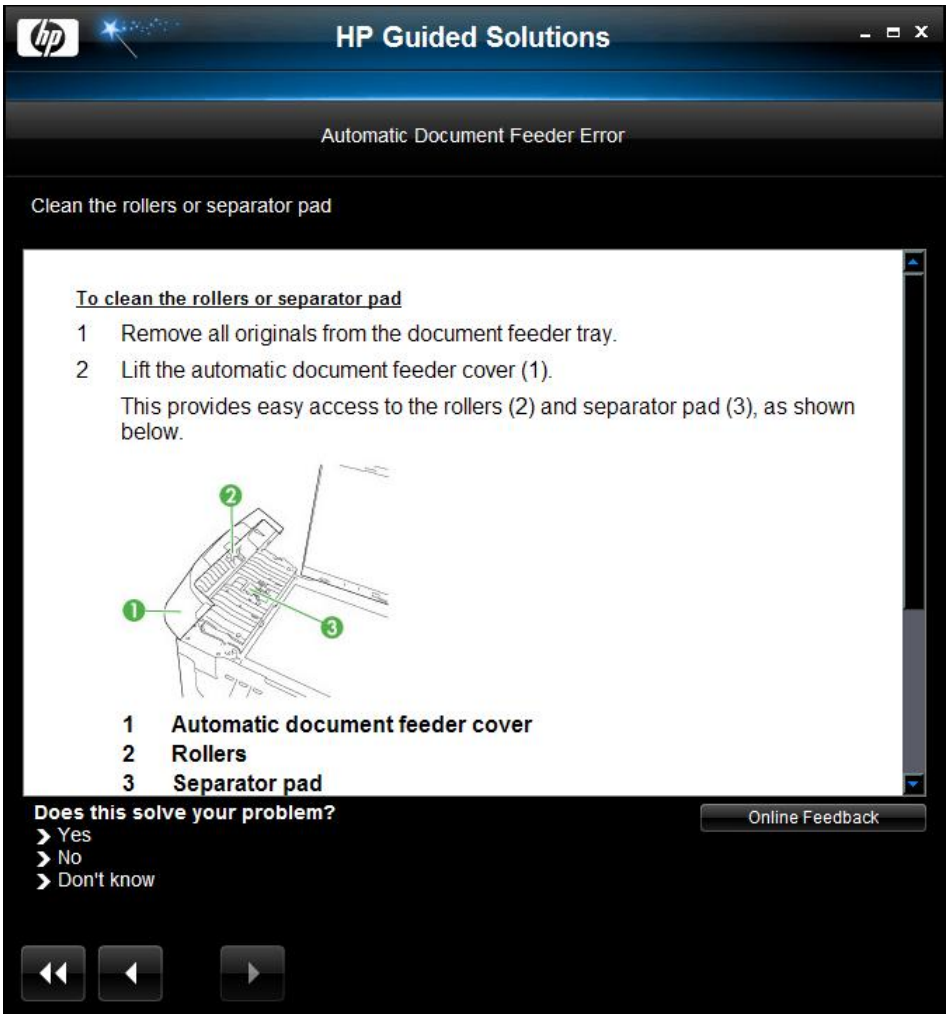


Figure 1.1: Screenshot of HP's self-service printer troubleshooter which helps users to fix the problem on their own (Dezide, 2010).

is capable of signaling certain error codes to the computer which then starts up the troubleshooter. The troubleshooter may first automatically collect information about the system to improve the accuracy of the pending repair process, and in certain circumstances (if the problem is software related) it may even fix the problem automatically. The troubleshooter automatically sends information about successful troubleshooting sessions to a central server which then uses these sessions to update the accuracy of the model. In turn, the new models on the server are automatically downloaded to the user's computer. In this manner, the system **learns** over time and users implicitly help each other to improve the performance of the system.

Let us briefly discuss the elements of a **troubleshooting model** (we shall return to this more formally in Chapter 3). Basically we describe a problem and its solution via three components: (a) causes of the problem, (b) observations which may narrow down the potential causes, and (c) repair actions which we may perform to fix the problem. For each observation or action we also associate a cost which describes the resources required to make the observation or perform the action. As a final component, the model describes the probabilistic relationship of the domain via some form of probabilistic model over causes, observations and actions.

1.2 Why Troubleshooting?

We can identify at least three reasons for conducting a formal study of troubleshooting:

1. Problem solving tasks take up *huge* amounts of work hours worldwide. For some people, it is their job, for other people it is a constant annoyance *hindering* their real job. Either way, there is an enormous potential for saving time, money and frustration.

2. It enables a *structured* approach to problem solving, in which we build and maintain knowledge bases on a computer. We capture our current knowledge in a computer model, which we refine over time. In turn, these knowledge bases put expert knowledge in the hands of non-experts, greatly enhancing their potential for problem solving.

3. Problem solving tasks are *difficult* and computers are far better than humans for optimizing these tasks. Computers can do billions of calculations per second, and with an appropriate mathematical model, the computer can potentially perform far better reasoning than any human.

To clarify case (1) above, we consider the following situations. If we think about it for a moment, we realize that most of us do problem solving on a daily basis. Programmers debug their programs and fiddle with installation of dependencies, utilities and servers. Secretaries try to make the word processor do as they like—for some reason the margin is printed incorrectly. Factory workers try to repair or calibrate a complicated machine that malfunctions. And Ph.D.-students try to figure out why their supervisor's Latex installation does not allow him to use certain symbols. The list is practically endless.

To elaborate on case (2) above, we give these considerations. When we try to solve all these daily problems, it is characteristic that we are not experts in solving the particular problem. We just need to make the device or

system work again, so we can get on with our job. As we gain experience, we can solve the problems faster, but because we tend to use ad hoc approaches like "trial and error", learning is a slow process. We tend to forget solutions over time and as a result, we end up starting from scratch each time. What is really needed is a structured approach where previous efforts are saved, reused and improved upon.

In general we can say that the simpler the problem is, the easier it is to make an expert system that guides inexperienced people to a solution, and the more complicated the problem is, the larger is the potential benefits of making an expert system. However, regardless of how complicated the domain is, there is a great potential for optimizing problem solving tasks by structuring, reusing and improving existing knowledge. That is why we need to study troubleshooting.

1.3 Troubleshooting and Simplicity

Expert systems come in many flavours and for a variety of domains. However, we may define the common *design goals* of expert systems as follows:

> The aim of any expert system is to establish a balance between user knowledge and data on one side, and model accuracy and computational efficiency on the other side.

This definition implies that there is no perfect expert system for all possible scenarios. We must always find a compromise, striking a balance between how difficult is to build and maintain the system compared to how accurately the underlying (mathematical) model describes the domain and how tractable the model is.

In decision theoretic troubleshooting we *deliberately* choose to use fairly simple models such that it is technically possible and economically affordable for domain experts to build and maintain these models. If expert systems should ever reach a wider audience, we see this simplicity as a prerequisite. In general we might summarize the benefits of this approach as follows:

1. **Small parameter space:** Any model depends on parameters, and when these parameters must be specified by a domain expert, a simpler model usually yields better estimates of these parameters. We often formulate this as easier **knowledge acquisition**.

2. **Computational efficiency:** In many cases, the expert system must deliver real-time answers. (For example, this is the case with the printer troubleshooter in Figure 1.1.) A complicated model may not admit good approximations in real-time.

3. **Model encapsulation:** A simple model may be completely shielded by a tool, which maximizes the number of potential domain experts that can use the system. For troubleshooting models, such tools have existed for a decade (Skaanning et al., 2000). For other domains in artificial intelligence, such tools are slowly emerging (see for example (ExpertMaker, 2010)).

4. **Model validation and tuning:** Domain experts often desire to validate that the model perform in accordance with their expectations (to programmers this would be similar to unit testing a piece of code). A complicated model makes it harder to derive answers to questions like "why is this step suggested at this point?". Furthermore, the domain expert might want to tell the system "the order of these two steps should be reversed, please adjust the parameters of the model for me".

Knowledge acquisition is a very real problem for complicated models. For troubleshooting models, the problem is quite manageable (Skaanning et al., 1999). For probabilistic models, people have suggested a number of ways to reduce the number parameters while introducing as few new assumptions as possible. Examples include the **noisy OR gates** (Pearl, 1986; Srinivas, 1993), and more recently the so-called **nonimpeding noisy-AND tree** (Xiang, 2010).

Of course, a simple model can also be a disadvantage in certain situations. We can think of at least two potential disadvantages:

1. **The assumptions of the model are unreasonable.** The effect of this is that the simple model leads to incorrect conclusions. On a case by case basis one might determine that certain assumptions are important to remove while others can be argued not to affect the performance of the system significantly.

2. **Plenty of data is available to learn a detailed model.** If this is the case, the model may be learned without human intervention, and advantage (1) and (3) above disappears somewhat for the simple model.

As such, we have two main forces that pull in opposite directions. The more complicated the model is, the better it can handle difficult scenarios, but the harder it is to specify correct parameters. On the other hand, the simpler the model, the easier it is to specify the parameters correctly, but the model will be appropriate for fewer scenarios. What is better: a correct model with inaccurate parameters, or an incorrect model with accurate parameters? There is no universal principle we can apply for all situations. However, Einstein's maxim:

"Make everything as simple as possible, but not simpler."

makes a good guideline. For a given domain, we should always seek to have a model that carries all the assumptions that are reasonable for the domain, but no other assumptions. If the model does have assumptions that are not inherent in the domain, we must justify that either (a) these assumptions cannot affect the accuracy of the model significantly, or (b) any model without these assumptions has other significant drawbacks with consequences that are worse (or equally bad).

Even though we have called troubleshooting models for *simple*, they are, however, among the most sophisticated models for a variety of applications. This is partly due to the fact that they utilize a sound probabilistic basis, and partly because many real-world situations fit the various assumptions quite well. Furthermore, real-time scenarios often preclude the use of even more advanced models. And despite their simplicity, most domains require use of troubleshooting models where optimal solutions are intractable to compute.

Already today, some of the techniques you will find in this thesis are being used in applications as diverse as troubleshooting of printers, customer self-service for internet service providers, cost-sensitive diagnosis in distributed systems (Rish et al., 2005) and repair of complicated equipment like windmills. With this thesis we hope to widen the range of applications of troubleshooting even further.

1.4 Thesis Overview

- In Chapter 2 on page 9, we shall deal mainly with probability theory and Bayesian networks. These two concepts provide a sound basis on which to describe troubleshooting models that can cope with uncertainties in a domain.

- In Chapter 3 on page 25 we review modern decision theory, including decision trees. We discuss several aspects including decision theoretic troubleshooting and various forms of influence diagrams. This comparison motivates the design choices made in decision theoretic troubleshooting.

- In Chapter 4 on page 59 we take an in-depth look at classical solution methods like depth-first search, A^* and AO^*. We discuss the complexity that these algorithms have for general troubleshooting problems.

- In Chapter 5 on page 83 the topic is troubleshooting with dependent actions. The key problem is that more than one action may repair

the same fault, and we must try to schedule actions under this assumption. We review heuristics for this problem and discuss several theoretical properties of solution methods. The chapter also describes empirical evaluations of these solution methods.

- In Chapter 6 on page 113 we take a somewhat theoretical look at troubleshooting with cost clusters. The idea is to group actions together if they share the same enabling work that must be carried out before any of the actions in the cluster can be performed. The majority of the chapter is devoted to prove optimality of existing and new polynomial-time algorithms.

- Finally, in Chapter 7 on page 135 we investigate new heuristics for scheduling actions when the system test cost is non-trivial. The heuristics are all motivated by theoretical considerations. We describe naive, but easy-to-implement, solution methods and use them to make an empirical evaluation of the heuristics.

Chapter 2

Probability Theory and Bayesian Networks

> Probability is not really about numbers, it is about the structure of reasoning.
> –*Glenn Shafer*

> In solving a problem of this sort, the grand thing is to be able to reason backwards. That is a very useful accomplishment, and a very easy one, but people do not practise it much.
> –*Sherlock Holmes*

In decision theoretical troubleshooting we are faced with a problem that needs to be solved by applying solution actions and by posing questions that gather information about the problem. The premise is that after each action we can observe whether the problem was solved. The domain is assumed to be uncertain, that is, solution actions may be imperfect (fail to repair a component) and information might be non-conclusive.

In this chapter we shall therefore review the basics of probability theory and Bayesian networks. Together, these two concepts provide a sound and efficient framework for reasoning and hypothetizing about an uncertain world before and after observations have been made.

2.1 Uncertain Worlds

As researchers we seek mathematically sound theories that can describe entities and their relationship in our surrounding world. The surrounding world is often highly **uncertain**, that is, we can not be certain of the true past, current, or future state of the world. This uncertainty can stem from many sources, for example:

1. An action has **non-deterministic** consequences, e.g., only 8 out of 10 cancer patients respond positively to a certain chemotherapy.

2. The world is only **partially observable**, e.g., it might be economically infeasible or simply impossible with current technology to observe the number of cod in the North Sea.

These two examples illustrate that a common source of uncertainty is **abstraction**. In principle, we expect that it will be possible to determine exactly if a cancer treatment works for a given patient, we just do not have the technological and medical insights currently. We are therefore *forced* to describe the world to our best ability. The second example is similar: in principle—given enough resources—we could count all the cod in the North Sea. But there are also cases where the uncertainty can never be removed by a deeper understanding of the problem. A typical example is card games in which we can *never* observe the cards on the opponents' hands. Therefore uncertainty is a fundamental concept that we cannot avoid dealing with. And therefore we must demand from any mathematical theory (in this context) that it handles uncertainty *rigorously* if it is to be useful for anything except the most trivial problems.

2.2 Basic Probability Theory

The most successful theory in artificial intelligence for describing an uncertain world is **probability theory**. The first part in our description of an uncertain world is to describe the entities that we wish to reason about. These entities may be described via **propositions** like "it will rain tomorrow at 12 a.m." or "replacing the battery will enable the car to start". The second part is then to assign a **probability** (or **degree of belief**) to these propositions stating one's subjective opinion or statistical knowledge. The first part can be seen as the *qualitative* aspect telling us *what* we wish to reason about, and the second part can be seen as a *quantitative* aspect describing our current belief about the first part.

Formally, we describe a particular entity X by means of a (discrete) **sample space** Ω_X which is an **exhaustive** and **mutually exclusive** set of states $\{\omega_1, \omega_2, \ldots, \omega_n\}$ that X can be in. When the states of the sample space are exhaustive *and* mutually exclusive, we are sure that X is in exactly one state $\omega \in \Omega_X$. Formulated slightly differently, if we were to inspect X to determine its true state, that state would be described by exactly one $\omega \in \Omega_X$, thus leaving *no ambiguity* about the true state of X. We can then describe the uncertainty about X by a function

$$P_X : \mathcal{P}(\Omega_X) \mapsto [0, 1]$$

mapping from subsets of Ω_X to real numbers on the unit-interval where each subset $E \in \mathcal{P}(\Omega_X)$ is referred to as an **event**. Specifically, we are interested in a certain class of such functions called **probability functions** (or

probability distributions) which are characterized by the following axioms (Kolmogorov, 1950):

Axiom 1 (**Non-negativity**).
For all $E \in \mathcal{P}(\Omega_X)$,

$$P_X(E) \geq 0 \,. \tag{2.1}$$

Axiom 2 (**Normalization**).

$$P_X(\Omega_X) = 1 \,. \tag{2.2}$$

Axiom 3 (**Finite additivity**).
For any sequence of disjoint *events* $E_1, E_2, \ldots, E_n \in \mathcal{P}(\Omega_X)$,

$$P_X\left(\bigcup_{i=1}^{n} E_i\right) = \sum_{i=1}^{n} P_X(E_i) \;. \tag{2.3}$$

The triple $\langle \Omega_X, \mathcal{P}(\Omega_X), P_X \rangle$ is then called a **probability space** for X. From these axioms it readily follows for events E and $E^c = \Omega_X \setminus E$ (the **complement event**) that

$$0 \leq P_X(E) \leq 1, \quad P_X(\emptyset) = 0, \quad P_X(E) = 1 - P_X(E^c) \,. \tag{2.4}$$

Furthermore, if $E_1 \subset E_2$, then

$$P_X(E_1) \leq P_X(E_2) \quad \text{and} \quad P_X(E_2 \setminus E_1) = P_X(E_2) - P_X(E_1) \,. \tag{2.5}$$

Since the **elementary events** $\{\omega\}$ (with $\omega \in \Omega_X$) partition the sample space into mutually exclusive events, then

$$\sum_{\omega \in \Omega_X} P_X(\{\omega\}) = P_X(\Omega_X) = 1 \,. \tag{2.6}$$

If we give a probability 1 to an event, we say the event is **certain**, and if we give a probability of 0 to an event, we say the event is **impossible**.

Before we continue, let us discuss the appropriateness of probability theory. The set of axioms above may appear somewhat *arbitrary*. However, it turns out that they follow from a more intuitive set of rules. Imagine that we are trying to state the rules that we want a reasoning computer to abide to. Then assume we come up with the following rules (Cox, 1979):

1. The degree of belief of a proposition can be represented by a real number.

2. The extremes of this scale must be compatible with logic.

3. An infinitesimal increase in the degree of belief of proposition Y given new evidence implies an infinitesimal decrease in the degree of belief for the proposition "not Y".

4. If conclusions can be reached in more than one way, then every possible way must lead to the same degree of belief.

5. Where possible, all the available evidence should be used in evaluating the degree of belief in a proposition. We may not selectively ignore any prior knowledge.

6. Equivalent propositions should have the same degree of belief. That is, if we have the same state of knowledge about two propositions, except perhaps for the labelling of the propositions, then we must have the same degree of belief in both.

It is quite astonishing to learn that the *only* mathematical theory, which is consistent with the above rules, is probability theory. A good description of this approach can be found in (Cheeseman, 1985), and in particular in (Jaynes, 2003).

To make notation a bit easier we shall reconciliate the entity X and its associated sample space Ω_X into a **variable** (or **chance node**) X with state space $sp(X) = \Omega_X$. When X is a variable, we allow the notation $x \in X$ for $x \in sp(X)$. We also drop the subscript X from P_X and write $P(X)$ for the *table* that determines a probability distribution over variable X. In what follows we let $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ be (non-empty) sets of variables where a probability distribution is defined over the **cartesian product space** of the set of variables. E.g., if $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, then $P(\mathcal{X})$ is a *multi-dimensional table* defined over the space $sp(\mathcal{X}) = sp(X_1) \times sp(X_2) \times \cdots \times sp(X_n)$ that determines a **joint probability distribution** over $\mathcal{X}$. (We often call a probability distribution simply for a distribution.) Again, we allow the notation $\mathbf{x} \in \mathcal{X}$ for $\mathbf{x} \in sp(\mathcal{X})$. We call the *elementary event* $\mathbf{x} \in \mathcal{X}$ for a **configuration** of the variables in $\mathcal{X}$, and $P(\mathbf{x})$ for the (point) **probability** of $\mathbf{x}$, corresponding to the entry in table $P(\mathcal{X})$ indexed by $\mathbf{x}$.

If we have a joint distribution $P(\mathcal{X})$ we may find the joint distribution of any (non-empty) subset of variables $\mathcal{Y} \subset \mathcal{X}$ by means of **maginalization**

$$P(\mathcal{Y}) \quad = \quad \sum_{\mathbf{x} \in \mathcal{X} \setminus \mathcal{Y}} P(\mathcal{Y}, \mathbf{x}) \tag{2.7}$$

where we *sum out* all irrelevant variables. If $\mathcal{X} \setminus \mathcal{Y} = \{X_1, \ldots, X_m\}$, the above notation is shorthand for

$$P(\mathcal{Y}) \quad = \quad \sum_{x_1 \in X_1} \cdots \sum_{x_m \in X_m} P(\mathcal{Y}, x_1, \ldots, x_m) \, . \tag{2.8}$$

Formulas involving probability tables will be meaningless until we define what it means to apply operations like sum, multiplication and equality on them. The general convention is to interpret all such formulas on the level of particular configurations, that is, Equation 2.8 is defined to mean

$$\text{for all } \mathbf{y} \in \mathcal{Y}, \qquad P(\mathbf{y}) = \sum_{\mathbf{x} \in \mathcal{X} \setminus \mathcal{Y}} P(\mathbf{y}, \mathbf{x}) \tag{2.9}$$

and as such the operations we can apply in this "calculus of tables" are exactly the operations of real numbers (with associativity and commutativity). We therefore also allow expressions such as $P(\mathcal{X}) > 0$ which simply means that for all $\mathbf{x} \in \mathcal{X}$, $P(\mathbf{x}) > 0$.

To describe the world *after* certain events have occurred, we define the **conditional probability** of $\mathcal{X}$ *given* $\mathcal{Y}$, $\mathcal{X} \cap \mathcal{Y} = \emptyset$, as

$$P(\mathcal{X} \,|\, \mathcal{Y}) = \frac{P(\mathcal{X}, \mathcal{Y})}{P(\mathcal{Y})} \qquad \text{whenever } P(\mathbf{y}) > 0\,. \tag{2.10}$$

It can then be shown that for any $\mathbf{y} \in \mathcal{Y}$ such that $P(\mathbf{y}) > 0, P(\mathcal{X} \,|\, \mathbf{y})$ also satisfies the three axioms of a probability distribution. We also define $P(\mathcal{X} \,|\, \emptyset) \equiv P(\mathcal{X})$. To distinguish between the world before and after certain events have occurred, it is customary to call $P(\mathbf{x})$ for a **prior probability** (or **unconditional probability**) and $P(\mathbf{x} \,|\, \mathbf{y})$ for a **posterior probability**. Similar terminology applies to probability distributions (or probability tables). Conditional probabilities are (rightfully) seen as a more fundamental concept than unconditional probabilities. This becomes clear when we try to decide what $P(Rain)$ should be, that is, the probability that it will rain tomorrow at 12 a.m. If we are in Denmark, it might be reasonable to take $P(Rain) = 0.2$ whereas in Sahara it would be reasonable to take $P(Rain) = 0.000001$. The example illustrates that we often *implicitly* specify an unconditional probability which is only valid in a certain *context*. Therefore, we can often view the unconditional probability as an *abstraction*, summarizing what we think about a proposition *given* everything else that we know and which we do not want to model explicitly.

From the definition of conditional probability, we immediately get that the joint distribution $P(\mathcal{X}, \mathcal{Y})$ can be found via the **fundamental rule**

$$P(\mathcal{X}, \mathcal{Y}) = P(\mathcal{X} \,|\, \mathcal{Y}) \cdot P(\mathcal{Y}) = P(\mathcal{Y} \,|\, \mathcal{X}) \cdot P(\mathcal{X})\,. \tag{2.11}$$

This generalizes to the so-called **chain rule**; e.g. for $\mathcal{X} = \{X_1, \ldots, X_n\}$ we get

$$P(\mathcal{X}) = P(X_n \,|\, X_1, \ldots, X_{n-1}) \cdots P(X_2 \,|\, X_1) \cdot P(X_1) \tag{2.12}$$

as one of the $n!$ different expansions of $P(\mathcal{X})$. From Equation 2.11 it is then trivial to derive the celebrated **Bayes' rule**

$$P(\mathcal{X}\,|\,\mathcal{Y}) = \frac{P(\mathcal{Y}\,|\,\mathcal{X}) \cdot P(\mathcal{X})}{P(\mathcal{Y})} \qquad \text{whenever } P(\mathbf{y}) > 0 \tag{2.13}$$

which states how one should update one's belief in $\mathcal{X}$ given that one has received evidence $\mathbf{y}$. The factor $P(\mathcal{Y}\,|\,\mathcal{X})$ is also called the **likelihood** $L(\mathcal{X}\,|\,\mathcal{Y})$ of $\mathcal{X}$ *given* $\mathcal{Y}$ because it tells us something about how likely we are to observe $P(\mathcal{X}\,|\,\mathcal{Y})$. Thus, in a somewhat informal manner we may write Bayes' rule as

$$Posterior \;\; = \;\; \alpha \cdot Likelihood \cdot Prior \tag{2.14}$$

where $\alpha = P(\mathbf{y})^{-1}$ is a normalization constant. We can also provide more general versions of the above formulas as they can always be understood to apply in some larger context $\mathcal{Z}$, effectively defining the assumptions that are taken as common knowledge (cf. the $P(Rain)$ example). For example, the **law of total probability**

$$P(\mathcal{X}) \;\; = \;\; \sum_{\mathbf{y}\in\mathcal{Y}} P(\mathcal{X}\,|\,\mathbf{y}) \cdot P(\mathbf{y}) \tag{2.15}$$

states that the belief over $\mathcal{X}$ is the weighted sum over the beliefs in all the distinct ways that $\mathcal{X}$ may be realized. Adding a context $\mathcal{Z}$, the formula reads

$$P(\mathcal{X}\,|\,\mathcal{Z}) \;\; = \;\; \sum_{\mathbf{y}\in\mathcal{Y}} P(\mathcal{X}\,|\,\mathbf{y},\mathcal{Z}) \cdot P(\mathbf{y}\,|\,\mathcal{Z}) \tag{2.16}$$

which in the case of Bayes' rule yields the general version

$$P(\mathcal{X}\,|\,\mathcal{Y},\mathcal{Z}) = \frac{P(\mathcal{Y}\,|\,\mathcal{X},\mathcal{Z}) \cdot P(\mathcal{X}\,|\,\mathcal{Z})}{\sum_{\mathbf{x}\in\mathcal{X}} P(\mathcal{Y}\,|\,\mathbf{x},\mathcal{Z}) \cdot P(\mathbf{x}\,|\,\mathcal{Z})} \;\text{ whenever } P(\mathbf{y}\,|\,\mathbf{z}) > 0\,. \tag{2.17}$$

To illustrate the power and often unintuitive results (to the layman, at least) of Bayes' rule, we give a small example.

Example 1 (Jensen and Nielsen (2007))**.** *Imagine that the police is stopping drivers under the suspicion of being influenced by alcohol. To test drivers they perform a blood test. This scenario can be modelled by a world with two variables: $Test?$ describing the test with states "positive" and "negative", and a variable $Drunk?$ describing the whether the driver is too drunk with states "yes" and "no". The interesting questions for the police is actually the probability*

$$P(Drunk? = yes\,|\,Test? = positive)$$

because they want to make sure they do not raise charges against an innocent driver (or at least minimize the likelihood of it).

Experience tells the police that 20% *of the drivers whom they stop are in fact too drunk. This means* $P(Drunk? = yes) = 0.2$. *The test is like most tests not perfect, and the laboratory tells the police that it has the following properties:*

	Drunk?=yes	*Drunk?=no*
Test?=yes	*0.99*	*0.001*
Test?=no	*0.01*	*0.999*

which corresponds to $P(Test?|Drunk?)$. *The number* 0.01 *is called the* ***false negatives rate*** *of the test, and the number* 0.001 *is called the* ***false positives rate*** *of the test. To apply Bayes' rule we either need* $P(Test?)$ *or to apply normalization. We use Equation 2.15 and find* $P(Test? = positive) \approx 0.1988$. *Then Bayes' rule yields*

$$P(Drunk? = yes \,|\, Test? = positive) \approx \frac{0.99 \cdot 0.2}{0.1988} \approx 0.996 \,.$$

Now imagine that the police changes their strategy such that all drivers are stopped, that is, not just those under suspicion. The police estimates that in this context $P(Drunk? = yes) = 0.001$. *This implies a new value* $P(Test? = positive) \approx 0.00199$, *and Bayes' rule gives*

$$P(Drunk? = yes \,|\, Test? = positive) \approx \frac{0.99 \cdot 0.001}{0.00199} \approx 0.497 \,.$$

So in this scenario, more than half of all people charged with the crime are actually innocent (!). This also illustrates the importance of considering prior probabilities when designing screening for cancer and other rare diseases.

Finally, we shall mention the following measure of distance between probability distributions. Given two probability distributions $P_1(\mathcal{X})$ and $P_2(\mathcal{X})$, we may measure the distance between them by employing the **Kullback-Leibler divergence** (or simply **KL-divergence**):

$$D_{KL}(P_1||P_2) \;=\; \sum_{\mathbf{x}\in\mathcal{X}} P_1(\mathbf{x}) \cdot \lg \frac{P_1(\mathbf{x})}{P_2(\mathbf{x})} \tag{2.18}$$

where we require that $P_1(\mathbf{x}) > 0$ implies $P_2(\mathbf{x}) > 0$ ($0/0$ is thus interpreted as 0). ($\lg x$ denotes the binary logarithm of x.) Notice that KL-divergence is not a symmetric distance function. The KL-divergence is widely used in sensitivity analysis of Bayesian network parameters (see e.g., (Renooij, 2010)).

2.3 Conditional Independence

As we have seen, the joint probability distribution $P(\mathcal{X})$ can tell us any probabilistic relationship between any two subsets of variables $\mathcal{Y}, \mathcal{Z} \subset \mathcal{X}$ (with $\mathcal{Y} \cap \mathcal{Z} = \emptyset$) from which we can make powerful predictions about the world described by $P(\mathcal{X})$. However, representing or specifying $P(\mathcal{X})$ is quite often impossible as the size of $P(\mathcal{X})$ grows *exponentially* in the number of variables in $\mathcal{X}$. Formulas like the chain rule (Equation 2.12) does not remedy this problem as the first factor has the same size as $P(\mathcal{X})$.

In probability theory we use the notion of **independence** and **conditional independence** to capture how probabilistic relationships between variables in $\mathcal{X}$ should change in response to new facts. The following example illustrates one aspect.

Example 2 (The wet grass conundrum)**.** *Seeing clouds in the sky ($Clouds?$) will influence our belief in whether it rains ($Rain?$) which in turn will influence our belief in whether the grass is wet ($Wet?$). Illustrated as a **causal chain** (in which the direction goes from **cause** to **effect**) the scenario looks like*

$$Clouds? \rightarrow Rain? \rightarrow Wet? . \tag{2.19}$$

However, if we first *observe that it actually rains, the* additional *observation that the sky is cloudy does not change our belief in the grass being wet. Conversely, knowing that it rains, then observing that the grass is wet does not change our belief in the sky being cloudy. Therefore we say variables $Clouds?$ and $Wet?$ are **conditionally independent** given $Rain?$.*

The following definition captures independencies in terms of probability distributions and shall be our main mathematical device for simplifying joint distributions.

Definition 1 (Pearl (1988))**.** *Let $\mathcal{U}$ be a finite set of variables, let $P(\mathcal{U})$ be a probability distribution over $\mathcal{U}$, and let $\mathcal{X}, \mathcal{Y}$ and $\mathcal{Z}$ stand for any three subsets of variables in $\mathcal{U}$. $\mathcal{X}$ and $\mathcal{Y}$ are said to be **conditionally independent** given $\mathcal{Z}$ if*

$$P(\mathcal{X} \,|\, \mathcal{Y}, \mathcal{Z}) = P(\mathcal{X} \,|\, \mathcal{Z}) \qquad \textit{whenever } P(\boldsymbol{y}, \boldsymbol{z}) > 0 \,. \tag{2.20}$$

If the set $\mathcal{Z}$ is empty we talk merely about **independence** (or **marginal independence** or **absolute independence**). For example, the chain rule in Equation 2.12 simply becomes $P(\mathcal{X}) = \prod_i^n P(X_i)$ when all variables X_i are pairwise independent. We write these properties succinctly as

$$\begin{aligned} (\mathcal{X} \perp\!\!\!\perp \mathcal{Y} \,|\, \mathcal{Z}) &\sim \text{conditional independence} \\ (\mathcal{X} \perp\!\!\!\perp \mathcal{Y}) &\sim \text{marginal independence} \end{aligned}$$

noting that they should always be understood in the context of a particular distribution $P(\mathcal{U})$.

From the definition of conditional independence we can derive a useful, characterizing (*axiomatic*) set of rules without reference to any concrete numerical representation of $P(\mathcal{U})$. These rules were originally derived in (Dawid, 1979), but given here in the form of (Pearl, 1988).

Theorem 1. *Let $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$, and $\mathcal{W}$ be four disjoint subsets of variables from $\mathcal{U}$. Then the conditional independence relation $(\mathcal{X} \perp\!\!\!\perp \mathcal{Y} | \mathcal{Z})$ with reference to a probabilistic model $P(\mathcal{U})$ satisfies*

Symmetry:	$(\mathcal{X} \perp\!\!\!\perp \mathcal{Y} \mid \mathcal{Z}) \Longrightarrow (\mathcal{Y} \perp\!\!\!\perp \mathcal{X} \mid \mathcal{Z})$.
Decomposition:	$(\mathcal{X} \perp\!\!\!\perp \mathcal{Y}, \mathcal{W} \mid \mathcal{Z}) \Longrightarrow (\mathcal{X} \perp\!\!\!\perp \mathcal{Y} \mid \mathcal{Z})$.
Weak union:	$(\mathcal{X} \perp\!\!\!\perp \mathcal{Y}, \mathcal{W} \mid \mathcal{Z}) \Longrightarrow (\mathcal{X} \perp\!\!\!\perp \mathcal{Y} \mid \mathcal{Z}, \mathcal{W})$.
Contraction:	$(\mathcal{X} \perp\!\!\!\perp \mathcal{Y} \mid \mathcal{Z})$ and $(\mathcal{X} \perp\!\!\!\perp \mathcal{W} \mid \mathcal{Z}, \mathcal{Y}) \Longrightarrow (\mathcal{X} \perp\!\!\!\perp \mathcal{Y}, \mathcal{W} \mid \mathcal{Z})$.
Intersection:	If $P(\mathcal{U})$ is strictly positive $(\mathcal{X} \perp\!\!\!\perp \mathcal{Y} \mid \mathcal{Z}, \mathcal{W})$ and $(\mathcal{X} \perp\!\!\!\perp \mathcal{W} \mid \mathcal{Z}, \mathcal{Y}) \Longrightarrow (\mathcal{X} \perp\!\!\!\perp \mathcal{Y}, \mathcal{W} \mid \mathcal{Z})$.

These rules are also known as the **graphoid axioms** because they have been shown to capture the notion of *informational relevance* in a wide variety of interpretations (Pearl, 2009). The intuition behind these axioms are as follows:

- Symmetry: given that $\mathcal{Z}$ is known, if $\mathcal{Y}$ is irrelevant to $\mathcal{X}$, then $\mathcal{X}$ is irrelevant $\mathcal{Y}$ (cf. Example 2).
- Decomposition: if two combined pieces of information are considered irrelevant to $\mathcal{X}$, then each separate item is irrelevant as well.
- Weak union: by learning the irrelevant information $\mathcal{W}$ we cannot make $\mathcal{Y}$ relevant to $\mathcal{X}$.
- Contraction: if (a) we learn about irrelevant information $\mathcal{Y}$ to $\mathcal{X}$, and (b) $\mathcal{W}$ is irrelevant to $\mathcal{X}$ in this new context, then $\mathcal{W}$ was also irrelevant to $\mathcal{X}$ before we observed $\mathcal{Y}$.
- Intersection: if (a) $\mathcal{Y}$ is irrelevant to $\mathcal{X}$ when we know $\mathcal{W}$, and (b) $\mathcal{W}$ is irrelevant to $\mathcal{X}$ when we know $\mathcal{Y}$, then neither $\mathcal{Y}$ nor $\mathcal{W}$ (nor their combination) is relevant to $\mathcal{X}$.

In the next section we shall describe a graphical criterion that mirrors these rules for directed acyclic graphs. Since the rest of this thesis contains many references to graphs, we shall briefly review some terminology.

A **graph** G is a pair (V, E) where V is the set of **vertices** (or **nodes**) and E is a set of **edges** (or **links**). Unless explicitly mentioned, we deal exclusively with directed graphs where an **edge** (or **link**) $(v, u) \in E$ starts at the vertex v and ends at the vertex u. Similarly, we always assume that the graphs are connected. A **path** in a graph is a sequence of nodes $\langle v_1, v_2, \ldots, v_k \rangle$ where each adjacent pair of vertices corresponds to an edge in the graph. A

node v is **reachable** from a node u if there is a path from u to v. For a node $v \in V$, $pa(v)$ is the set of **parents** (or the set of **predecessors** $pred(v)$), $ch(v)$ is the set of **children** (or **successors** $succ(v)$), $anc(v)$ is the set of **ancestors** consisting of all nodes from which v is reachable, and $desc(v)$ is the set of **descendants** consisting of all nodes reachable from v. A graph has a **cycle** if there is a path starting and ending in the same node, and a graph without cycles is called **acyclic** If one node v has no parents, and all other nodes have exactly one parent, the graph is called a (directed) **tree** with **root** v, and a path between two nodes v and u (u reachable from v), denoted $path(v, u)$, is unique. The set of nodes with no children in a graph G is called the **leaves** of G, denoted $\mathcal{L}(G)$. We may also talk about the leaves reachable from a node v, denoted $\mathcal{L}(v)$. An edge or node may have a **label** (or other information) associated with it; for example, the label of a node v is denoted $label(v)$. For trees, we shall also denote the labels of all edges between ancestors of a node v by $past(v)$ or sometimes simply as $\boldsymbol{\epsilon}^v$ because the set of labels are interpreted as a set of evidence. We may identify a particular successor node by the label on the edge leading to that node, e.g., $succ(Q = q)$ identifies the successor found by following the edge labelled with "variable Q being in state q". Often, we do not distinguish between a node v and the entity we have attached to the node; for example, we may describe a graph over variables in $\mathcal{X}$ thereby letting $X \in \mathcal{X}$ correspond both to the variable and to the node X in the graph (depending on the context).

2.4 Bayesian Networks

Even though conditional independence allows us to simplify joint distributions $P(\mathcal{X})$ over a domain $\mathcal{X}$, tables of numbers are inherently unsuited for humans as well as computers. To counter this problem, Bayesian networks (Pearl, 1985) combine a directed acyclic graph (or DAG), stating the conditional independencies between variables, with local probability distributions attached to the nodes of the graph. In total there are three main reasons why Bayesian networks are useful:

1. They provide a *human-friendly* way of expressing assumptions about conditional independencies and cause-effect relationships between variables in the domain $\mathcal{X}$.

2. They provide an *efficient representation* of joint probability distributions by representing $P(\mathcal{X})$ in a factorized manner that avoids storing $P(\mathcal{X})$ explicitly.

3. They provide a basis from which we can *efficiently calculate* any marginal or conditional distribution attainable from $P(\mathcal{X})$.

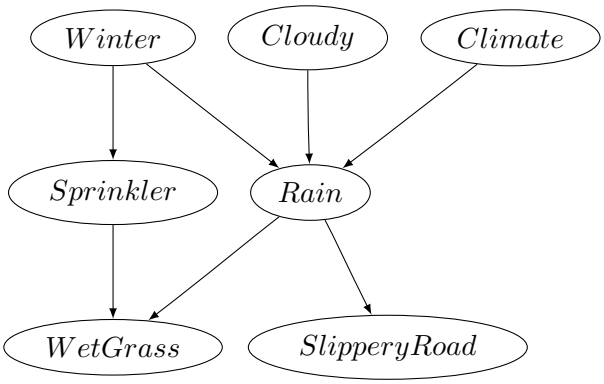


Figure 2.1: Bayesian network structure for Example 3. The structure shows the cause-effect relationship between the six variables.

Before we discuss how these properties are realized, we shall first give a formal definition of the networks. Although Bayesian networks predates (Pearl, 1988), we choose a mathematically simpler definition inspired by the one given in Darwiche (2009).

Definition 2. *Let $\mathcal{X}$ be a set of variables describing some domain. Then a* ***Bayesian network*** *BN is a pair (G, Θ) where*

- *G is a directed acyclic graph over the variables in $\mathcal{X}$ such that each variable corresponds to a node in G, called the* ***network structure****.*
- *Θ is a set of conditional probability tables of the form $P(X \mid pa(X))$, one for each variable X in $\mathcal{X}$, called the* ***network parametrization****.*

The following example extends Example 2.

Example 3 (More on wet grass)**.** *Consider the Bayesian network structure in Figure 2.1 (which is a slight extension of a network from Darwiche (2009)). The variables should be self-explanatory. For example, the variable $Winter$ would be binary with states "yes" and "no", whereas $Climate$ might have states "tropical, desert, temperate" etc. To complete the specification of the Bayesian network, we need to specify the following seven (conditional) probability tables:*

$$P(Winter) \qquad P(Cloudy)$$
$$P(Climate) \qquad P(Sprinkler \mid Winter)$$
$$P(Rain \mid Winter, Cloudy, Climate) \qquad P(WetGrass \mid Sprinkler, Rain)$$
$$P(SlipperyRoad \mid Rain)$$

We can see it is quite straightforward to identify the structure of the required tables when inspecting the graph.

The above example does not specify the particular numbers for the seven tables. These numbers can be provided by experts in the domain or computed from statistical data (or a combination thereof). If we have no information about a particular variable, we may assign a uniform probability table to it. As long as we are dealing with *discrete* variables, this is a sound procedure for stating ignorance.

From the network structure in Figure 2.1 we can directly see how the parents of a variable X are always relevant for X. In other words, X is usually *not* independent from $pa(X)$, that is, $P(X) \neq P(X \,|\, pa(X))$. However, we are also interested in knowing which variables that are irrelevant to X, because this can simplify inference in the network as well as facilitate a discussion about the correctness or appropriateness of the structure of the network. To this end we introduce the following (now famous) graphical criterion.

Definition 3 (Pearl (1988))**.** *If $\mathcal{X} \neq \emptyset, \mathcal{Y} \neq \emptyset$, and $\mathcal{Z}$ are three disjoint subsets of nodes in a DAG G, then $\mathcal{Z}$ is said to **d-separate** $\mathcal{X}$ from $\mathcal{Y}$, denoted $\mathcal{X} \perp \mathcal{Y} \,|\, \mathcal{Z}$, if along every path between a node in $\mathcal{X}$ and a node in $\mathcal{Y}$ there is a node w satisfying one of the following two conditions:*

(a) w has a converging connection and none of w or its descendants are in $\mathcal{Z}$, or

(b) w does not have a converging connection and w is in $\mathcal{Z}$.

Note that we also say that $\mathcal{X}$ and $\mathcal{Y}$ are d-separated *given* $\mathcal{Z}$ which makes the notation $\mathcal{X} \perp \mathcal{Y} \,|\, \mathcal{Z}$ more intuitive. If $\mathcal{X}$ and $\mathcal{Y}$ are not d-separated given $\mathcal{Z}$, we say that they are **d-connected** and write this $\mathcal{X} \not\perp \mathcal{Y} \,|\, \mathcal{Z}$. Note that a node w is always d-connected to all nodes in $pa(w) \cup ch(w)$.

Example 4 (D-separation)**.** *If we consider the DAG in Figure 2.1, we can infer the following d-separation properties (among others). These illustrate the three types of connections that we may encounter:*

Serial connection	$Winter \not\perp WetGrass \,\vert\, \emptyset$	*(case (b))*
Diverging connection	$WetGrass \not\perp SlipperyRoad \,\vert\, \emptyset$	*(case (b))*
Converging connection	$Winter \perp Climate \,\vert\, \emptyset$	*(case (a))*

For non-empty $\mathcal{Z}$ we get

Serial connection	$Winter \perp WetGrass \,\vert\, Sprinkler, Rain$	*(case (b))*
Diverging connection	$WetGrass \perp SlipperyRoad \,\vert\, Rain$	*(case (b))*
Converging connection	$Winter \not\perp Climate \,\vert\, Rain$	*(case (a))*

We can see that serial and diverging connections implies d-connection until there is a set of variables $\mathcal{Z}$ that blocks the path whereas, conversely, the converging connections d-separate until we condition on a set of variables $\mathcal{Z}$. This latter property

is closely connected to the so-called ***explaining away effect****: if we know nothing of* $Rain$*, then learning about* $Winter$ *cannot change our belief in* $Climate$ *(and vice versa)—however, if we know* $Rain = "no"$*, then learning that* $Winter = "no"$ *will make us more confident that* $Climate = "dessert"$ *(and vice versa). The explaining away effect is discussed in detail in (Wellman and Henrion, 1994).*

We are now ready to discuss how a Bayesian network simplifies the joint distribution $P(\mathcal{X})$.

Definition 4. *Let* BN *be a Bayesian network over a set of variables* $\mathcal{X}$*. Then the joint distribution*

$$P(\mathcal{X}) \quad = \quad \prod_{X \in \mathcal{X}} P(X \,|\, pa(X)) \tag{2.21}$$

is called the ***distribution induced*** *by* BN*.*

One can prove that Equation 2.21 defines a unique probability destribution, and so the definition is sound. Equation 2.21 is also called the **chain rule for Bayesian networks** as it is similar to the general chain rule (Equation 2.12) taking into account conditional independencies implied by the Bayesian network structure.

We have now discussed Bayesian networks, their induced distributions and the d-separation properties induced by their network structure. In another context we have discussed conditional independencies and how they relate to joint probability distributions. The following theorem provides a remarkable relationship between these two contexts.

Theorem 2 (Jensen and Nielsen (2007))**.** *Let* $BN = (G, \Theta)$ *be a Bayesian network over a set of variables* $\mathcal{U}$*, and let* $P(\mathcal{U})$ *be the distribution induced by* BN*. Then any d-separation property* $\mathcal{X} \perp \mathcal{Y} | \mathcal{Z}$ *in* G *implies* $(\mathcal{X} \perp\!\!\!\perp \mathcal{Y} | \mathcal{Z})$ *in* $P(\mathcal{U})$*.*

The reverse statement is not true in general and so $P(\mathcal{U})$ may contain conditional independencies that cannot be derived from the DAG structure. For example, take any network structure (with at least 3 variables) and assign a uniform conditional probability table to all nodes; then every pair of variables will be conditionally independent given a third. Nevertheless, the theorem implies that we safely can use d-separation to derive independence statements about the induced distribution $P(\mathcal{U})$. We can also see how Definition 4 was formed: apply the normal chain rule while choosing the variables on the left of $|$ in reverse topological ordering; then the parents of a variable X d-separates X from all other variables on the right of $|$.

We have at times said that an edge in the network structure implied a cause-effect relationship between parent and child. If this is true for all edges, it is common to call the Bayesian network for a **causal network**.

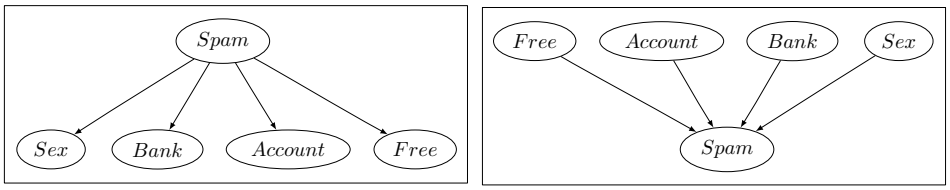


Figure 2.2: Two Bayesian network structures for spam filtering (Example 5). All variables are binary.

However, it is *not* a requirement that edges imply causality for the Bayesian network to be a valid representation of some domain: as long as the d-separation properties of the network are satisfied in the modelled world, we have a sound model.

Principle 1 (**D-separation**)**.** *Use d-separation to verify that a Bayesian network is a correct model of a domain.*

Remark 1. *On the other hand, if one does have reasonable causal assumptions, this enables a whole new paradigm for causal reasoning (Pearl, 2009); for example, one may answer counterfactual queries like "what is the probability that Joe would have died had he not been treated?" (Shpitser and Pearl, 2007).*

Experience also suggests that cause-effect relationships are much simpler to comprehend for humans, and this is especially true when trying to specify the parameters of the network. Most people can probably come up with a reasonable estimate of $P(\mathit{WetGrass} \mid \mathit{Rain}, \mathit{Sprinkler})$ whereas specifying $P(\mathit{Rain} \mid \mathit{WetGrass})$ and $P(\mathit{Sprinkler} \mid \mathit{WetGrass})$ is conceptually very challenging; its also misleading because we cannot specify any combined effect of two common causes. We summarize this observation as follows.

Principle 2 (**Causality**)**.** *Whenever possible, specify Bayesian networks such that they correspond to causal assumptions.*

The following example illustrates some aspects of the two principles above.

Example 5 (Spam filtering)**.** *Consider the two structures in Figure 2.2. Here the idea is that we want to build a spam filter. The variables* Sex, Bank, $\mathit{Account}$ *and* Free *are binary and describe whether the word appears in a particular mail. We then classify a mail by calculating the probability* $P(\mathit{Spam}|\boldsymbol{\epsilon})$ *where* $\boldsymbol{\epsilon}$ *is all the evidence on the four other variables in the network. The key question is: which of the two models is the most correct one? If we apply the causality principle, then we are in an ambivalent situation: it is difficult to say if e.g. the word "sex" is an effect of a mail being "spam" or if the word "sex" causes a mail to be spam. And viewed in isolation, we have no problems specifying either of* $P(\mathit{Spam}|\mathit{Sex})$ *and* $P(\mathit{Sex}|\mathit{Spam})$.

On the other hand, if we apply the d-separation principle, things become more clear. In the model on the right, if we know $Spam =$ "yes" then we expect learning $Free =$ "yes" should increase the likelihood of the three other attributes being present. So, e.g., in this model $Free \not\perp Sex|Spam$ whereas $Free \perp Sex|Spam$ in the model on the left. Hence we deem the model on the right more correct as it is consistent with our d-separation assumptions.

We shall conclude this section by giving a small overview of **inference** (or **propagation**) in Bayesian networks. The three most common inference tasks in a Bayesian network over $\mathcal{U} = \{X_1, \ldots, X_n\}$ are the following:

1. **Belief updating**: calculate $P(X|\mathbf{y})$ for some (or all) $X \in \mathcal{U}$ and some evidence $\mathbf{y} \in \mathcal{Y} \subset \mathcal{U}$ ($\mathcal{Y}$ possibly empty) where $\mathcal{Y} \cap X = \emptyset$. This task has been shown to be NP-hard (Cooper, 1990), even for approximating algorithms (Dagum and Luby, 1993). However, the complexity remains polynomial in the number of nodes when the network structure is a **polytree** (where each node has at most one parent, but may have an arbitrary number of children) (Kim and Pearl, 1983).

2. **The maximum a posteriori assignment**: calculate

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in \mathcal{X}} \sum_{\mathbf{z} \in \mathcal{Z}} P(\mathbf{x}, \mathbf{z}, \mathbf{y}) \tag{2.22}$$

   where $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z} = \mathcal{U}$ are three disjoint sets, and $\mathbf{y} \in \mathcal{Y}$ is the received evidence. We also call this the **MAP assignment** for $\mathcal{X}$ given $\mathbf{y}$. Not surprisingly, this is also an NP-hard task for both exact (Shimony, 1994) and approximate solutions (Abdelbar and Hedetniemi, 1998).

3. **The most-probable explanation**: a special case of the MAP assignment problem where $\mathcal{Z} = \emptyset$.

A recent alternative to MAP assignments is called **most relevant explanation** and appears to have several advantages, most notably that the best explanation can be a partial instantiation of the target variables $\mathcal{X}$ (Yuan et al., 2009). Specialized algorithms exist for all of the above scenarios, and a good overview can be found in (Cowell et al., 1999), (Kjærulff and Madsen, 2008) and (Darwiche, 2009).

For general belief updating, a typical approach is to compile the Bayesian network $BN = (G, \Theta)$ into a special optimized data structure called a **junction-tree** (Jensen and Nielsen, 2007). The procedure consists of three main steps:

(a) **Moralization**: the network structure is **moralized** by connection all nodes that share one or more children. Then all directions on the edges are removed. The resulting undirected graph $G_M = (V, E)$ is called the **moral graph**.

(b) **Triangulation**: the moral graph is **triangulated**, that is, we add a set of edges T such that the graph $G_T = (V, E \cup T)$ has a chord on all cycles of length four or more (thereby connecting two non-adjacent nodes on the cycle).

(c) **Tree construction**: G_T is known to contain at most $|V|$ cliques (maximal subsets of completely connected nodes) which are then arranged into a tree fulfilling the **junction-tree property**: for any pair of cliques (C, K) the set $C \cap K$ appear in every clique on the path between them.

The junction-tree can now be used as a quite efficient basis for a variety of inference tasks. Finding the best possible triangulation in step (b) is actually also intractable (Yannakakis, 1981; Wen, 1990), but we note that once a sufficiently good triangulation has been found, we can save it and be certain that belief updating is tractable hereafter. In this aspect, belief updating is quite special compared to other NP-hard problems: before we perform inference, we already know if that is possible.

2.5 Summary

In this chapter we have reviewed basic discrete probability theory to be able to reason consistently about uncertain worlds. We described how probabilistic relationships naturally emerges either as an abstraction mechanism or as inherent properties of the domain which we are trying to model formally. Most prominently, Bayes' rule allowed us to reason backwards and update probability distributions in light of new evidence. Via marginalization on a joint distribution we could compute conditional or marginal distributions of interest, and Bayesian networks provided a general, compact and efficient basis for such calculations by exploiting conditional independencies. We discussed causality as a simplifying principle and briefly reviewed a number of common inference tasks for Bayesian networks. In summary, probability theory and Bayesian networks provide a flexible and sound theoretical foundation for troubleshooting models.

Chapter 3

Decision Theory and Troubleshooting

> It is a mistake to look too far ahead. Only one link of the chain of destiny can be handled at a time.
> –*Winston Churchill*

Given a troubleshooting model that describes the uncertainty and cost of actions and questions, the goal is to compute a strategy that continuously tells us what to do next given the previous steps. To formalize this task we apply modern decision theory which can be seen as the fusion of probabilistic reasoning with utilities. Therefore we briefly investigate utility theory and the principle of maximum expected utility. Then we introduce decision trees as a general graphical representation of any decision scenario. This enables us to give a formal definition of decision theoretic troubleshooting, which we compare to alternative modelling methods.

3.1 Utility Theory

In decision theory we want to create models that are helpful in making decisions. In principle, there are two types of decisions:

(a) **test decisions** (or **observations**) which provide evidence on a variable of interest, and

(b) **action decisions** (or **actions**) which affect the current state of the world when performed.

In the framework of Bayesian networks we can already model observations and measure the effect of new evidence in terms of posterior distributions over variables of interest. We may also model the actions as variables in a Bayesian network where choosing an action amounts to fixing the state of the variable unambiguously (**instantiation**). This will in general lead to a new posterior distribution over the remaining uninstantiated variables.

However, we are still not able to describe the *usefulness* of the consequences of making the decision. To this aim we introduce **utilities** which can be seen as real-valued functions of the decision as well as the state of the world. We assume that our notion of usefulness can be measured on a numerical scale called a **utility scale**, and if several types of utility appear in a decision problem, we assume that the scales have a common unit.

Assume we have a utility function $U : sp(\mathcal{X}) \times sp(D) \mapsto \mathbb{R}$ describing the joint desirability of the variables in $\mathcal{X}$ which are judged to be influenced by a **decision variable** D with mutually exclusive states $\{d_1, \ldots, d_k\}$ each describing a particular decision (e.g., to repair a component or perform a test). $\mathcal{X}$ is often called the **consequence set** of D. Then the **expected utility** of decision $d \in D$ given evidence $\mathbf{y} \in \mathcal{Y}$ (where $\mathcal{Y} \cap \mathcal{X} = \emptyset$) is found by

$$EU(d|\mathbf{y}) \quad = \quad \sum_{\mathbf{x} \in \mathcal{X}} U(\mathbf{x}, d) \cdot P(\mathbf{x} | d, \mathbf{y}) \tag{3.1}$$

which is simply the average utility value of the consequences, that is, the utility of the consequences weighted by the probability of the consequence occurring. The principle of **maximum expected utility** then states that the best (most rational) decision is given by

$$d^* \quad = \quad \arg\max_{d \in D} EU(d|\mathbf{y}) \; . \tag{3.2}$$

The principle can be carefully justified by axioms about preferences of decision outcomes (Morgenstern and Von-Neumann, 1947; Pearl, 1988).

Principle 3 (**Instrumental Rationality**)**.** *If a decision maker is to act rationally, that is, consistently with his preferences, then his only option is to choose decisions that maximize the expected utility.*

In general, there is no unique utility function $U(\mathcal{X}, D)$ for $(\mathcal{X}, D)$. Specifically, if $U(\mathcal{X}, D)$ is a utility function, then

$$V(\mathcal{X}, D) = \alpha \cdot U(\mathcal{X}, D) + \beta \qquad \text{with } \alpha, \beta \in \mathbb{R} \tag{3.3}$$

also a valid utility function.

Example 6 (Humans and rationality)**.** *Under the second world war, Churchill was worried that the French Mediterranean fleet would fall into German hands. He ordered an attack on the French fleet off the coast of French Algeria on the 3rd of July 1940, sinking the fleet and killing 1,297 frenchmen. This can be seen as a rational decision if he believed that the expected casualties would have been higher if the French fleet fell into German hands.*

But humans often fail to act in accordance with the principle of maximum expected utility, and still we should be careful not to label their actions as irrational. In game shows like "Who want's to be a millionaire?", the contestant often chooses not to continue in the game even though that decision would have a higher expected utility. However, since the contestant cannot be part in the show many times, he wisely chooses to stop when the loss implied by a wrong answer becomes too high.

3.2 Decision Trees

In the following we shall politely ignore Churchill's advice about never looking more than one step ahead. In particular, we are concerned with **sequential decision problems** where we must make a (finite) sequence of decisions, and where the utility of decisions that we are about to make is potentially influenced by *all* future decisions and observations.

To visualize and reason about such sequential decision scenarios, we may use a **decision tree** in which square nodes represent decision variables, round nodes represents chance variables and diamond nodes represent utilities. A path from the root node to a leaf node represents one possible **realization** of the decision problem; for example, in a decision tree for a car start problem we may have a path

$$\begin{aligned} \textit{Clean the sparc plugs} &\longrightarrow \textit{Is the problem gone?} \\ &\overset{no}{\longrightarrow} \textit{Put gas on the car} \\ &\longrightarrow \textit{Is the problem gone?} \\ &\overset{yes}{\longrightarrow} -100\$ \end{aligned}$$

where the last information is the utility of preceeding observations and actions. In general, a decision tree describes all possible sequences of observations and actions valid under a certain set of assumptions.

Definition 5 (Decision Tree)**.** *A **decision tree** is a labelled directed tree $\mathcal{T}$ over a set of decision nodes $\mathcal{D}$, a set of chance nodes $\mathcal{X}$, and a set of utility nodes $\mathcal{V}$ with the following structural constraints:*

(i) Every leaf node is a utility node and every non-leaf node is a decision or chance node.

(ii) Every outgoing edge of a decision node D is labelled with a unique state $d \in D$.

(iii) Every outgoing edge of a chance node X is labelled with a unique state $x \in X$, and there is a successor node for each possible (currently meaningful) $x \in X$.

To each utility node $V \in \mathcal{V}$ we attach a utility $U(past(V))$, describing the utility of ending the decision making after $past(V)$ has been observed. Furthermore, to every outgoing edge of a chance node $X \in \mathcal{X}$ we attach the probability $P(x|past(X))$ describing the probability of observing $X = x$ after observing $past(X)$. Finally, for all decision nodes $D \in \mathcal{D}$ we define $P(d|past(D)) \equiv 1$ for an outgoing edge labelled with decision $d \in D$

To get the conditional probabilities $P(X|past(X))$ we may use a Bayesian network. A decision tree also incorporates the **no-forgetting assumption**

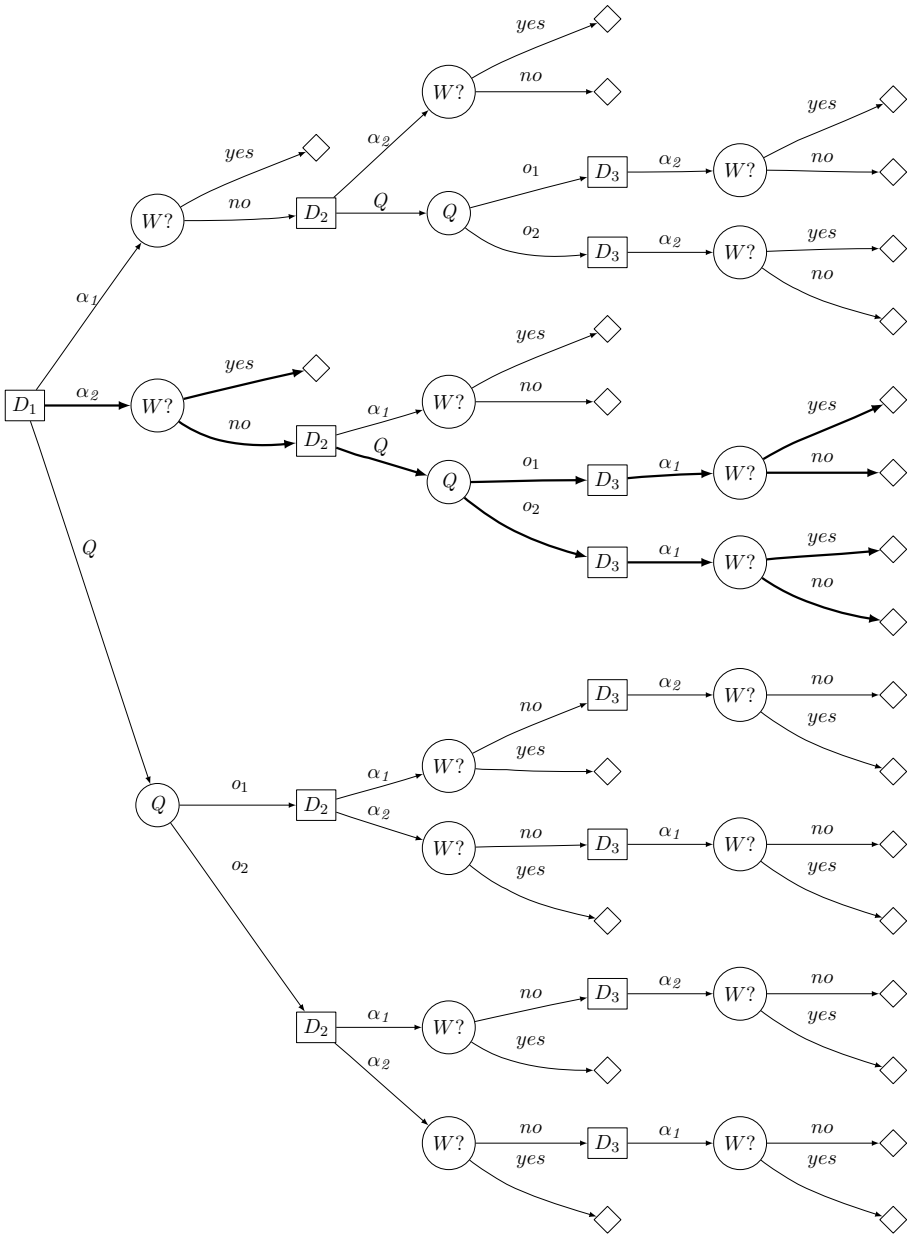


Figure 3.1: A decision tree for a simple repair scenario with two actions α_1 and α_2 (which may fix the problem) and two observations Q (providing background information) and $W?$(describing if the system is working). Utilities and probabilities have been left unspecified. The subtree induced by the bold edges is an example of a strategy.

which states that when decision D is taken, the whole past $past(D)$ is known to the decision maker. Note that the root ρ of the decision tree may be a utility node, in which case the tree contains only the root node. The following example illustrates some aspects of decision trees.

Example 7 (Asymmetrical decision tree)**.** *Consider the decision tree in Figure 3.1 which describes a simple troubleshooting scenario with two actions and two observations. At the root D_1 we have three decisions at our disposal: we can perform an action α_1 (e.g., clean the spark plugs) or an action α_2 (e.g., put gas on the car), or we may make an observation Q (e.g., what is the fuel meter standing? (empty,non-empty)). Every decision tree is build on some assumptions, and in this case we see that the variable W? (e.g., is the car working?) is always a successor of the two actions α_1 and α_2. Another assumption is that we have no further decisions after both of α_1 and α_2 have been performed. A third assumption is that we can only make a decision once which lead to a somewhat* ***asymmetrical*** *decision tree.*

Given a decision problem and some associated decision tree, we describe one possible way to solve the decision problem via a strategy:

Definition 6 (Strategy)**.** *Let $\mathcal{T}$ be a decision tree for some decision problem. Then a* ***strategy*** *in $\mathcal{T}$ is a subtree of $\mathcal{T}$ with the constraint that each decision node has exactly one successor node. If the strategy starts with the root node of $\mathcal{T}$, we call it a* ***full strategy****; otherwise it is a* ***partial strategy****.*

In Figure 3.1 one strategy is highlighted with bold edges. A particular (full) strategy therefore *prescribes* how to act at any given time in the decision process. Since a decision tree contains many strategies, we want to measure how good they are, so we can pick the best one. The following definitions provide such a measure.

Definition 7 (Expected Utility of Strategies)**.** *Let n be a node in a strategy s. The* ***expected utility*** *of n is given recursively by*

$$EU(n) \quad = \quad \begin{cases} \displaystyle\sum_{m \in succ(n)} P(m \,|\, past(m)) \cdot EU(m) & \text{if } n \text{ is a chance node} \\ EU(m) & \text{if } n \text{ is a decision node with successor } m \\ U(past(n)) & \text{if } n \text{ is a utility node} \end{cases}$$

The ***expected utility*** *of s, $EU(s)$, is given by the expected utility of its root node.*

Definition 8 (Optimal Strategy)**.** *Let $\mathcal{S}(\mathcal{T})$ be the set of all strategies for a decision tree $\mathcal{T}$. Then an* ***optimal strategy*** *in $\mathcal{T}$ is given by*

$$s^* \quad = \quad \arg\max_{s \in \mathcal{S}(\mathcal{T})} EU(s) \tag{3.4}$$

Algorithm 1 Computing the optimal strategy recursively. The function returns the expected utility of the optimal strategy while (a) computing the expected utility for all internal nodes in the decision tree, and (b) marking the best decision for each decision node in the decision tree. In general we use '&' to denote that arguments are passed as references.

function MARKOPTIMALSTRATEGY(&$\mathcal{T}$)
  Input: *A decision tree $\mathcal{T}$ with root ρ*
  if ρ is a chance node **then**
    Set $EU(\rho) = \sum_{m \in succ(\rho)} P(m \,|\, past(m)) \cdot$ MARKOPTIMALSTRATEGY(m)
    return $EU(\rho)$
  else if ρ is a decision node **then**
    Let $m^* = \arg\max_{m \in succ(\rho)}$ MARKOPTIMALSTRATEGY(m)
    Set $EU(\rho) = EU(m^*)$
    Mark the edge from ρ to m^* as optimal
    return $EU(\rho)$
  else if ρ is a utility node **then**
    return $U(past(\rho))$
  end if
end function

These definitions lead directly to a simple recursive procedure for calculating the optimal strategy of a decision tree by choosing the decision option with highest expected utility (see Algorithm 1). If the procedure works in a bottom-up fashion, that is, starting at the leaves and working its way towards the root, it is known as the **average-out and fold-back algorithm** (Jensen and Nielsen, 2007). The algorithm assumes a complete decision tree is given and this has several computational drawbacks—we shall return to search procedures over decision trees in Chapter 4.

Decision trees very quickly become intractably large as they grow (potentially) super-exponentially in the number of decision nodes. To reduce the size of the tree such that it is only of size exponential in the number of decision nodes, we may employ **coalescing** of decision nodes. The key observation is that when two decision nodes n and m have $past(n) = past(m)$ the future (remaining) decision problem is the same for n and m and hence $EU(n) = EU(m)$. In Figure 3.2 is depicted the **coalesced decision tree** for the decision problem in Figure 3.1 on page 28. We can see how decision nodes have been merged to avoid duplicating identical remaining subtrees. Strictly speaking, the graph is no longer a tree, but as it represents a decision tree (compactly), the name seems justified.

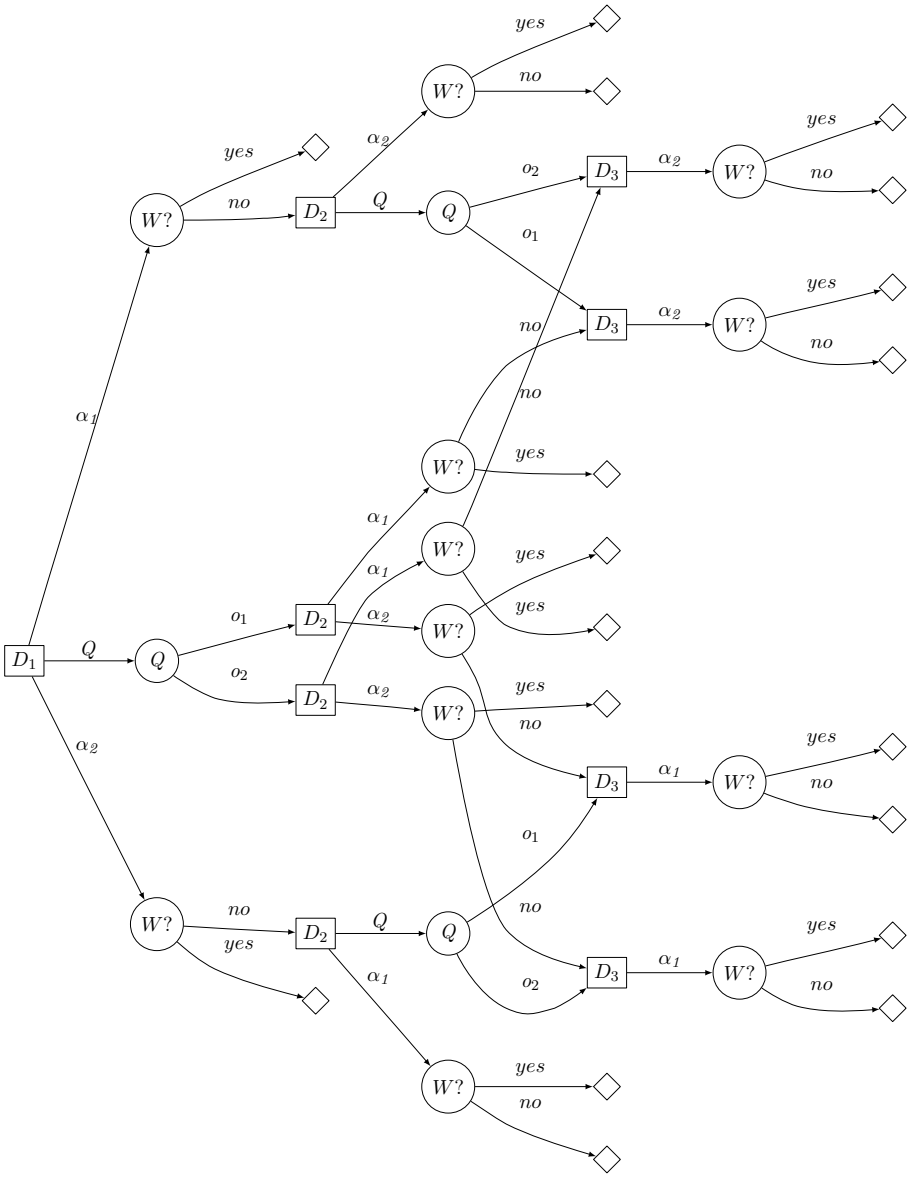


Figure 3.2: Coalesced decision tree for the decision tree in Figure 3.1. All pairs of decision nodes (n, m) with $past(m) = past(n)$ in Figure 3.1 have the same future and hence $EU(m) = EU(n)$.

Albeit decision trees quickly become too large to be useable as a general specification language, they serve as a useful framework for (a) defining the precise semantics of more compact representations of decision problems, and (b) solving such decision problems by search procedures working on the full or partial decision tree. In particular, decision trees provide the most general framework for decision problems where we can model all possible asymmetries explicitly. We shall end our discussion by considering these forms of asymmetry. We first characterize a symmetric decision problem as follows.

Definition 9 (Jensen and Nielsen (2007))**.** *A decision problem is said to be* ***symmetric*** *if*

(a) in all of its decision tree representations, the number of strategies is the same as the cardinality of the cartesian product space over all chance and decision variables, and

(b) in at least one decision tree representation, the sequence of chance and decision variables is the same in all strategies.

At the other end of the spectrum we might have an asymmetrical decision problem. This asymmetry may be of three types:

1. **Functional asymmetry**: when the possible outcomes of a variable depends on the past. For example, we may only have the decision of building one type of oil rig if a preliminary test drill has actually found a specific kind of oil in the oil well.

2. **Structural asymmetry**: when the actual occurrence of a variable depends on the past. This can be seen as a extreme case of functional asymmetry where all options are invalid. As an example, we might only have the decision of building an oil rig if a preliminary test discovered oil in the oil well.

3. **Order asymmetry**: when the ordering of observations and decision are not specified by the decision model. For example, when repairing a car, it is unknown what the best order of repair actions should be.

Returning to Figure 3.1 on page 28, we can see that this decision problem is highly asymmetrical; in fact, the tree embodies all three asymmetrical aspects above: (i) it has functional asymmetry because the number of decision options decrease as we move down the tree, (ii) it has structural asymmetry because there is no further decision options when decision α_1 and α_2 have been made, and (iii) it is order asymmetrical as there is no order constraint on the three decisions α_1, α_2 and Q (other than Q cannot come after the other two).

3.3 Decision Theoretic Troubleshooting

Decision theoretic troubleshooting is motivated by the need to model the repair of a complicated human-made device so we can minimize the expected cost of repairing the device. There are other areas which overlap considerably with troubleshooting, e.g., adaptive testing (Vomlel, 2004), real-time cost-efficient probing (Rish et al., 2005), or value of information analysis with misclassification costs (Bilgic and Getoor, 2007), but we shall stick with the "repair" metaphor and base our notation and terminology on it. We therefore use the following terminology and assumptions. However, we shall first summarize the required ingredients for any decision theoretic troubleshooting model.

Definition 10 (Required Elements of A Troubleshooting Model)**.** *Any **decision theoretic troubleshooting model** is a decision model based on the principle of maximum expected utility with the following elements:*

1. *A set of **faults** $\mathcal{F} \neq \emptyset$ describing the potential causes of the problem of the device.*

2. *A set of **actions** $\mathcal{A} \neq \emptyset$ which may be performed to fix the device.*

3. *A set of **questions** $\mathcal{Q}$ (potentially empty) which may provide background information on the potential faults in $\mathcal{F}$.*

4. *Some form of **probabilistic model** $P(\mathcal{F}, \mathcal{A}, \mathcal{Q})$ over faults, actions and questions where elements in $\mathcal{F}$ (however modelled) need not be directly observed. In this probabilistic model we can observe evidence $\boldsymbol{\epsilon} \in \mathcal{E}$ where $\mathcal{E}$ is the **set of all possible evidence** over previously performed (and failed) actions and questions allowed by the probabilistic model in question.*

5. *A **system test** W (also called $Working?$) with $sp(W) = \{yes, no\}$ which may be performed after any action to verify whether the problem has been removed by the previous action(s). Furthermore, we demand that the probability that troubleshooting must continue (because the device is still broken) can be derived in some fashion. We write this probability as $P_w(\boldsymbol{\epsilon})$ for $\boldsymbol{\epsilon} \in \mathcal{E}$.*

6. *A **cost function** $\mathcal{C} : (\mathcal{A} \cup \mathcal{Q} \cup \{W\}) \times \mathcal{E} \mapsto [1, \infty)$ describing the required resources of performing an action or question given all possible evidence.*

Remark 2. *Having 1 as a lower bound on the costs is merely numerical convenience to diminish floating point inaccuracies; as we saw earlier, we can safely scale our utilities linearly to achieve this.*

It should be noted that the notation $P_w(\boldsymbol{\epsilon})$ is somewhat confusing. The probability that troubleshooting must continue *after* seeing evidence $\boldsymbol{\epsilon}$ is

not a conditional probability. Specifically, it is not the same as the conditional probability that we attach to the outgoing link labeled "no" from a $Working?$ chance node in a decision tree (cf. Figure 3.1). Rather, it should be equal to the *joint* probability that we may associate with the node n that is the endpoint of the link. The reason that we do not simply call the probability for $P(\boldsymbol{\epsilon})$ is that we want to use it as a sanity check (or constraint) on the actual $P(\boldsymbol{\epsilon})$ induced by the probabilistic model(s) to ensure this model is sound (under certain reasonable assumptions). For example, we may always compute $P(\boldsymbol{\epsilon})$ from the decision tree, but that does not imply that this value agrees with other ways to compute $P_w(\boldsymbol{\epsilon})$ from the probabilistic model(s). This distinction will become clear when we discuss multi-fault troubleshooting models later in this chapter.

3.3.1 Basic Troubleshooting Assumptions

First of all, we are dealing with a malfunctioning device which is known to be broken when troubleshooting begins. We need to model this explicitly as it has profound impact on how we may construct a sound model for our decision problem. As such, this is not necessarily a simplifying assumption.

Basic Assumption 1 (The Faulty Device Assumption)**.** *The device is faulty when troubleshooting begins.*

To describe the potential causes of the faulty device, we have an exhaustive set of **faults** $\mathcal{F}$ which may be either present or absent. For example, with $f \in \mathcal{F}$, we may talk about the probability of the fault begin present $P(f = "present")$ and absent $P(f = "not\ present")$ which we write concisely as $P(f)$ and $P(\neg f)$. Assumption 1 leads directly to the following constraint:

Constraint 1 (Fault Probabilities)**.** *The fault probabilities of a troubleshooting model must obey*

$$\sum_{f \in \mathcal{F}} P(f) \quad \geq \quad 1 \tag{3.5}$$

when troubleshooting begins. Furthermore, for all $\boldsymbol{\epsilon} \in \mathcal{E}$

$$\sum_{f \in \mathcal{F}} P(f|\boldsymbol{\epsilon}) \quad \geq \quad 1 \tag{3.6}$$

must also be true when the system test has just confirmed that the device is still malfunctioning.

Now, to repair the device we need to apply a number of repair operations which shall come from a set of **actions** $\mathcal{A}$. Each action $\alpha \in \mathcal{A}$ may address any non-empty subset of the faults in $\mathcal{F}$, and by performing an action we

change the distribution $P(\mathcal{F})$ into some conditional distribution (specifically, conditioned on the fact that the action was performed and that the subsequent system test returned that the device was malfunctioning). We describe this compound event of performing an action α and observing its outcome by the evidence $\alpha = \neg a$ and $\alpha = a$ thereby stating that the action failed or succeeded, repectively (so $sp(\alpha) = \{a, \neg a\}$, and we use the concise notation $P(a)$ and $P(\neg a)$). Since the particular way that we model actions will affect our notation, we shall defer further notation to a later section.

To improve the repair process we may perform questions $Q \in \mathcal{Q}$ which are simply modelled via a normal chance node for each question. The questions can help narrow down the potential causes and provide evidence that affects the optimal ordering of actions. We then use the term **step** to denote either a question or an action. This leads us to our next assumption.

Basic Assumption 2 (The Carefulness Assumption)**.** *Each action and question have no unintended side-effects, that is, by performing an action or question no new faults can be introduced.*

We consider the above assumption as reasonable in device-repair scenarios. For example, when dealing with complicated equipment, the actions are usually carried out by trained professionals who know what they are doing. The assumption is simplifying in the sense that we do not have to model how performed actions or question may have altered the state of the equipment. Arguably, a model without this assumption is *potentially* more accurate, but the knowledge elicitation task is also somewhat more complex. In a medical treatment situation the assumption may be deemed unrealistic; for example, giving a certain drug to a patient may cure one disease while inflicting or worsening others. The assumption leads directly to the following constraint.

Constraint 2 (Monotone Faulty Device Probability)**.** *For all pairs of evidence $\boldsymbol{\eta}, \boldsymbol{\epsilon} \in \mathcal{E}$ such that $\boldsymbol{\eta} \subset \boldsymbol{\epsilon}$, we must have*

$$P_w(\boldsymbol{\eta}) \quad \geq \quad P_w(\boldsymbol{\epsilon}) \,, \tag{3.7}$$

that is, the more steps we perform, the more likely it is that the device is repaired.

We allow for equality to model, e.g., that an action has no effect whatsoever. This can easily happen in a model if two actions address the same fault: after performing the first action, the second action may be of no use.

The next assumption simplifies models with **imperfect** actions, that is, actions that are not guaranteed to repair the set of faults that they can address.

Basic Assumption 3 (The Idempotent Action Assumption)**.** *Repeating a failed action will not fix the problem.*

This assumption can be seen as reasonable if the person carrying out the repair process does not improve his skills during the troubleshooting session. In reality, one needs to provide detailed and easy-to-follow descriptions of how to carry out the particular action to avoid breaking this assumption. In any case, the assumption puts *some* bound on the time horizon that we need to consider for the troubleshooting process.

Constraint 3 (End of Troubleshooting)**.** *The troubleshooting process continues until the system is working or until all actions have been performed in which case the system may still be malfunctioning.*

To get an definite bound on the time horizon, we furthermore limit questions in the following manner.

Basic Assumption 4 (The Idempotent Question Assumption)**.** *Repeating a question will provide the same answer as the first time that the question was asked.*

Remark 3. *Notice that the system test W is inherently different from questions: repeating the system test will often provide a different answer.*

One may ask, what if (a) the question identified a certain fault as a cause of the problem with absolute certainty, and (b) there are multiple faults present in the domain? Would this not mean that the question could now give a different answer after we have repaired the fault that the question identified? We have to acknowledge that this could happen because actions can *change* the world observed by questions. In the case where there is only one fault present, and when the question has identified this fault with certainty, all that is left is to perform an action that can repair this fault—the result is a functioning device and therefore there is no reason to model further troubleshooting.

When there are several faults present, the situation becomes somewhat more complicated. However, if we want to remove the two idempotence assumptions above, we should probably do *more* than merely allow actions and questions to be repeated; in (Breese and Heckerman, 1996) and (Pernestål, 2009) one may find interesting modelling techniques that describe the concept of **persistence** in detail, explicitly modelling when previously performed steps and faults are persistent or invalidated by an action.

Basic Assumption 5 (The Faults Are Partially Observable)**.** *We can never guarantee that the true state of all the faults can be observed directly, and so we may only get indirect evidence provided by questions.*

This assumption is made because we deem it too costly or simply impossible to inspect the device and arrive at the true cause(s) of the problem. In principle, at least when we talk about man-made devices, we should be able to discover the true causes of the problem. However, such evidence

is often costly to observe and we might get almost as good evidence in a much cheaper way. For example, if the battery is broken in a car, we could apply some testing procedure to the battery directly, but it would be much easier just to turn on the lights and see what happens—of course, if the lights do not turn on, this could also be a problem with the electric wiring of the car.

Basic Assumption 6 (Imperfect Actions and Questions)**.** *The outcome of an action may be uncertain, that is, it may fail to fix the faults that it can address. Furthermore, questions may provide non-conclusive evidence on the possible faults in the system.*

These last two assumptions implies that a troubleshooting model have to cope with a great deal of uncertainty, and this uncertainty is most naturally modelled via a Bayesian network. The uncertainty associated with actions serve two primary goals: (a) abstraction over the differences in competences of the repair men, for example, skilled repair men always succeed whereas less skilled sometimes fail to carry out the repair correctly, and (b) abstraction over unobservable variables or inaccuracies in the model, for example, a drug might cure cancer with a certain probability. The uncertainty of actions may also be used to model somewhat unlikely events; for example, if one replaces the spark plugs of a car, then there is a small chance that the new spark plugs are also broken.

Basic Assumption 7 (Perfect System Test)**.** *The system test W (which is usually carried out immediately after each action) is assumed to be perfect, that is, if the test says that the device is malfunctioning, then it is really malfunctioning, and if the test says that the device is working, then it is really working.*

This is not as strict an assumption as it may sound like. After all, if one cannot determine if the problem has been solved, then one cannot determine if the problem exists. But this basic fact is implied by our very first assumption. Finally, we have the following assumption:

Basic Assumption 8 (The Asymmetrical Assumption)**.** *There is a partial order (but never a total order) imposed on actions and questions, that is, we are allowed to carry out a sequence of actions and observation as long as the sequence obey the constraints implied by the partial order. In particular, the partial order may be "empty" which implies that there are no order constraints whatsoever imposed on actions and questions.*

Remark 4. *In the remainder of this thesis we mostly ignore such order constraints and develop our theory and results under the assumption of complete asymmetry.*

This ends our discussion of assumptions that all troubleshooting models should adhere to. In the following we shall investigate more assumptions, but these are specific to the particular type of troubleshooting model.

3.3.2 Expected Cost of Repair

We are now ready to give a formal definition of the troubleshooting decision problem. One may view the following definitions as a concise summary of the discussion above.

Definition 11 (Troubleshooting Tree)**.** *A **troubleshooting tree** is a decision tree encoding all possible sequences of actions (from $\mathcal{A}$) and observations (from $\mathcal{Q}$) with the following constraints:*

1. *each decision node D has $sp(D) \subseteq \mathcal{A} \cup \mathcal{Q}$,*
2. *whenever each outgoing edge of a decision node is labelled with an action from $\mathcal{A}$, it leads to a chance node describing the application of the system test W,*
3. *the outgoing edge labelled "yes" from W is always followed by a utility node because this indicates that the problem was fixed,*
4. *the parent node of a utility node is always a system test node W,*
5. *on any path from the root node to a leaf node, an action or question appears at most once, and*
6. *on any path from the root node to a leaf node, no question appears after all actions have appeared.*

An example of this type of decision tree is depicted in Figure 3.1 on page 28, describing the process of performing repair actions and observations until the problem is fixed or all actions have been performed. In this case, the tree is completely asymmetrical, having no order constraints on actions and questions. As usual, a decision tree encodes all possible strategies:

Definition 12 (Troubleshooting Strategy)**.** *A **troubleshooting strategy** is a strategy in a troubleshooting tree.*

Note that this is often called a **strategy tree** (Vomlelová, 2003). In Figure 3.3 we can see an example of a troubleshooting strategy. Since the system test appears after each action, it is customary to leave out the system test; furthermore, instead of having decision nodes with outgoing edges labelled with actions and questions, these can be replaced with a node representing the action or question directly. Of course, this concise notation is only possible for strategies where each decision node is followed by exactly one outgoing edge.

We can now give a definition of the measure used to compare different troubleshooting strategies, but first we need a little terminology. The set of all **terminal nodes** (or **leaf nodes**) of a troubleshooting strategy s is de-

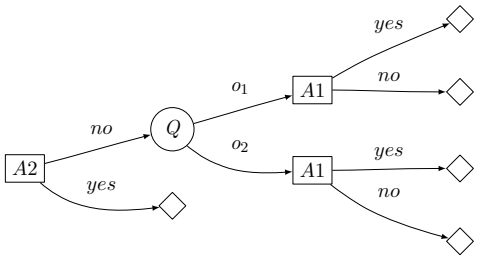


Figure 3.3: Troubleshooting strategy that is equivalent to the strategy highlighted in the decision tree in Figure 3.1 on page 28.

noted $\mathcal{L}(s)$ whereas the set of all **non-terminal nodes** is denoted $nodes(s)$. The **label** of an edge e or node n is given by $label(e)$ and $label(n)$, respectively. The **evidence along a path** $path(n)$ from the root node to n is given by $\boldsymbol{\epsilon}^n = \bigcup_{e \in path(n)} \{label(e)\}$. Even though a set of evidence $\boldsymbol{\epsilon}$ is really a set of instantiations of variables, we shall misuse notation a little and allow $\boldsymbol{\epsilon}$ to be interpreted as a set of actions and questions. We use that interpretation in the definition below.

Definition 13 ((Vomlelová and Vomlel, 2003))**.** *Let s be a troubleshooting strategy, and let $\mathcal{L}(s)$ be the set of all leaves of s. The **expected cost of repair** of s is defined as*

$$ECR(s) = \sum_{\ell \in \mathcal{L}(s)} \left(P\left(\boldsymbol{\epsilon}^\ell\right) \cdot \left[\sum_{\alpha \in \boldsymbol{\epsilon}^\ell \cap \mathcal{A}} C_\alpha + \sum_{Q \in \boldsymbol{\epsilon}^\ell \cap \mathcal{Q}} C_Q \right] \right) \tag{3.8}$$

where $P\left(\boldsymbol{\epsilon}^\ell\right)$ is the probability of getting to the leaf ℓ where we have accumulated evidence $\boldsymbol{\epsilon}^\ell$, and C_α and C_Q are the cost associated with performing an action and a question, respectively.

It is worth noticing that this definition does not allow the cost of actions and questions to depend on the decisions taken before they are carried out. We shall deal with special scenarios where steps have **dependent costs** in later chapters. Similarly, if the cost of the system test C_W is greater than zero, it should be included in the cost of actions in the formula above; this topic we shall also revisit in a later chapter.

Remark 5. *Equation 3.8 is often defined to include a **penalty term** $C_\ell > 0$ in the sum of costs associated with a leaf node ℓ. The interpretation given to this penalty is usually that it is the cost of **call service**, that is, expert help from the outside. As we shall see later, (a) if this cost is the same for all leaf nodes where the problem has not been solved, and (b) the model includes no questions, then we need not model the penalty at all. In the general situation, on the other hand, it is*

very difficult to predict how it will affect the choice of an optimal strategy. Under the assumption that the penalty is the same for all leaves, there is, however, a more natural and consistent way to include this aspect into the decision model: simply add a special action that is guaranteed to solve the problem (thereby modelling the expert help). This has the benefit that we are not forced to use call service only after every normal action has been carried out, but we leave it up to the usual algorithms to compute when it is beneficial to use call service. When the call service cost is not constant, we have the problem of specifying an exponential number of such cost, making elicitation impractical. As such, it is much simpler (and reasonable) just to assume that this penalty is (a) not modelled, (b) modelled with a constant cost, or (c) modelled with a special action.

Unfortunately, modelling call service with a special action is not trivial either because troubleshooting is ended after performing this action. This is dicussed briefly in (Heckerman et al., 1995). In general, we can say that the smaller this cost is, the more important it is to model it with a normal action. This is because if the cost is sufficiently high, it will never be beneficial to apply this action before any of the other actions (cf. also Lemma 4.1 in (Vomlelová, 2001)). If we do not model the cost of call service, it correponds to a situation were a service center seeks to minimize contact with the customers, that is, the customers should try to help themselves as much as possible; on the other hand, if the service center wishes to maximize the customers' satisfaction, it would be wise to allow call service earlier. And in this latter case, it makes more sense to model call service as an action. The former case is also common: we might be in the situation that there is no call service option; for example, the repair men on a factory floor are already the experts in repairing the machinery, in which case we need not model call service. As an example of where such penalties are very important to model, we refer to models with ***misclassification costs*** *(Bilgic and Getoor, 2007).*

Just as in the case for general decision trees, we define our optimization problem as the task of finding an optimal strategy:

Definition 14 (Optimal Troubleshooting Strategy)**.** *Let $\mathcal{T}$ be a troubleshooting tree, and let $\mathcal{S}(\mathcal{T})$ be the set of all troubleshooting strategies in $\mathcal{T}$. Then an* ***optimal troubleshooting strategy*** *s^* is given by*

$$s^* = \arg\min_{s \in \mathcal{S}(\mathcal{T})} ECR(s) . \tag{3.9}$$

Let us first see how the definition of ECR may be calculated in various contexts. In many contexts we are interested in ignoring the questions of the model, leaving only actions as decisions in the model. In this case, a troubleshooting strategy degenerates into a sequence of actions as illustrated in Figure 3.4, and we call such a sequence for a **troubleshooting sequence**. The evidence that appears in such a strategy consists solely of evidence on actions, and we shall call such evidence for **sequence evidence** and denote

it by $\boldsymbol{\epsilon}^i = \{\neg a_1, \ldots, \neg a_i\}$ thereby stating that the first i actions have failed to fix the problem. Note that $\boldsymbol{\epsilon}^0 = \emptyset$ and $P(\boldsymbol{\epsilon}^0) \equiv 1$ by convention because the device is faulty when troubleshooting starts. For $y \geq x$ we also allow the notation $\boldsymbol{\epsilon}^{x:y} = \{\neg a_x, \ldots, \neg a_y\}$ for evidence of a subsequence (and note that $\boldsymbol{\epsilon}^{0:y} \equiv \boldsymbol{\epsilon}^{1:y} \equiv \boldsymbol{\epsilon}^y$). The ECR may then be computed by the following formula.

Proposition 1 (Vomlelová (2001))**.** *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be a troubleshooting sequence. Then the expected cost of repair can be computed as*

$$ECR\,(s) = \sum_{i=1}^{n} P\big(\boldsymbol{\epsilon}^{i-1}\big) \cdot C_{\alpha_i}\,. \tag{3.10}$$

Proof. Any troubleshooting strategy corresponding to a troubleshooting sequence s has $|s| + 1 = n + 1$, and by Definition 13 we then get $n + 1$ terms (cf. Figure 3.4):

$$\begin{aligned} ECR\,(s) = {} & P(a_1) \cdot C_{\alpha_1} + P(\neg a_1, a_2) \cdot (C_{\alpha_1} + C_{\alpha_2}) + \cdots + \\ & P(\neg a_1, \ldots, \neg a_{n-1}, a_n) \cdot \left(\sum_{i=1}^{n} C_{\alpha_i} \right) + P(\neg a_1, \ldots, \neg a_n) \cdot \left(\sum_{i=1}^{n} C_{\alpha_i} \right) \end{aligned}$$

Then we exploit the identity

$$P\Big(\boldsymbol{\epsilon}^{1:k-1}, a_k\Big) = \Big[1 - P\Big(\neg a_k \,|\boldsymbol{\epsilon}^{1:k-1}\Big)\Big] \cdot P\Big(\boldsymbol{\epsilon}^{1:k-1}\Big) = P\Big(\boldsymbol{\epsilon}^{1:k-1}\Big) - P\Big(\boldsymbol{\epsilon}^{1:k}\Big)$$

for all $k \in \{2, \ldots, n\}$ and get:

$$\begin{aligned} ECR\,(s) &= \big[1 - P(\boldsymbol{\epsilon}^1)\big] \cdot C_{\alpha_1} + \big[P(\boldsymbol{\epsilon}^1) - P(\boldsymbol{\epsilon}^{1:2})\big] \cdot (C_{\alpha_1} + C_{\alpha_2}) + \cdots + \\ &\quad \big[P(\boldsymbol{\epsilon}^{1:n-1}) - P(\boldsymbol{\epsilon}^{1:n})\big] \cdot \left(\sum_{i=1}^{n} C_{\alpha_i} \right) + P(\boldsymbol{\epsilon}^{1:n}) \cdot \left(\sum_{i=1}^{n} C_{\alpha_i} \right) \\ &= C_{\alpha_1} + \sum_{i=2}^{n} P(\boldsymbol{\epsilon}^{1:i-1}) \cdot C_{\alpha_i} \end{aligned}$$

and with the assumption $P(\boldsymbol{\epsilon}^0) \equiv 1$ and since $P(\boldsymbol{\epsilon}^{1:i}) \equiv P(\boldsymbol{\epsilon}^i)$, the result follows. □

Remark 6. *Had we included a constant penalty term C_{pen} in the utility for the branch where the problem is not solved, it is easy to see that it would have resulted in an additional term $P(\boldsymbol{\epsilon}^{1:n}) \cdot C_{pen}$ which is the same for all troubleshooting sequences and hence pointless to model.*

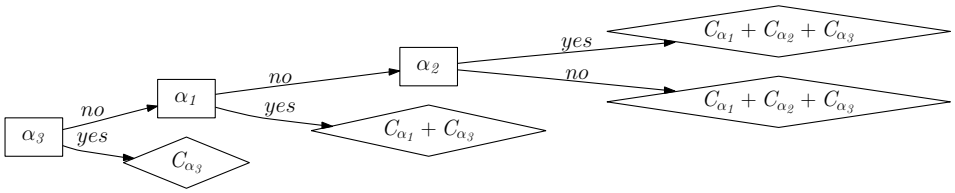


Figure 3.4: Troubleshooting strategy for a model that has only actions. In this case the tree degenerates into a sequence of actions.

Proposition 2. *Let s be a troubleshooting strategy, and let $nodes(s)$ be the set of all non-leaf nodes of s. If no leaf node contains a penalty term, then the ECR of s may be calculated as*

$$ECR\,(s) \quad = \quad \sum_{n \in nodes(s)} P(\boldsymbol{\epsilon}^n) \cdot C_{label(n)} \tag{3.11}$$

that is, the sum over all troubleshooting steps of the probability of performing the step multiplied with the cost of the step.

Proof. We look at any node n, and let $\mathcal{L}(n)$ be all the leaves reachable from n. It is then clear that $P(\boldsymbol{\epsilon}^n) = \sum_{\ell \in \mathcal{L}(n)} P\left(\boldsymbol{\epsilon}^\ell\right)$ and from Equation 3.8 on page 39 and because n was chosen arbitrarily, the results follows. On the other hand, had Equation 3.8 included a penalty term, the result would no longer hold. □

The above theorem is important since it allows for more efficient top-down solution procedures where partially expanded strategies can be used to estimate the ECR of the full strategy. We shall see more about this in subsequent chapters. Finally, the following proposition completes the picture, and we are back to where we started.

Proposition 3. *Definition 7 of expected utility is equivalent to Definition 13 of expected cost of repair (also in the case when penalty terms are included).*

Proof. First we note that the utility $U(past(V))$ assigned to leaf nodes is exactly the cost terms of Equation 3.8 (penalty terms can be included without any problems). Secondly, the probability for a leaf node $P\left(\boldsymbol{\epsilon}^\ell\right)$ equals the product along a path from the root to the leaf ℓ:

$$P\left(\boldsymbol{\epsilon}^\ell\right) = \prod_{n \in past(\ell) \cup \{\ell\}} P(n \,|\, past(n))$$

remembering $P(n \,|\, past(n)) \equiv 1$ when n is a decision node. □

This leads us to a slightly more tangible formulation:

Proposition 4. *Let s be a troubleshooting strategy without penalty terms and with root ρ. Then the ECR of s may be computed as*

$$ECR(s) = \begin{cases} C_Q + \sum_{q \in Q} P(Q = q | \boldsymbol{\epsilon}^Q) \cdot ECR(succ(Q = q)) & \text{if } \rho \text{ is a question } Q \\ C_\alpha + P(\alpha = \neg a | \boldsymbol{\epsilon}^\alpha) \cdot ECR(succ(\alpha = \neg a)) & \text{if } \rho \text{ is an action } \alpha \\ 0 & \text{if } \rho \text{ is a utility node.} \end{cases}$$

Proof. Follows immediately from Proposition 3. □

3.4 The Single-Fault Troubleshooting Model

We shall now review the troubleshooting model that underlies all the work in this thesis. The model forms the backbone of the seminal work found in (Heckerman et al., 1995), (Breese and Heckerman, 1996) and (Jensen et al., 2001). In particular, it distinguishes itself from other models by adding the following assumption:

Basic Assumption 9 (The Single-fault Assumption)**.** *The device is malfunctioning due to a single fault.*

In turn, this assumption leads to an elegant, simple and very efficient model for the troubleshooting tasks. Below is given the formal definition.

Definition 15 (Single-fault Troubleshooting Model)**.** *A **single fault troubleshooting model** M is a 6-tuple $(\mathcal{F}, \mathcal{A}, \mathcal{Q}, W, BN, \mathcal{C})$ where*

- *$\mathcal{F}$ is the set of faults,*
- *$\mathcal{A}$ is the set of actions,*
- *$\mathcal{Q}$ is the set of questions,*
- *W is the system test,*
- *BN is a naive Bayesian network with one top node representing $\mathcal{F}$, and this top node has one child node for each action and question. Moreover, $P(\alpha | \mathcal{F} = f)$ represents how likely an action is to resolve a particular fault and $P(Q | \mathcal{F} = f)$ represents the distribution over Q if f is present, and*
- *$\mathcal{C} : (\mathcal{A} \cup \mathcal{Q} \cup \{W\}) \times \mathcal{E} \mapsto [1, \infty)$ is a cost function describing the cost of performing a step.*

Furthermore, the probability that troubleshooting must continue after performing action α is defined as

$$P_w(\boldsymbol{\epsilon}) = \begin{cases} 1 & \text{if } \boldsymbol{\epsilon} = \emptyset \\ P(\neg a | \boldsymbol{\eta}) \cdot P(\boldsymbol{\eta}) = P(\boldsymbol{\epsilon}) & \text{otherwise} \end{cases} \tag{3.12}$$

where $\{\neg a\} \cup \boldsymbol{\eta} = \boldsymbol{\epsilon}, \boldsymbol{\epsilon} \neq \boldsymbol{\eta}$ and $\boldsymbol{\eta}$ may contain evidence from questions.

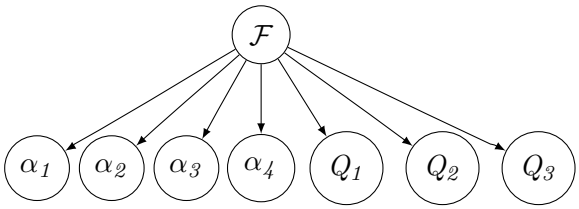


Figure 3.5: An example of a naive Bayesian network structure used in single-fault troubleshooting models.

An example of a naive Bayesian network structure used in a single-fault troubleshooting model is given in Figure 3.5. This model has several advantages:

1. it is a simple model with a low number of parameters (probabilities and costs) which leads to a simpler elicitation task (see (Skaanning et al., 1999) for a discussion),
2. it covers many troubleshooting cases without introducing assumptions that are not true or far from the reality of the domains, and
3. it is very efficient.

The most restrictive assumption is probably the d-separation properties implied by the naive Bayes network structure:

Basic Assumption 10 (d-separation in single-fault troubleshooting)**.** *The single-fault troubleshooting model assumes the following d-separation properties:*

$$S_1 \perp S_2 \,|\, \mathcal{F} \qquad \textit{whenever } S_1, S_2 \in \mathcal{A} \cup \mathcal{Q} \textit{ and } S_1 \neq S_2 \,. \tag{3.13}$$

For many repair scenarios, the above assumption and the single-fault assumption are probably acceptable.

As for efficiency, a full propagation in the Bayesian network takes only $O(|\mathcal{F}|\cdot|\mathcal{A}\cup\mathcal{Q}|)$ time and storing the model merely takes $O(|\mathcal{F}|\cdot|\mathcal{A}\cup\mathcal{Q}|)$ memory. Furthermore, the probability $P_w(\boldsymbol{\epsilon})$ can be easily computed from the nodes representing actions using the chain rule. Of course, this efficiency during propagation does not imply that troubleshooting tasks themselves are efficient—as we shall see, the majority of troubleshooting tasks are in fact NP-hard. However, as both heuristics and solution methods need to make thousands or even millions of propagations, a model which allows fast propagations has its merits.

Finally, let us review how the constraints from Section 3.3.1 are satisfied by the single-fault model. Due to the single-fault assumption,

$$\sum_{f\in\mathcal{F}} P(f|\boldsymbol{\epsilon}) \geq 1$$

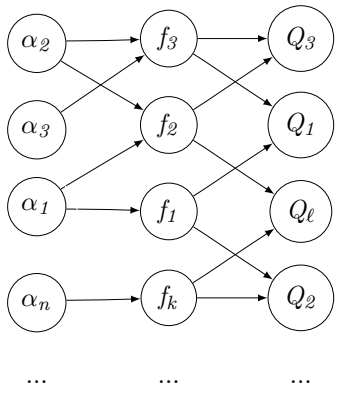


Figure 3.6: An initial attempt of a Bayesian network structure for multi-fault troubleshooting. Both action and fault nodes have a different interpretation than in the single-fault model that we discussed earlier.

is trivially satisfied for all $\boldsymbol{\epsilon} \in \mathcal{E}$ by having a single chance node with all faults as states. Similarly, from Equation 3.12 it is clear that Constraint 2 ($P_w(\boldsymbol{\eta}) \geq P_w(\boldsymbol{\epsilon})$) is satisfied since $P(\neg a \,|\, \boldsymbol{\eta}) \leq 1$.

3.5 Models for Troubleshooting Multiple Faults

The importance of the single-fault assumption should not be underestimated. In this section we shall discuss possible multi-fault troubleshooting models and the advantages and disadvantages of such models. This will also increase our understanding of the single-fault troubleshooting model.

To this date, we believe that no completely general and sound multi-fault model has been devised. We refer the reader to the interesting work of (Pernestål, 2009) (Section III.3) and (Langseth and Jensen, 2003) for earlier research in this area. In particular, the method of (Langseth and Jensen, 2003) groups multiple components into (potentially overlapping) **cut sets**; the equipment is then faulty if *all* components of a cut set are broken, and only one cut set is assumed to be in this state when troubleshooting starts (thereby we have a single-fault assumption on the cut set level, but a multi-fault assumption on the component level).

Below we discuss a multi-fault model that have not been described or published elsewhere. As we shall see, we will fail to come up with a theoretically and practically satisfying multi-fault model. The intention of this section is therefore to highlight the issues relating to this type of troubleshooting model. We hope that by making these problems explicit, we have taken the first step towards a coherent multi-fault model.

The cost parameters and the underlying troubleshooting trees remain essentially the same, and so this discussion focuses on the computation of probabilities. To make the discussion more tangible, let us consider the Bayesian network structure in Figure 3.6. There are two significant changes.

First, instead of a single fault node with a state for each fault in $\mathcal{F}$, we have a node for each fault f with $sp(f) = \{\text{"working"}, \text{"not working"}\}$ which we write $\{w, \neg w\}$ in short-hand. Of course, this means that we can correctly model that several faults are present at the same time. Even though the graph does not show any direct dependencies between fault nodes (or between question nodes), we can of course also do that now.

Secondly, the semantics of action nodes are forced to change somewhat. In the single-fault model we get $P_w(\boldsymbol{\epsilon})$ from $P(\boldsymbol{\eta})$ and $P(\neg\alpha\,|\boldsymbol{\eta})$. However, this is no longer possible, as there may be multiple faults present. This implies that the interpretation of $P(\alpha = y\,|\boldsymbol{\epsilon})$ as "the probability that this action will solve the problem when carried out (given evidence $\boldsymbol{\epsilon}$)" is no longer valid. Instead we model the effect that actions have on the distribution over $P(\mathcal{F} \cup \mathcal{Q})$ by making them parents of the fault nodes. Furthermore, action nodes α now have $sp(\alpha) = \{\text{"performed"}, \text{"not performed"}\}$ and an action node is always instantiated to one of these states depending on whether we have performed the action or not. We can then define

$$P_w(\boldsymbol{\epsilon}) \;=\; \begin{cases} 1 & \text{if } \boldsymbol{\epsilon} = \emptyset \\ 1 - P(f_1 = w, \ldots, f_k = w\,|\boldsymbol{\epsilon}) & \text{otherwise} \end{cases} \tag{3.14}$$

because the device is working if and only if all faults are repaired. Instead of viewing the Bayesian network as one big network, we can view it as $2^{|\mathcal{A}|}$ networks induced by the configurations of the action variables.

At this point we can give the following comparison with the single-fault model:

1. It is worth noticing that d-separation properties remain the same for the nodes representing questions (knowing the state of all fault node, the question nodes are d-separated).

2. The elicitation task is slightly harder for actions, but the fact we may have several parents of a fault node increases the expressive power of the model; for example, we may now specify that the combined effect of two actions is the same as any of the two actions in isolation—such an effect was not possible in the single-fault model (except for the special case where the actions fix the fault with certainty).

3. The elicitation task is similarly somewhat harder for questions because the conditional probability tables grow exponentially with the number of parents. Several approaches have been suggested to deal with this problem, notably Noisy-Or models (Pearl, 1986), (Srinivas, 1993) and NIN-AND Tree models (Xiang, 2010).

4. We constantly need to compute $P(f_1 = w, \ldots, f_k = w \,|\boldsymbol{\epsilon})$ which should not be difficult because the computation is greatly simplified by (a) the presence of evidence on the fault nodes, and (b) the many **barren** question variables (that is, variables where neither themselves nor their descendants have received evidence).

5. Updating the Bayesian network when changing the instantation of an action or when instantiating a questions is likely to be a computationally expensive operation. Unfortunately, this step can not be avoided as we need to get the marginal distribution on the question nodes (either when building a strategy or for heuristic procedures).

Given the above, it seems like the multi-fault model should be an attractive troubleshooting model yielding more precision at the expense of extra complexity of elicitation and inference. However, the two constraints of Section 3.3.1 leads to major problems for the multi-fault model.

The first constraint $\sum_{f\in\mathcal{F}} P(f = \neg w \,|\boldsymbol{\epsilon}) \geq 1$ is by no means satisfied automatically by the network structure that we have suggested for the multi-fault model. And, unfortunately, the same can be said for the second constraint given our definition of $P_w(\boldsymbol{\epsilon})$ as any question will influence $P(f_1 = w, \ldots, f_k = w \,|\boldsymbol{\epsilon})$ to become smaller. Let us elaborate a little on this second point. We may compute $P(f_1 = w, \ldots, f_k = w)$ using the chain rule as

$$\begin{aligned} P(f_1 = w, \ldots, f_k = w) &= \sum_{Q_1} \cdots \sum_{Q_\ell} \prod_{i=1}^{k} P(f_i = w) \cdot \prod_{i=1}^{\ell} P(Q_i \,|pa(Q_i) = w) \\ &= \prod_{i=1}^{k} P(f_i = w) \cdot \sum_{Q_1} \cdots \sum_{Q_\ell} \prod_{i=1}^{\ell} P(Q_i \,|pa(Q_i) = w) \\ &= \prod_{i=1}^{k} P(f_i = w) \cdot 1 \end{aligned}$$

Should we get evidence on any question, it will lead to the multiplication with a number less or equal to one, hence increasing $P_w(\boldsymbol{\epsilon})$. One way to resolve this problem could be to relax the constraint such that it only applies to when an action is performed; for example, upon making some medical test, it might be apparent that the patient suffers from many diseases.

Anyway, our biggest hurdle is still Constraint 1. The problem is that we have not explicitly said what should take the place of the single-fault assumption in the multi-fault model. We went from saying "there is exactly one fault" to "we do not know how many faults there are; maybe there are none". Constraint 1 says that we should add the sentence "but there is at least one fault when the device is not working". Thus, we need a way to say, e.g., "there is exactly two faults" or "there is either one fault, two faults

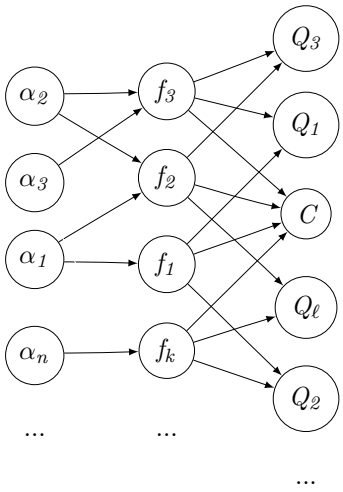


Figure 3.7: A modified Bayesian network structure for multi-fault troubleshooting. A constraint node C is now instantiated and so enforces the modelled constraint; for example, the constraint might be that there is either one or two faults presents.

or three faults" etc. To remedy this problem is actually quite easy, but the ramifications are unpleasant. Consider the network structure in Figure 3.7 where the node C has been introduced as a **constraint node** that enforces our explicit assumptions (see, e.g., Chapter 3 of (Jensen and Nielsen, 2007)). The node has all the fault nodes as its parents, and this is similar to the constraint node described in (Langseth and Jensen, 2003) albeit their constraint node enforces a single-fault assumption. Furthermore, the node needs only have two states: "true" and "false" describing whether the constraint is enforced. For example, to add the explicit assumption that the device has either one or two faults when troubleshooting begins, we specify

$$P(C = true \,|\, pa(C) = \boldsymbol{x}) \quad = \quad \begin{cases} 1 & \text{if } \boldsymbol{x} \text{ contains either one or two faults} \\ 0 & \text{otherwise} \end{cases}$$

and instantiate C to state "true".

The problem has now been reduced to finding a reasonable assumption for the constraint. Unless we have special domain knowledge, the most neutral assumption would be to state that there is either one or two or three or ... (up to some maximum) faults. Another issue is that the constraint node could have unwanted side-effects on other ancestral parts of the network; this is not a problem for the network in Figure 3.7 as all ancestors are instantiated. However, it is not hard to imagine more elaborate models where this could happen. In such a case the methods described in (Crowley et al., 2007) may help.

Remark 7. *The fact that instantiating the constraint node d-connects all the fault nodes should not be seen as a problem because this is exactly what we want.*

While the introduction of the constraint variable C now leads to a reasonable model, the fact that C has every fault node as its parent leads to a model that is very likely to be intractable to do inference in. This can be seen as a *fundamental* problem, not only for troubleshooting, but also for domains like medical diagnosis models: when a patient enters the hospital, we are quite certain that he has at least one disease. Any model that ignores such information is likely to be imprecise. Contrast this with the single-fault model that quite elegantly avoids all these serious modelling problems. (It should be noted that it might be possible to apply **divorcing** to the constraint node; if we are lucky, this could lead to a more tractable model.)

To make matters worse, multi-fault troubleshooting contains additional problems that needs to be addressed. (Heckerman et al., 1995) has the following remark:

> Furthermore, in the single-fault case it is optimal to repair a component immediately after it has been observed to be faulty. In the multi-fault case, this policy is not necessarily optimal, although it typically makes sense to repair a component immediately.

We shall not try to address these problems further, but simply recognize that multi-fault troubleshooting (and diagnosis) is indeed a very difficult problem to formalize.

3.6 Alternative Approaches

We shall now give a brief overview of alternative models that may be used for troubleshooting and decision making in general. They all have in common a form of decision tree formalism as their underlying description of the problem.

3.6.1 Influence Diagrams and Unconstrained Influence Diagrams

The first type of models we shall consider is **Influence Diagrams** (Howard and Matheson, 1981). Decision trees are very flexible and general, but they quickly become too large to be manageable for humans. As a result, Influence Diagrams were created as a compact representation of *symmetric* decision trees (cf. Definition 9). Instead of going into technical details, we shall give a small example, see Figure 3.8.

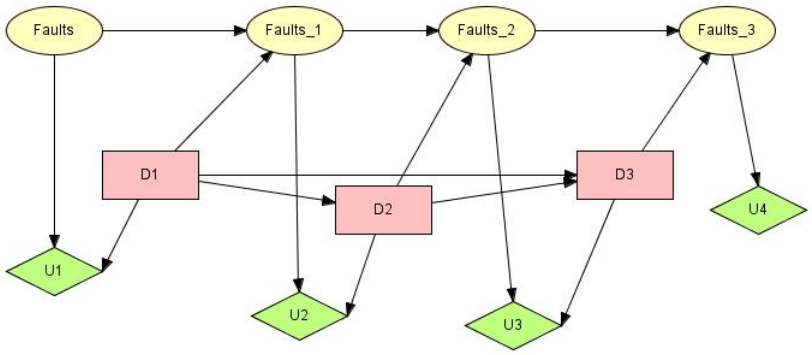


Figure 3.8: An Influence Diagram structure for emulating a troubleshooting model with three actions. We have $sp(D_1) = sp(D_2) = sp(D_3) = \{\alpha_1, \alpha_2, \alpha_3\}$ and each of the *Faults* nodes corresponds (roughly) to the fault node in the single-fault troubleshooting model.

Influence Diagrams extends Bayesian networks with utility nodes (diamond shaped nodes) and decision nodes (rectangular nodes). In this example, all decision nodes have the same states $\{\alpha_1, \alpha_2, \alpha_3\}$ since the model must take into account any sequence of actions. It is required that the decisions are temporarily linearly ordered, and in this case we have that decision D_i is always taken before decision D_{i+1}. $P(Faults)$ corresponds to the initial distribution given in the single-fault troubleshooting model (with the addition of a state "fixed" describing that the problem is resolved) whereas $P(Faults\_1|Faults, D_1 = \alpha_1)$ may look as follows:

	$D_1 = \alpha_1$			
Faults	f_1	f_2	f_3	"fixed"
Faults_1				
f_1	0	0	0	0
f_2	0	1	0	0
f_3	0	0	1	0
"fixed"	1	0	0	1

For example, this table describes that if *Faults* is in state f_1, then *Faults_1* will be in state "fixed" with probability 1, and that this action has no influence on any of the other faults. We also need to specify a table of utilities for all the utility nodes. If all the actions have cost 1, then we may specify $U(Faults = f_1, D_1)$

	$Faults = f_1$		
D_1	α_1	α_2	α_3
	-1	-1	-1

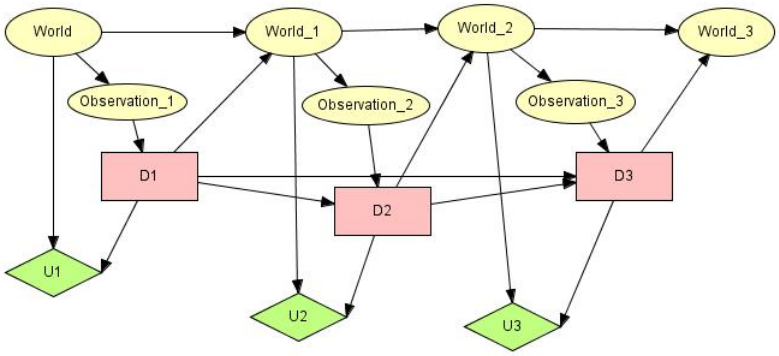


Figure 3.9: An example of a Partially Observable Markov Decision Process modelled via an Influence Diagram. The state of the world is only partially observable via the $Observation$ nodes.

and similarly for f_2 and f_3 as the cost of actions is constant. For $Faults$ = "fixed" we should specify 0 as there is no need to perform an action if the problem is solved. The fourth utility node is used to force the influence diagram to prefer the state of $Faults\_3$ to be "fixed" by giving a negative utility to all other states and zero to state "fixed".

Given the influence diagram description, we may now solve it, that is, compute an optimal strategy. This may be done by unfolding the influence diagram into a decision tree and using Algorithm 1 on page 30. Details of such approaches may be found in (Jensen and Nielsen, 2007) and (Kjærulff and Madsen, 2008). The method described in (Changhe Yuan, 2010) involves a top-down branch-and-bound (depth-first) search and it is the current state-of-the art of solving Influence Diagrams. For Influence Diagrams with many decisions, computing an exact solution is hard, and this has lead to approximate techniques like **Limited Memory Influence Diagrams** (Lauritzen and Nilsson, 2001) where the no-forgetting assumption is relaxed.

The Influence Diagram in Figure 3.8 can be seen as a special case of a larger class of models known as **Partially Observable Markov Decision Processes** (or POMDPs). In the general case, the models take the form illustrated in Figure 3.9 where each decision is preceeded by a partial observation of the current state of the world. The predicate **Markov** is used with any model where the future state of the (modelled) world ($World_{i+1}$) is d-separated from the past (e.g., $World_{i-1}$) given the current state of the world ($World_i$). In general, we may think of Figure 3.9 as a template where each chance node may be stored efficiently (factored) as a Bayesian network.

The single-fault troubleshooting model may be seen as a very simple and efficient POMDP. Interestingly, a seminal paper in the POMDP community uses machine maintenance as a motivating example (Smallwood and Sondik, 1973).

From the preceeding example from Figure 3.8 it was clear that it was rather difficult to emulate troubleshooting models with Influence Diagrams. This is mainly due to two assumptions of the troubleshooting models:

1. Constraint 3 on page 36 stating that troubleshooting ends when no more actions can be performed, and

2. Assumption 8 on page 37 stating that a troubleshooting tree is completely asymmetrical.

One often seen suggestion for Influence Diagrams is to add a "do nothing" action with a zero cost, albeit it increases the dimensionality of the problem. To deal more directly with these limitations, various extensions have been made to Influence Diagrams. For example, (Jensen and Vomlelová, 2002) describes **Unconstrained Influence Diagrams** and (Jensen et al., 2006) investigates **Sequential Influence Diagrams**. Such frameworks enable a quite accurate (and complex) description of decision problems with asymmetries. We suspect that some of the problems encountered with the multi-fault model carries over to these frameworks if we wish to employ them for troubleshooting. A major obstacle is the difficulty of solving such models (see (Luque et al., 2008) and (Ahlmann-Ohlsen et al., 2009) for details). An interesting discussion of Influence Diagrams and troubleshooting may be found in (Gökcay and Bilgic, 2002).

3.6.2 Models Without Actions

In this section we shall briefly discuss two approaches that enable troubleshooting without any explicit representation of actions. The first approach is the one provided by the GeNIe Bayesian network shell developed at the University of Pittsburgh (Druzdzel, 1999), and this discussion is based on the documentation that comes with the tool (version 2.0, 2010).

Basically, the approach requires the creation of a normal Bayesian network over faults and questions. Furthermore, with each question variable one can associate the cost of performing that test. As (repair) actions are not explicitly modelled, the user can decide when to carry them out based on the diagnosis (over faults) offered by the network. If the user feels that the diagnosis is not certain enough, he can then ask the system to rank the questions based on some criterion that seeks to measure the informativeness of the question compared to the cost of the question. In simple terms, a question is worthwhile to answer if it makes us more certain in the correct diagnosis. However, to understand precisely how the system ranks

the questions, we need a few well-known definitions relating to **value-of-information** (VOI) analysis (Kjærulff and Madsen, 2008).

Given a probability distribution $P(\mathcal{X})$ over a set of variables $\mathcal{X}$, the **entropy** of $\mathcal{X}$ is given by

$$H(\mathcal{X}) = -\sum_{\mathbf{x}\in\mathcal{X}} P(\mathbf{x}) \cdot \lg P(\mathbf{x}) \tag{3.15}$$

where $0 \cdot \lg 0 \equiv 0$ (this interpretation we uphold in all formulas involving $x \cdot \lg x$). The entropy achieves its minimum 0 when all probability mass is located in a single state, and it achieves its maximum $\lg |sp(\mathcal{X})|$ when the distribution is uniform. Entropy therefore measures how "focused" the probability distributions is. We may also define the **conditional entropy** over variables $\mathcal{X}$ given variables $\mathcal{Y}$

$$H(\mathcal{X}|\mathcal{Y}) = -\sum_{\mathbf{y}\in\mathcal{Y}} P(\mathbf{y}) \cdot \sum_{\mathbf{x}\in\mathcal{X}} P(\mathbf{x}|\mathbf{y}) \cdot \lg P(\mathbf{x}|\mathbf{y}) \tag{3.16}$$

to measure the uncertainty on $\mathcal{X}$ given an observation on $\mathcal{Y}$. Finally, we define the **mutual information** (or **cross entropy**) as a measure of the value of observing $\mathcal{Y}$ for the target variables $\mathcal{X}$ as

$$\begin{aligned} I(\mathcal{X},\mathcal{Y}) &= H(\mathcal{X}) - H(\mathcal{X}|\mathcal{Y}) \\ &= \sum_{\mathbf{y}\in\mathcal{Y}} P(\mathbf{y}) \cdot \sum_{\mathbf{x}\in\mathcal{X}} P(\mathbf{x}|\mathbf{y}) \cdot \lg \frac{P(\mathbf{x},\mathbf{y})}{P(\mathbf{x}) \cdot P(\mathbf{y})} \,. \end{aligned} \tag{3.17}$$

The cross entropy is a non-negative quantity with a value of zero if and only if $\mathcal{X}$ and $\mathcal{Y}$ are independent sets.

In GeNIe we may call the target variables for the set $\mathcal{F}$. Then, to rank the questions, GeNIe computes the following for each question Q:

$$VOI(Q) = I(\mathcal{F}, Q) - \alpha \cdot C_Q \tag{3.18}$$

where α is a non-negative constant that converts the cost to a number with the same unit as the cross entropy $I(\cdot)$. The constant is chosen by the user when diagnosing. Since (cross) entropy is measured in millibits, α may be converting from time or money to millibits. Compared to methods that rank questions based on mutual information exclusively (e.g., (Rish et al., 2005)), the GiNIe approach can be seen as an improvement as it allows us to take the cost of the question into account.

However, there are several problems with this approach to troubleshooting, especially relating to Equation 3.18. First, there are many choices of α and we may ask which is the best one as different values will lead to different rankings. Secondly—and this is by far the biggest issue—the unit "millibits" is somewhat difficult to understand for humans. Of course, we

can easily say that "more information" is better than "less information", but it is exceedingly hard for humans to determine how one should convert, e.g., minutes to millibits. Conceptually, they have *nothing* in common. Converting between minutes and money is much more straightforward, as we know (or can obtain) the price of salaries, equipment and lost production time.

To avoid converting between utility units, special Multi-currency Influence Diagrams have been proposed (Nielsen et al., 2007). We agree that such considerations cannot be ignored by sound decision models. This leads us to the following principle.

Principle 4 (Utility Conversion). *Any conversion between utilities of two different units can only be considered sound when the relationship between the two units is well-understood and meaningful.*

In the case of GeNIe, it is probably simpler and less error-prone just to avoid the problem altogether by defining

$$VOI(Q) \quad = \quad \frac{I(\mathcal{F}, Q)}{C_Q} \tag{3.19}$$

which also removes the need for the user to select α. This would rank a question highest if it yields the highest information gain per unit cost. As we shall see in the next chapter, fractions of this type appear naturally as a consequence of the decision tree formalism.

Instead of using a full Bayesian network model, let us now consider another class of simpler troubleshooting models which may be characterized by (a) that only a few probability parameters are used, and (b) that no actions are explicitly modelled.

This type of model has been extensively studied in a series of articles (Raghavan et al., 1999a,b,c), and (Tu and Pattipati, 2003) devoted to the problem of **test sequencing**. Furthermore, among the articles are also subjects like multi-fault diagnosis (Shakeri et al., 2000; Tu et al., 2003) and conditional costs parameters (Raghavan et al., 2004). The simplest test sequence problem takes the following form:

1. We have a set of $m + 1$ system states $\mathcal{S} = \{s_0, \ldots, s_m\}$ with s_0 being the fault-free state and the other states denote the m potentially faulty states of the system.

2. For each system state $s \in \mathcal{S}$ we have the prior probability $P(s)$.

3. We have a set of n tests $\mathcal{T} = \{t_1, \ldots, t_n\}$ with respective costs $\mathcal{C} = \{c_1, \ldots, c_n\}$.

4. Finally, a binary matrix $D = [d_{ij}]$ identifies which system states (i) that can be detected by a test (j). So d_{ij} is 1 if t_j can detect if state s_i is faulty and 0 otherwise. The test are assumed correct, and so no uncertainty is involved here.

We can readily translate this specification to our troubleshooting model: (a) $\mathcal{F}$ corresponds (roughly) to $\mathcal{S}$ (and $P(\mathcal{F})$ to $P(S)$), but the inclusion of s_0 means that the problem is not assumed to exist when troubleshooting begins, (b) $\mathcal{T}$ corresponds to $\mathcal{Q}$ with the assumption that all questions are binary, and (c) D corresponds to the conditional probability tables $P(Q|\mathcal{F})$ with the assumption that these relations are deterministic. The deterministic relations are relaxed in (Raghavan et al., 1999b) leading to a POMDP problem (cf. also (Heckerman et al., 1995) for troubleshooting under full observability). In the simplest form, the goal is to find a strategy s over the tests that can identify any faulty state (unambiguously) while minimizing the **expected testing cost**

$$ETC(s) = \sum_{i=0}^{m}\sum_{j=1}^{n} a_{ij} \cdot P(s_i) \cdot c_j \tag{3.20}$$

where a_{ij} is 1 if test t_j is used in the path leading to the identification of system state s_i (in s) and 0 otherwise. In this manner, the problem may be understood as finding a strategy in a special, simple type of decision tree. Although it is not always stated explicitly, we believe there is an underlying single-fault assumption in Equation 3.20. We speculate that any of these special strategies has exactly m leaf nodes (cf. Equation 3.8 on page 39) where a system state is uniquely identified, and furthermore (non-trivially) that such a leaf is reached with probability $P(s_i)$.

3.6.3 The Importance of Detailed Models

Both types of models presented above have the assumption that once a fault is identified (sufficiently accurately), we shall of course repair the component that is involved. On the other hand, in troubleshooting (and Influence Diagrams) we explicitly model actions. The first advantage of this approach is that it allow us to model uncertainties relating to the action itself. The assumption is that such uncertainties are important for a correct model of the world. Secondly, we can model that an action may repair several components. To give a formal argument of why models that mix actions and questions are (theoretically) superior, we define the following special troubleshooting strategies:

$s^{\mathcal{A}}$ The optimal troubleshooting strategy consisting only of actions. The strategy is found by minimizing over strategies in the troubleshooting tree $\mathcal{T}^{\mathcal{A}}$.

$s^{\mathcal{Q}}$ The optimal troubleshooting strategy under the constraint that either (a) all questions have been performed before any action appears, or (b) only questions appear before a set of applicable actions ($P(a\,|\,\epsilon) > 0$) such that if any further questions would appear before the actions, the set of applicable actions would be empty. The strategy is found by minimizing over strategies in the troubleshooting tree $\mathcal{T}^{\mathcal{Q}}$.

$s^{\mathcal{A}\cup\mathcal{Q}}$ The optimal troubleshooting strategy. The strategy is found by minimizing over strategies in the troubleshooting tree $\mathcal{T}^{\mathcal{A}\cup\mathcal{Q}}$ which is a completely general troubleshooting tree mixing actions and questions.

Since (by definition) $\mathcal{T}^{\mathcal{A}} \subseteq \mathcal{T}^{\mathcal{A}\cup\mathcal{Q}}$ and $\mathcal{T}^{\mathcal{Q}} \subseteq \mathcal{T}^{\mathcal{A}\cup\mathcal{Q}}$, we immediately have the following result (cf. Lemma 1.5 in (Vomlelová, 2001)):

Proposition 5. *It always holds that*

$$ECR\left(s^{\mathcal{A}\cup\mathcal{Q}}\right) \leq ECR\left(s^{\mathcal{A}}\right) \quad \textit{and} \quad ECR\left(s^{\mathcal{A}\cup\mathcal{Q}}\right) \leq ECR\left(s^{\mathcal{Q}}\right) \tag{3.21}$$

This may be used to justify mixed models from a theoretical perspective; the practical applications are a different matter. Real world use of decision models require us to strike a balance between how fine-grained and close to reality a model should be compared to how hard it is to specify, maintain, and find optimal or approximate solutions in. We believe that the troubleshooting model introduced in this chapter provides a simple, accurate and practical abstraction for technical repair scenarios.

Remark 8. *The above considerations also allow us to answer a general question about model details from a theoretical perspective. For example, it will always be beneficial to split large actions into smaller actions until the resulting actions are as small as logically possible. An example could be that we are asked to repair the battery of a car, but this actually consists of two independent tasks: (a) check the connections to the battery, and (b) try to recharge the battery and replace it if that fails.*

3.7 Summary

This chapter described modern probabilistic decision theory and, in particular, decision theoretical troubleshooting. We started with a review of utility theory and, in particular, the principle of maximum expected utility as a theoretical basis of modern probabilistic decision theory. We then introduced decision trees as formal description of decision problems that encodes all possible contingencies a decision maker can face when solving the decision problem. Our optimization task is then to find an optimal (with respect to expected utility) sub-tree of the decision tree (that is, a strategy) encoding which action or observation to perform next given the history of previously performed steps.

Decision theoretical troubleshooting was introduced as a special decision problem that stands out by being highly asymmetrical. A range of simplifying assumptions was discussed in depth, and we argued how any troubleshooting model must meet certain constraints to be a valid model of reality. The simplifications induced by the troubleshooting assumptions lead to a definition of troubleshooting trees as simplified decision trees, and it was discussed how we may define the expected cost of repair of a troubleshooting strategy in various equivalent ways.

We then discussed the predominant single-fault troubleshooting model and compared it with a particular multi-fault model. This revealed several problems with multi-fault troubleshooting and enforced our belief in the appropriateness of the single-fault model. A brief description of alternative approaches to troubleshooting was given, ranging from very accurate Influence Diagram-based approaches to quite simple fault isolation approaches. In this process we saw that decision theoretic troubleshooting problems fall in the category of partially observable Markov decision processes. In the end, the single-fault troubleshooting model stands out as an appropriate compromise between complicated and somewhat simpler models for technical repair scenarios.

Chapter 4

Classical Solution Methods for Troubleshooting

> A thousand roads lead men forever to Rome.
> *–Alain de Lille*

We all know the common form of the above proverb as "all roads lead to Rome". While this might be true, we may add that some roads are bound to be shorter than others and hence more interesting if we want to visit Rome. In this chapter we shall start with a short overview of various troubleshooting domains and their complexity. We then investigate in detail how to solve decision scenarios by performing systematic searches in the underlying decision tree representation. The sheer complexity of finding and storing an optimal strategy often forces us to give up that goal and settle for an approximate solution. Therefore we also discuss ways to tweak the solution methods such that they may be used as any-time heurstics with bounded optimality.

4.1 Overview of Troubleshooting Domains

In this section we shall briefly discuss the wide range of assumptions leading to slightly different troubleshooting domains. These different assumptions have a profound impact on tractability and often lead to quite specialized heuristics and solution methods. Let us first introduce these domains. They all have in common the three sets $\mathcal{F}$, $\mathcal{A}$, and $\mathcal{Q}$ that make up the basis of any troubleshooting model.

It would be possible to define many of the domains below irrespective of a single- or multi-fault assumption. However, since no theoretically satisfying, general multi-fault model has been invented, we shall ignore such multi-fault models except when explicitly mentioned. Our first domain requires the following notation. For any action $\alpha \in \mathcal{A}, fa(\alpha) \subseteq \mathcal{F}$ is the **fault set** of the action denoting the set of faults for which $P(a\,|f) > 0$, that is, all the faults f that α may repair.

Definition 16 (Independent/Dependent Actions)**.** *Let* $M = (\mathcal{F}, \mathcal{A}, \mathcal{Q}, W, BN, \mathcal{C})$ *be a troubleshooting model. If for all pairs* $\alpha, \beta \in \mathcal{A}$ *we have that*

$$fa(\alpha) \cap fa(\beta) = \emptyset \tag{4.1}$$

then the model is said to have ***independent actions****—otherwise the model is said to have* ***dependent actions****.*

Models with dependent actions is the topic of Chapter 5.

Definition 17 (Multiple Independent/Dependent Faults)**.** *Assume we associate a binary sample space with each fault* $f \in \mathcal{F}$ *in some multi-fault troubleshooting model. If*

$$P(\mathcal{F}|\boldsymbol{\epsilon}) \;=\; \prod_{f \in \mathcal{F}} P(f) \tag{4.2}$$

for all $\boldsymbol{\epsilon} \in \mathcal{E}$ *then the model is said to have* ***multiple independent faults****—otherwise the model has* ***multiple dependent faults****.*

In general, the costs of actions and questions may depend on evidence $\boldsymbol{\epsilon} \in \mathcal{E}$. If this is the case, we write $C_\alpha(\boldsymbol{\epsilon})$ and $C_Q(\boldsymbol{\epsilon})$ for the associated costs and these are said to be **conditional costs** (or **dependent costs**); otherwise we simply write C_α and C_Q and say that the costs are **constant costs** (or **unconditional costs**).

Definition 18 (Constant/Conditional Costs)**.** *Let* $M = (\mathcal{F}, \mathcal{A}, \mathcal{Q}, W, BN, \mathcal{C})$ *be a troubleshooting model. If all costs are constant costs, the model is said to have* ***constant costs****—otherwise the model is said to have* ***conditional costs****.*

A special type of models with conditional costs deserves its own category.

Definition 19 (Cost Clusters)**.** *Let* $M = (\mathcal{F}, \mathcal{A}, \mathcal{Q}, W, BN, \mathcal{C})$ *be a troubleshooting model. If*

1. *the set* $\mathcal{A} \cup \mathcal{Q}$ *can be partitioned into a set of (non-overlapping)* ***clusters*** $\{\mathcal{K}_1, \ldots, \mathcal{K}_\ell\}$,
2. *the clusters can be arranged in a DAG prescribing the order in which the clusters must be opened to be able to perform the actions and questions in a cluster, and*
3. *for each cluster we have a cost* $C_{\mathcal{K}_i}$ *associated with opening and closing the cluster* given *that a parent cluster is open,*

then the model is said to be a ***cost cluster model****. If the DAG equals a tree, we call the model for a* ***tree cost cluster model****.*

In Chapter 6 we take a closer look at cost cluster models. We shall distinguish between two types of cost cluster models: those with inside information and those without. If the clusters have **inside information**, it means that we do not have to close the clusters again until the problem has been fixed because it is possible to perform the system test when the equipment is in this (partially or fully) open state. On the other hand, if the clusters do not admit inside information, then we are forced to close the open clusters before we can perform the system test. This latter scenario is also closely related to the following:

Definition 20 (Non-trivial System Test)**.** *Let $M = (\mathcal{F}, \mathcal{A}, \mathcal{Q}, W, BN, \mathcal{C})$ be a troubleshooting model. If the system test W has a cost $C_W > 0$, then we say that the model has a **non-trivial system test**.*

We also call this domain for troubleshooting with **postponed system test** because it may not be optimal to perform the system test immediately after an action has been performed. This is the topic of Chapter 7.

In the table below we summarize the scenarios that have been proven to be intractable (see (Vomlelová, 2003) and (Lín, 2011)).

Assumption	**Complexity**
Questions (General)	NP-hard
Dependent actions	NP-hard
Dependent multiple faults	NP-hard
Conditional costs (general)	NP-hard
Cost clusters with no inside information	NP-hard
Non-trivial system test	NP-hard

The fact that troubleshooting with multiple faults is NP-hard may not seem so surprising as belief propagation in Bayesian networks is NP-hard.

Remark 9. *(Lín, 2011) also discusses the notion of **precedence constraints** on the order of actions which again may lead to an NP-hard problem. We shall not discuss such models further, but just remark that search methods have been developed to cope with such constraints while solving a decision problem (Dechter and Mateescu, 2004).*

There are, however, also some more encouraging results. The table below summarizes the types of troubleshooting models that are known to have polynomial complexity. Except for (Heckerman et al., 1995), all these models have in common that they only contain actions (that is, $\mathcal{Q} = \emptyset$) (see (Kadane and Simon, 1977), (Srinivas, 1995), (Ottosen and Jensen, 2010) and (Ottosen and Jensen, 2011)).

Assumption	Complexity
Independent actions	$O(\|\mathcal{A}\| \cdot \lg \|\mathcal{A}\|)$
Independent multiple faults	$O(\|\mathcal{F}\| \cdot \lg \|\mathcal{F}\|)$
Independently and fully observable faults	$O(\|\mathcal{A}\| \cdot \lg \|\mathcal{A}\|)$
Tree cost clusters with inside information	$O(\|\mathcal{A}\| \cdot \lg \|\mathcal{A}\|)$
Non-trivial system test in non-reordable models	$O(\|\mathcal{A}\|^3)$

We may add that the last three domains of course also requires independent actions (see Chapter 6 and Chapter 7). One of the few interesting, remaining problems that have not been investigated is to extend the tree cost cluster model to a DAG of clusters[1].

Remark 10. *The algorithm derived from the assumption "Independently and fully observable faults" above is the one described in (Heckerman et al., 1995). A question in this model can only detect if one particular fault is present, and the tests have no false positives or false negatives. However, the authors appear to have overlooked that their algorithm may not be optimal if the cost of a question is high compared to the cost of the associated repair action. In such cases, it may be better to skip the question and simply perform the repair action.*

Let us end this section with discussion of the actual worst-case complexity of a standard troubleshooting problem. The worst case happens when we have to investigate the entire decision tree induced by the model.

Proposition 6. *Let $M = (\mathcal{F}, \mathcal{A}, \mathcal{Q}, W, BN, \mathcal{C})$ be a troubleshooting model with no special assumptions other than dependent actions and general questions. Let $\gamma(M)$ be a function describing the time it takes time to generate a successor node in the troubleshooting tree induced by M, and let δ be the average number of states for the questions in $\mathcal{Q}$. Then an optimal troubleshooting strategy can be found in $O\left(\gamma(M) \cdot 2^{|\mathcal{A}|+\delta\cdot|\mathcal{Q}|}\right)$ time.*

Proof. By Proposition 2 on page 42 we can restrict our analysis of the troubleshooting tree to the internal nodes. Furthermore, the number of chance nodes is proportional to the number of decision nodes as every decision node is followed by a chance node. The complexity is therefore bounded by the number of decision nodes in the coalesced troubleshooting tree.

This number is most easily estimated by looking at the size of the evidence set $\mathcal{E}$ induced by M. Every time we take a decision, we get to a chance node, and following an outgoing edge of this chance node leave us with new evidence. This can happen at most $|\mathcal{A}| + |\mathcal{Q}|$ times. At each level (depth) i in the troubleshooting tree we have at most

$$\binom{|\mathcal{A}| + \delta \cdot |\mathcal{Q}|}{i}$$

[1] Václav Lín informs me that this is in fact also NP-hard (private communications).

sets of evidence (somewhat less in reality as we may never pick more than one state from the same question). And by summing over all levels we get

$$\sum_{i}^{|\mathcal{A}|+|\mathcal{Q}|} \binom{|\mathcal{A}| + \delta \cdot |\mathcal{Q}|}{i} < \sum_{i}^{|\mathcal{A}|+\delta\cdot|\mathcal{Q}|} \binom{|\mathcal{A}| + \delta \cdot |\mathcal{Q}|}{i} = 2^{|\mathcal{A}|+\delta\cdot|\mathcal{Q}|} .$$

We still need to argue how the coalesced troubleshooting tree can be exploited to get a solution in $O(\gamma(M) \cdot 2^{|\mathcal{A}|+\delta\cdot|\mathcal{Q}|})$ time. This is done by first building the complete troubleshooting tree top-down by performing a breadth-first search—this type of search ensures that coalescing completes before a decision node is expanded. We can then run the bottom-up average-out and fold-back algorithm to ensure no subtree is processed more than once, basically visiting the decision nodes in reverse order (a map of decision nodes may be generated for each level in the troubleshooting tree). □

We have now established an upper bound on the complexity of solving a standard troubleshooting model. In the next sections we shall see how we may avoid exploring the whole troubleshooting tree. The algorithms we shall review are A* , AO* and depth-first search with heuristic information (sometimes referred to as a branch-and-bound algorithm). Even though the algorithms are almost fifty years of age, they remain among the most important algorithms in AI and graphical models in particular (when optimal solutions must be found). Their significance is supported by the fact that many state-of-the art methods are instantiations of these algorithmic templates.

4.2 Solution Methods for Models with Actions

In this section we shall review the basic exhaustive search methods that are guaranteed to find an optimal solution provided that sufficient memory is available (such algorithms are also called **admissible**). We shall focus the discussion on troubleshooting models without questions as this leads to a simpler theoretic exposition. Similar arguments can be made for more complicated models that include questions.

There are several reasons why it makes sense to investigate such algorithms even though the problem may be NP-hard and we have to fall back on heuristics in most practical applications:

1. The algorithms provide us with a systematic tool for evaluating the qualitative performance of heuristics such that we can benchmark different heuristics not merely against each other but also against an optimal solution.

2. The algorithms may be easily transformed into powerful anytime heuristics.

3. The algorithms may be modified such that they can be guaranteed to find a solution within a certain bound of an optimal solution (also called **semi-optimization**).

As we have seen, (probabilistic) decision trees provide us with the formal graphical representation of our troubleshooting decision problems. A key aspect of this graphical representation is that it allows us to formulate the task of finding an optimal strategy as a search procedure in the decision tree. The search procedures we shall review in this section can often perform considerably better than buttom-up algorithms like the average-out and fold back algorithm (Algorithm 1 on page 30). Overall there are two general reasons for this:

1. The algorithms work "top-down", starting at the root and working their way towards the leaves of the decision tree, growing the decision tree incrementally as needed. This enables the algorithms to prune partial solutions that are already too poor to be optimal. In this manner the algorithms can try to avoid doing unneccessary work.

2. The algorithms may use additional **heuristic information** to estimate the remaining cost of a partial solution. If this information obeys certain properties, it enables the algorithms to prune partial solutions earlier which may further reduce the running time.

There are several reasons why a top-down solution method is potentially more efficient and appropriate than bottom-up approaches.

1. Propagation in Bayesian networks is most easily done by adding to the set of evidence rather than starting with a large set of evidence and subtracting from it (although methods do exist for **fast retractions** (Jensen, 1995)).

2. Asymmetries in the decision tree make it more difficult to determine where exactly the leaves are located. Again, this can be investigated by propagating in the Bayesian network (is the evidence set in conflict?), but this may lead to several propagations that would not have been necessary in a top-down approach.

3. In practice we are most interested in estimating the initial sub-tree of a troubleshooting strategy (perhaps only the very first step). This may be accomplished by turning top-down approaches into anytime heuristics whereas a bottom-up approach is inherently unsuitable as a basis for anytime heuristics.

Before we proceed, we need a little terminology. We use Proposition 2 on page 42 (and its variants) as a convenient, formal basis for solving troubleshooting models as shortest path problems. In general we call the troubleshooting tree for the **search graph** and a node in this graph will represent either a decision or chance node. The **start node** is denoted s and t is any **goal node**. Nodes found between s and t will be denoted n and m. If m is a successor node of n, then $c(n, m)$ is the cost associated with the edge between n and m. The **total cost** from the start node to a node n is given by the function $g(n)$ also denoted $c(s, n)$. We let $h(n)$ denote a **heuristic function** that guides (or misguides) the search by estimating the remaining cost of the problem. The **cost of an optimal path** from s to n is denoted $g^*(n)$, and from n to t it is denoted $h^*(s)$. The **smallest cost between two nodes** $c^*(n, m)$ is denoted $k(n, m)$. We also write C^* for $k(s, t)$. Finally, for any node n we call

$$f(n) \quad = \quad g(n) + h(n) \tag{4.3}$$

for the **evaluation function** of n (and we may call the actual values for **f-values** and **g-values**). The evaluation function can used to determine which node that should be expanded next in the search algorithms or it can be used for pruning purposes. If $h(n) \equiv 0$ for all nodes n, we say that the search is **uninformed**—otherwise we call the search **informed**. Since we grow the search graph dynamically, a node can be (a) **unexplored** in which case we have not generated it yet, (b) **explored** if has been generated, and (b) **expanded** if it has been generated and all its successors have been explored as well.

Remark 11. *Although it may seem confusing, it is customary to call $h(\cdot)$ for a "heuristic function", and it should not be confused with a "heuristic" which simply means some general algorithm that is not guaranteed to find an optimal solution.*

We are now interested in a special class of heuristic functions that may be used in the various search algorithms:

Definition 21 (Admissible Heuristic Function)**.** *Let $h(\cdot)$ be a heuristic function for some search problem. If*

$$h(n) \quad \leq \quad h^*(n) \;\; \forall n \tag{4.4}$$

*then $h(\cdot)$ is **admissible**.*

Remark 12. *Even though it is a little confusing, the notation $h^*(n)$ for $\min_t k(n, t)$ should not be confused with the heuristic function $h(n)$. This notation has been customary for decades (Pearl, 1984).*

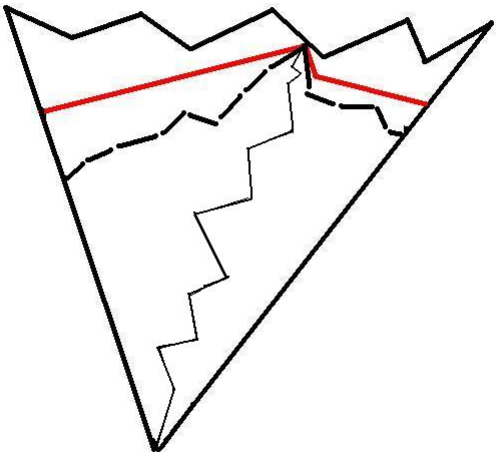


Figure 4.1: Visualization of the search graph for a shortest path problem. The root is located in the bottom, and the thin path going from the root to the edge of the top of the graph illustrates an optimal path. Nodes above the top-most line have a higher cost than the optimal path, and nodes n above the stippled line have $g(n) + h(n)$ larger than the cost of an optimal path. Uninformed search procedures need to explore the whole region between these to lines whereas informed search procedures need only explore the first successor node in this region.

In Figure 4.1 is depicted a search graph. Note that if this was the search graph for a troubleshooting problem, it should have a more diamond-like shape due to coalescing. The thin line represents an optimal path and all nodes n that lie above the thick top-most line have $g(n) > C^*$ (call this region 1). Furthermore, say the stippled line meets the optimal path in node n, then all the nodes m above the stippled line have $g(m) + h(m) > g(n) = C^*$ (call this region 2). These two regions of the search space illustrate the benefit an informed search may have over an uninformed one. Let us call the region below region 2 for region 3. Region 3 illustrates the *minimal* part of the search graph that search algorithms are forced to explore to be able to prove optimality. Furthermore, to achieve this goal, they also need to explore all the immediate successors of nodes in region 3 that lies in region 2, thereby establishing that such nodes cannot be part of a shortest path.

The first algorithm that we shall discuss is a **depth-first search** (or **branch-and-bound search**) with coalescing and pruning. The algorithm is given in Algorithm 2, and an explanation follows. Note that we use '&'

to denote that arguments to functions are passed as references. In line 5 we utilize some heuristic to get a good initial upper bound which will help the algorithm to prune unpromising paths earlier (cf. Line 21-22). In line 15 the function `IsGoalNode(·)` simply test whether any more actions can be performed; if not, then we have reached a goal node. In Line 20 we call the heuristic function $h(\cdot)$ to get an f-value for the new node m. At Line 24 we check if we have already seen the node before in which case the node is stored in the coalescing map variable map. If the new instance of the node has a worse g-value than previously, we prune it; otherwise we update the map with this new better node (Line 27). Finally, in Line 28 we make a recursive call.

Once the algorithm terminates (Line 8), we can recreate the optimal sequence by looking at the node $best$. Here we assume that each node stores the sequence that lead to the node. Alternatively, if we want to save memory, we may keep a single (global) sequence that is updated every time a new better goal node is found. The admissibility of depth-first search follows from the simple fact that it scans all relevant nodes in the search graph, and that all other nodes are pruned by a bound that is never lower than the optimal cost.

As the algorithm is written, it takes $O(\gamma(M) \cdot 2^{|\mathcal{A}|})$ memory (for the coalescing map) and $O(\gamma(M) \cdot |\mathcal{A}|!)$ time ($\gamma(M)$ being the per-node overhead for the model). It can therefore not be guaranteed to terminate within the $O(\gamma(M) \cdot 2^{|\mathcal{A}|})$ time bound suggested by Proposition 6 on page 62. (However, we shall later see how coalescing can indeed be improved to achieve this.) Alternatively we could discard the coalescing map to get an algorithm that takes up $O(\gamma(M) \cdot |\mathcal{A}|)$ memory and runs in $\Theta(\gamma(M) \cdot |\mathcal{A}|!)$ time. This is usually *not* a good idea as the memory needed for the coalescing map is relatively low compared to storing the full search graph as done by other algorithms. Despite its apparent high complexity, node expansion can usually be made rather efficient for depth-first search which together with pruning can lead to a quite competitive algorithm ((Ottosen and Vomlel, 2010a) investigates this issue, albeit in another context). In later chapters we shall discuss further pruning techniques.

Whereas depth-first search scans more nodes than strictly necessary, the **A* algorithm** uses a best-first strategy to minimize the number of node expansions. As we shall see, A* is under certain conditions *the* best algorithm in terms of node expansions. The A* algorithm for troubleshooting with actions is given in Algorithm 3. The algorithm works by continuously expanding the node with the lowest f-value (Line 9). This may be done efficiently by means a priority queue which in this context is traditionally called the **open set**, denoted $\mathcal{O}$. The nodes resident in the open set collectively make up the **frontier** (or **fringe**) of the search where nodes have been explored but not yet expanded.

Algorithm 2 Depth-first search with coalescing, upper-bound pruning and initial upper-bound. With each node n we associate the attributes $n.\boldsymbol{\epsilon}$ (the current evidence), $n.g$ (the current cost from the start node to n), and $n.f$ (the evaluation function for n). The algorithm assumes that the heuristic function $h(\cdot)$ is admissible.

```
1:  function DFSForTroubleshootingWithActions(M)
2:      Input: A troubleshooting model M = (F, A, Q, W, BN, C)
3:             with A ≠ ∅, Q = ∅, C_W = 0
4:      Let s = (g = 0, f = 0, ε = ∅)                    ▷ The start node
5:      Let best = Heuristic(s)                          ▷ Initial upper bound
6:      Let map = ∅                                      ▷ Coalescing map
7:      ExpandNode(s, best, map)
8:      return best
9:  end function
10: procedure ExpandNode(n, &best, &map)
11:     for all α ∈ A(n.ε) do                            ▷ For all remaining actions
12:         Let m = Copy(n)
13:         Set m.ε = m.ε ∪ {{α = ¬a}}                   ▷ Update evidence
14:         Set m.g = m.g + P(n.ε) · C_α                 ▷ Compute new cost
15:         if IsGoalNode(m) then
16:             if m.g < best.g then
17:                 Set best = m
18:             end if
19:         else
20:             Set m.f = m.g + h(m)                     ▷ Call heuristic function
21:             if m.f ≥ best.g then
22:                 continue                             ▷ Upper-bound pruning
23:             end if
24:             if map[m.ε].g ≤ m.g then                 ▷ Coalescing-based pruning
25:                 continue
26:             end if
27:             Set map[m.ε] = m
28:             ExpandNode(m, best, map)
29:         end if
30:     end for
31: end procedure
```

Traditionally, the algorithm keeps all relevant explored nodes in memory by adding expanded nodes (Line 13) to the so-called **closed set** which we have not shown in the algorithm. This allows for the following trade-off: when the algorithm terminates (returning a goal node, Line 10-11) we may recreate an optimal sequence as we did for depth-first search. However, since we have all the expanded nodes in memory in the closed set, we

Algorithm 3 A* search with coalescing and upper-bound pruning. With each node n we associate the attributes $n.\boldsymbol{\epsilon}$ (the current evidence), $n.g$ (the current cost from the start node to n), and $n.f$ (the evaluation function for n). The algorithm assumes that the heuristic function $h(\cdot)$ is admissible.

```
1: function AStarForTroubleshootingWithActions(M)
2:     Input: A troubleshooting model M = (F, A, Q, W, BN, C)
3:             with A ≠ ∅, Q = ∅, C_W = 0
4:     Let s = (g = 0, f = 0, ε = ∅)                    ▷ The start node
5:     Let map = ∅                                       ▷ Coalescing map
6:     Let O = {s}                                       ▷ The open set
7:     Let best = Heuristic(s)                           ▷ Initial upper bound
8:     while O ≠ ∅ do
9:         Let n = arg min_{m∈O} m.f                     ▷ Find node with lowest f-value
10:        if IsGoalNode(n) then
11:            return n
12:        end if
13:        Set O = O \ {n}
14:        for all α ∈ A(n.ε) do                         ▷ For all remaining actions
15:            Let m = Copy(n)
16:            Set m.ε = m.ε ∪ {{α = ¬a}}
17:            Set m.g = m.g + P(n.ε) · C_α
18:            if map[m.ε].g ≤ m.g then                  ▷ Coalescing-based pruning
19:                continue
20:            end if
21:            if IsGoalNode(m) then
22:                if m.g < best.g then
23:                    Set best = m
24:                end if
25:                continue
26:            end if
27:            Set m.f = m.g + h(m)                      ▷ Call heuristic function
28:            if m.f ≥ best.g then                      ▷ Upper-bound pruning
29:                continue
30:            end if
31:            Set map[m.ε] = m
32:            Set O = O ∪ {m}
33:        end for
34:    end while
35: end function
```

can simply track pointers back to the start node—this alleviates the memory requirement a little as we do not have to store the past in each node. Having a closed set also allow us to trace descendents of m that have already been expanded even though we have now found a shorter path to m (Line 18-20). In this manner we may avoid re-expanding nodes and simply update the g- and f-values. On the other hand, having a closed set leads to slightly higher memory requirements.

Lines 21-26 and 28-29 applies pruning to nodes that are not worth putting into the priority queue because a better goal node has already been found. This pruning requires an admissible heuristic function. Finally, it is in order to mention this classical result:

Theorem 3 (Hart et al. (1968))**.** *A* terminates with an optimal solution when the heuristic function $h(\cdot)$ is admissible.*

Compared to depth-first search, it is somewhat more difficult to establish that A* returns an optimal solution. A thorough theoretical exposition may be found in (Pearl, 1984). Algorithm 3 has complexity $O(\gamma(M) \cdot |\mathcal{A}|!)$ which is still not optimal according to Proposition 6 on page 62. We cannot claim that the algorithm has $O(\gamma(M) \cdot 2^{|\mathcal{A}|})$ complexity because we risk that nodes are expanded several times (or at least updated several times if we use a closed set). To get closer to this bound, we need some additional definitions and results.

Definition 22 (Consistent Heuristic Function)**.** *Let $h(\cdot)$ be a heuristic function for some search problem. If*

$$h(n) \quad \leq \quad k(n,m) + h(m) \;\; \forall n, m \tag{4.5}$$

*where m is reachable from n, then $h(\cdot)$ is **consistent**.*

Definition 23 (Monotone Heuristic Function)**.** *Let $h(\cdot)$ be a heuristic function for some search problem. If*

$$h(n) \quad \leq \quad c(n,m) + h(m) \;\; \forall n, m \,|\, m \in succ(n) \tag{4.6}$$

*then $h(\cdot)$ is **monotone**.*

The importance of consistency and monotonicity can be seen by the following (incomplete) list of theorems (Hart et al., 1968), (Pearl, 1984):

> **Property 1.** Monotonicity and consistency are equivalent properties.
>
> **Property 2.** Every consistent heuristic is also admissible.
>
> **Property 3.** A* guided by a monotone heuristic finds optimal paths to all expanded nodes, that is, $g(n) = g^*(n)$ for all n in the closed set.

Property 4. If $h(\cdot)$ is monotone, then a necessary condition for expanding node n is given by

$$g^*(n) + h(n) \quad \leq \quad C^* \tag{4.7}$$

and the sufficient condition is given by

$$g^*(n) + h(n) \quad < \quad C^* \tag{4.8}$$

Property 5. If $h(\cdot)$ is monotone, then the f-values of the sequence of nodes expanded by A* is non-decreasing.

Let us discuss a few of these properties. Property 1 and 2 implies that we can focus on proving motonicity and get all the analytical power of consistency for free. Property 4 implies that the choice of tie-breaking rules employed in the priority queue (Line 9) is largely insignificant because we need to investigate all ties anyway. The most significant property, however, is perhaps Property 3 because it means that whenever we expand a node, then we have already found an optimal path to that node. This completely removes the need to re-expand any node (or re-update the g- and f-values of any node). Therefore, if A* is guided by a monotone heuristic function, we immediately get the $O(\gamma(M) \cdot 2^{|\mathcal{A}|})$ complexity bound. The importance of monotonicity is furthermore captured by the following result.

Theorem 4 (Dechter and Pearl (1985))**.** *Let a search problem be given and let $h(\cdot)$ be a monotone heuristic function that can guide any search algorithm for this problem. Then, among all possible admissible algorithms, A* is the algorithm that leads to the fewest number of node expansions.*

It also turns out we can make any non-monotone, admissible heuristic function $h(\cdot)$ into a monotone one (Mero, 1984; Mahanti and Ray, 1987). If $h_1(n) \leq h_2(n)$ for all non-goal nodes n, then the heuristic function $h_2(\cdot)$ is said to be **at least as informed** as $h_1(\cdot)$. Korf then gives (approximately) the following formulation:

Theorem 5 (Korf (1985))**.** *For any admissible heuristic function $h(\cdot)$, we can construct a monotone heuristic function $h'(\cdot)$ which is at least as informed as $h(\cdot)$ by defining*

$$h'(m) \quad = \quad \max\left(h(m), h(n) - c(n,m)\right) \tag{4.9}$$

for all successor nodes m of n.

Thus, if at some point the f-value is about to drop due to inconsistency, we simply keep the same f-value in m as we had in the parent node n. To our knowledge this is not a widely adopted idea (at least not experimentally), even though it is briefly mentioned in (Nilsson, 1998). Finally, we may add the following result which (Pearl, 1984) cites Nilsson for:

Theorem 6. *Let A^*_1 and A^*_2 be two instances of the A^* algorithm with heuristic functions $h_1(\cdot)$ and $h_2(\cdot)$, respectively. Futhermore, let $h_1(\cdot)$ be at least as informed as $h_2(\cdot)$, and let A^*_1 and A^*_2 apply the same tie-breaking rules when selecting a node to expand. Then every node expanded by A^*_1 is also expanded by A^*_2.*

Recently, inconsistent heuristic functions have been shown to be desirable (Zahavi et al., 2007) for use with **iterative deepening A*** (Korf, 1985, 1993).

It is well-known that we may obtain a solution faster if we give up on admissibility and instead use a non-admissible heuristic function. However, we then face the problem that we have no guarantee about how good the acquired solution actually is compared to an optimal one. An interesting class of algorithms derived from A* is called **ϵ-admissible algorithms** because they guarantee that the found solution has a cost within $(1+\epsilon)$ of the cost of an optimal solution. $(1+\epsilon)$ is also called the **inflation factor**. We have the following encouraging result.

Theorem 7 (Pearl (1984))**.** *Let $h(\cdot)$ be an admissible heuristic function for some search problem. Then for all $\epsilon \geq 0$, A^* guided by*

$$h'(n) \quad = \quad (1+\epsilon) \cdot h(n) \;\; \forall n \tag{4.10}$$

finds a solution with cost $C \leq (1+\epsilon) \cdot C^*$.

Corollary 1. *Depth-first search using an non-admissible heuristic function like $h'(\cdot)$ above finds a solution with cost $C \leq (1+\epsilon) \cdot C^*$.*

Interestingly, Pearl (1984) also gives detailed information about applying a second non-admissible heuristic function during the search without sacrificing ϵ-admissibility. Yet another ϵ-admissible scheme may be found in (Pohl, 1973). The above theorem may then be used to perform a series of searches where ϵ is gradually lowered based on how much computational time we have available (Likhachev et al., 2004).

We shall end this section with a brief overview of some of the most interesting extensions for A* that have been proposed. For an overview of heuristic search we refer to (Stewart et al., 1994) and (Rios and Chaimowicz, 2010). Since the biggest problem with A* is the memory requirements, much effort has been put into lowering the size of the open and closed set. In general these schemes trade less memory for more time such that the algorithm can solve larger problem instances. Sometimes, however, the algorithms may also improve the running time.

(Stern et al., 2010) describes a method where we perform a **depth-first look-ahead search** in each node n. This look-ahead search is used to establish an additional lower-bound (f-value) for n by taking the smallest such found during the depth-first search. The open set is then ordered in terms of this new lower-bound. If the termination conditions are changed

slightly, admissibility is preserved, and space requirements are reduced exponentially in the look-ahead depth. In this case the authors also report significant faster completion times for certain problems.

The second idea is called **frontier search** (Korf et al., 2005). The basic idea is to avoid storing the closed set by exploiting consistency of the heuristic function. Their method includes novel approaches for recovering the solution path without storing the past in each node. They report large speedups for some problems.

A third idea is reduce the size of the open set by not storing all successors of a newly expanded node n (Yoshizumi et al., 2000). Instead we label all successors of n as either "promising" or "unpromising". The promising nodes are put into the open set immediately whereas n is given the minimal f-value among the unpromising nodes and put into the open set again. The unpromising nodes may then be generated again at a later time if we need to expand n again. For this reason the scheme is called **partial expansion**, and it works particularly well in problems with large branching factors where the open set is quite large compared to the closed set. Troubleshooting problems can reasonably be categorized as problems with large branching factors. However, in troubleshooting we may easily run a heuristic to find an initial upper-bound like we did in our presentation of depth-first search (Algorithm 2) and A* (Algorithm 3). Using this upper-bound for pruning can probably bring much of the savings of partial expansions.

Another approach is to limit the number of priority queue operations done on the open set (Sun et al., 2009). This is achieved by determining (during expansion) whether a newly generated node will be expanded next—if so, then the node is not put into the open set (and immediately removed) but expanded after all other newly generated nodes have been put into open.

Let us briefly discuss an idea which we have not had time to investigate empirically. It is usually assumed that a heuristic function is fast to compute. We have used the neutral factor $\gamma(M)$ to hide the real complexity of node expansion, including the cost of computing the heuristic function and updating the open/closed set. The open set usually takes $O(\lg|\mathcal{O}|)$ time to update per node, but the priority queue grows exponentially large and may overall lead to a significant run-time cost. At least for troubleshooting, it is virtually guaranteed than any good heuristic function takes at least $O(|\mathcal{A}|)$ time to compute. Therefore computing this function is at least as expensive as updating the priority queue. A simple way to reduce this overhead is to postpone the recomputation of the heuristic value until it is actually needed or until it can be deemed beneficial to do so. In particular, we are only forced to recompute the heuristic function for a node n if $g(n)$ is larger than the lower bound originally established by a predecessor of n. How much this speeds up the search (if at all) remains to be seen.

Finally we shall discuss the idea of bidirectional search (Kaindl and Kainz, 1997). The main limitation stems from the fact that a bidirectional search requires a single goal node. (In principle, however, it seems that a bidirectional search should be feasible whenever bottom-up approaches work, even though this complicates the matter somewhat.) This limitation narrows the class of troubleshooting problems where a bidirectional search is possible:

Definition 24 (Soft Troubleshooting Model)**.** *Let M be a troubleshooting model containing only actions. Then a bidirectional search is feasible if for all $\alpha \in \mathcal{A}$*

$$P(a\,|f) < 1 \;\; \forall f \in fa(\alpha) \tag{4.11}$$

*and we call M for a **soft troubleshooting model**—otherwise it is called a **non-soft troubleshooting model**.*

Such soft models have an associated goal state where the Bayesian network does not admit any evidence conflict. Traditionally, a bidirectional search uses a separate heuristic function for each direction. In case we do not have a heuristic function for the reverse direction, we may simply take $h(\cdot) \equiv 0$ as a very optimistic, but monotone heuristic function. It is an open question whether bidirectional search yields any improvement over a normal unidirectional search for troubleshooting.

4.3 Solution Methods for Models with Questions

In this section we shall briefly review algorithms that return optimal solutions for models that include questions. The main difference lies in the fact that a solution is now a tree and not just a sequence. Troubleshooting trees are a special case of **AND-OR search graphs** where (a) the chance nodes represent **AND nodes** where all successors must be part of any solution involving the node and (b) the decision nodes represents **OR nodes** where only a single successor node must be part of a solution that includes the node.

This implies that the termination criterion cannot be based on solely on a single node (cf. the `IsGoalNode(·)` function which determined if a node was solved), but must take into account if a set of explored nodes collectively make up a solution tree (troubleshooting strategy). To achieve this goal, the impact of newly generated successor nodes must be propagated to parent nodes in the (partial) solution tree. Conceptually, we have found a solution tree if the root node can be labelled "solved" by the following labelling procedure.

Algorithm 4 The helper function for depth-first search (Algorithm 5).

```
1: procedure EXPANDQUESTIONS(&n, &map)
2:     for all Q ∈ 𝒬(n.ϵ) do                          ▷ For all remaining questions
3:         Let n_q = MAKECHANCENODE(n, Q)
4:         for all q ∈ Q do
5:             Let m = COPY(n)
6:             Set m.ϵ = m.ϵ ∪ {{Q = q}}                          ▷ Update evidence
7:             if map[m.ϵ].cost < ∞ then                  ▷ Problem already solved?
8:                 Set m.cost = map[m.ϵ].cost
9:             else if n.cost ≤ h(m) then                  ▷ Upper-bound pruning
10:                continue
11:            else if P(m.ϵ | n.ϵ) > 0 then                ▷ Is sub-problem valid?
12:                EXPANDNODE(m, map)
13:            end if
14:            Set n_q.cost = n_q.cost + P(m.ϵ | n.ϵ) · m.cost
15:        end for
16:        if n.cost > n_q.cost then                  ▷ Update best successor node
17:            Set n.cost = n_q.cost
18:            Set n.best = n_q
19:        end if
20:    end for
21: end procedure
```

Definition 25 (AND/OR graph "solved" labelling rules (Pearl, 1984))**.** *A node in an AND-OR graph is solved if either*

i. it is a terminal node (representing a primitive problem),

ii. it is non-terminal OR node, and at least one of its OR links point to a solved node, or

iii. it is a non-terminal AND node, and all of its AND links point to solved nodes.

The following exposition is mainly based on (Pearl, 1984), (Vomlelová and Vomlel, 2003) and (Nilsson, 1980). We start our presentation with a depth-first search for troubleshooting with questions. As was the case for depth-first search for models with actions, the biggest practical problem lie in the fact that a newly found improved solution must be copied from the stack (alternatively, node expansion can take place on the heap, in which case swapping the root of the tree is sufficient). In Algorithm 2 on page 68 we employed coalescing to speed up the search, but when we are dealing with AND-OR graphs, the coalescing map is also essential for recovering a solution tree in a cheap way. To each decision node n we now add the additional attribute $n.best$ describing the best successor node.

Algorithm 5 Depth-first search with coalescing and upper-bound pruning for models that include questions. With each node n we associate the attributes $n.\boldsymbol{\epsilon}$ (the current evidence), $n.cost$ (the cost of the partial solution rooted at n), and $b.best$ (the best successor node). The algorithm assumes that the heuristic function $h(\cdot)$ is admissible.

```
1: function DFSForTroubleshootingWithQuestions(M)
2:     Input: A troubleshooting model M = (F, A, Q, W, BN, C)
3:             with A ≠ ∅, Q ≠ ∅, C_W = 0
4:     Let s = (cost = ∞, ε = ∅, best = null)              ▷ The start node
5:     Let map = ∅                                          ▷ Coalescing map
6:     ExpandNode(s, map)
7:     return CreateSolutionTree(s, map)
8: end function
9: procedure ExpandNode(&n, &map)
10:    if map[n.ε].cost < ∞ then
11:        Set n.cost = map[n.ε].cost
12:        Set n.best = map[n.ε].best
13:        return
14:    end if
15:    ExpandActions(n, map)
16:    ExpandQuestions(n, map)                              ▷ See Algorithm 4
17:    Set map[n.ε] = n        ▷ Make sure sub-problem is solved only once
18: end procedure
19: procedure ExpandActions(&n, &map)
20:    for all α ∈ A(n.ε) do                                ▷ For all remaining actions
21:        Let m = Copy(n)
22:        Set m.ε = m.ε ∪ {{α = ¬a}}                       ▷ Update evidence
23:        if IsTerminalNode(m) then
24:            Set m.cost = C(m.ε)                          ▷ Cost of path
25:        else if map[m.ε].cost < ∞ then                   ▷ Problem already solved?
26:            Set m.cost = map[m.ε].cost
27:        else if n.cost ≤ h(m) then                       ▷ Upper-bound pruning
28:            continue
29:        else
30:            ExpandNode(m, map)
31:        end if
32:        if n.cost > m.cost then                          ▷ Update best successor node
33:            Set n.cost = m.cost
34:            Set n.best = m
35:        end if
36:    end for
37: end procedure
```

Algorithm 6 AO* search with coalescing for models that include questions. With each node n we associate the attributes $n.\boldsymbol{\epsilon}$ (the current evidence), $n.cost$ (the cost of the partial solution rooted at n), $n.parents$ (the parents that have explored n), $n.children$ (the children found by expanding n), $n.step$ (the action or question associated with a decision node), and $n.best$ (the best successor node). Note the we assume that $n.parents, n.children$ and $n.step$ are implicitly updated as the search graph is explored. The algorithm assumes that the heuristic function $h(\cdot)$ is admissible.

function AOSTARFORTROUBLESHOOTINGWITHQUESTIONS(M)
  Input: A troubleshooting model $M = (\mathcal{F}, \mathcal{A}, \mathcal{Q}, W, BN, \mathcal{C})$
    with $\mathcal{A} \neq \emptyset$, $\mathcal{Q} \neq \emptyset$, $C_W = 0$
  Let $s = (cost = 0, \boldsymbol{\epsilon} = \emptyset, best = \textbf{null})$ ▷ The start node
  Let $map = \emptyset$ ▷ Coalescing map
  Let $\mathcal{O} = \{s\}$ ▷ The open set
  while $\mathcal{O} \neq \emptyset$ **do**
    Let $n = \arg\min_{m \in \mathcal{O}} m.cost$ ▷ Find node with lowest remaining cost
    Remark: n is a decision node
    EXPANDNODE(n, map)
    UPDATEANSCESTORS(n, map)
    Set $\mathcal{O} =$ FINDUNEXPANDEDNODES($s.best, map$)
  end while
  return CREATESOLUTIONTREE(s, map)
end function
procedure EXPANDNODE($\&n, \&map$)
  for all $\alpha \in \mathcal{A}(n.\boldsymbol{\epsilon})$ **do** ▷ For all remaining actions
    Let $m =$ COPY(n)
    Set $m.\boldsymbol{\epsilon} = m.\boldsymbol{\epsilon} \cup \{\{\alpha = \neg a\}\}$ ▷ Update evidence
    if ISTERMINALNODE(m) **then**
      Set $m.cost = C_\alpha$
    else if $map[m.\boldsymbol{\epsilon}].cost > 0$ **then**
      Set $m.cost = map[m.\boldsymbol{\epsilon}].cost$ ▷ re-use existing value
    else ▷ This is the first time we explore m
      Set $m.cost = C_\alpha + P(m.\boldsymbol{\epsilon} \,|\, n.\boldsymbol{\epsilon}) \cdot h(m)$
      Set $map[m.\boldsymbol{\epsilon}] = m$
    end if
  end for
  EXPANDQUESTIONS(n, map)
end procedure

Algorithm 7 The helper functions for AO* search (Algorithm 6).

procedure EXPANDQUESTIONS($\&n, \&map$)
  for all $Q \in \mathcal{Q}(n.\boldsymbol{\epsilon})$ **do** ▷ For all remaining questions
    Let $n_q = \text{MAKECHANCENODE}(n, Q)$
    Set $n_q.cost = C_Q$
    for all $q \in Q$ **do**
      Let $m = \text{COPY}(n)$
      Set $m.\boldsymbol{\epsilon} = m.\boldsymbol{\epsilon} \cup \{\{Q = q\}\}$ ▷ Update evidence
      if $map[m.\boldsymbol{\epsilon}].cost > 0$ **then**
        Set $m.cost = map[m.\boldsymbol{\epsilon}].cost$
      else if $P(m.\boldsymbol{\epsilon} \,|\, n.\boldsymbol{\epsilon}) = 0$ **then** ▷ Is sub-problem invalid?
        continue
      else ▷ This is the first time we explore m
        Set $m.cost = h(m)$
        Set $map[m.\boldsymbol{\epsilon}] = m$
      end if
      Set $n_q.cost = n_q.cost + P(m.\boldsymbol{\epsilon} \,|\, n.\boldsymbol{\epsilon}) \cdot m.cost$
    end for
  end for
end procedure
procedure UPDATEANCESTORS($\&n, \&map$)
  Let $queue = \emptyset$ ▷ FIFO queue
  $\text{PUSH}(queue, n)$
  repeat
    Let $m = \text{POP}(queue)$
    Let $newCost = 0$ ▷ New node estimate for m
    if $\text{ISDECISIONNODE}(m)$ **then**
      Set $m.best = \underset{m' \in m.children}{\arg\min} \; m'.cost$
      Set $newCost = m.best.cost$
    else
      Set $newCost = C_{m.step} + \sum_{m' \in m.children} P\big(m'.\boldsymbol{\epsilon} \,|\, m.\boldsymbol{\epsilon}\big) \cdot m'.cost$
    end if
    if $newCost > m.cost$ **then** ▷ Is further updating necessary?
      Set $m.cost = newCost$
      Set $map[m.\boldsymbol{\epsilon}].cost = m.cost$
      for all $m' \in m.parents$ **do** ▷ Schedule all parent nodes
        if $m' \notin queue$ **then** ▷ Avoid duplicates due to coalescing
          $\text{PUSH}(queue, m')$
        end if
      end for
    end if
  until $queue = \emptyset$
end procedure

This attribute allows us to reconstruct a solution tree once the root node has been solved as we can recursively figure out what evidence sets that we need to index the coalescing map with. The depth-first algorithm is described in Algorithm 5. As with Algorithm 2 we may improve the pruning process by finding a good initial solution tree.

The type of coalescing we perform in Algorithm 5 can actually also be performed for Algorithm 2 on page 68. In fact, it is the best form of the two as it implies that we completely avoid processing any sub-problem twice. Therefore we easily get that the complexity of the algorithm must be $O(\gamma(M) \cdot 2^{|\mathcal{A}|+\delta\cdot|\mathcal{Q}|})$. On the other hand, the type of coalescing performed in Algorithm 2 on page 68 may only be performed for action sequences. In the following, the actions of a particular sequence s shall be denoted $\mathcal{A}(s)$. We then summarize these considerations in the following result (cf. (Lín, 2011)).

Proposition 7. *Let s^* be an optimal troubleshooting strategy, and let $s = \langle \alpha_x, \ldots, \alpha_{x+k} \rangle$ be any troubleshooting sequence embedded in s^* starting after evidence $\boldsymbol{\epsilon}$ is given. Then*

1. *s must be an optimal sequence of the actions $\mathcal{A}(s)$ given evidence $\boldsymbol{\epsilon}$, and*
2. *for each $0 < i < k$, $s^{\leq i} = \langle \alpha_x, \ldots, \alpha_{x+i} \rangle$ and $s^{>i} = \langle \alpha_{x+i+1}, \ldots, \alpha_{x+k} \rangle$ must be optimal troubleshooting sequences of $\mathcal{A}(s^{\leq i})$ and $\mathcal{A}(s^{>i})$ given evidence $\boldsymbol{\epsilon}$ and $\boldsymbol{\epsilon} \cup \boldsymbol{\epsilon}^{x:x+i}$, respectively.*

Proof. For all cases, assume the opposite. Then we immediately get a contradiction to s^* be an optimal troubleshooting strategy. □

Next we shall discuss the seminal AO* algorithm (Nilsson, 1980). The algorithm is described in Algorithm 6. Before we explain the details of the algorithm, we shall give an overview of how it works. The algorithm can be viewed as consisting of the following steps:

1. At all times, the algorithm keeps track of the currently best partial strategy (the attribute $n.best$ is used for this purpose).
2. Based on the best partial strategy s, the algorithm re-computes the open set as the explored, but yet unexpanded nodes of s. The nodes in the open set are all decision nodes.
3. Then the node with the smallest f-value in open is selected for expansion (other approaches may be considered, see e.g. (Vomlelová and Vomlel, 2003) or (Pearl, 1984)).
4. After expanding a node n, the ancestors of n need to have their f-values updated, and as a result we may need to also update $m.best$ for all ancestors.

5. This process continues until the best partial strategy becomes a full strategy and the resulting open set therefore will be empty. At this point we have found an optimal strategy if we employ an admissible heuristic function.

Let us now walk through the algorithm. Like depth-first search, it uses the attribute $n.best$ and the coalescing map to recreate a solution tree (Line 14). The same procedure is also used to find unexpanded nodes of the currently best strategy (Line 12).

The procedure `ExpandNode(·)` is slightly different from depth-first search (see also Algorithm 7). The purpose is now only to (a) expand the successor nodes and compute their lower bound using the heuristic function, and (b) update the coalescing map with parent pointers (not shown) and lower bound estimates (Line 20-27). These pointers are needed for the procedure `UpdateAncestors(·)` (see Algorithm 7) which takes care of propagating the effect of the new heuristic estimates up the search graph.

`UpdateAncestors(·)` uses a queue to ensure that parent nodes are not updated before all child nodes have been updated (Line 21). In Line 25-31 we compute the new lower bound, and in this connection it might be easiest to think of the search graph as a decision tree and not a troubleshooting tree (where the action nodes have been merged with the system test). This means that the sum in Line 30 is just a single term if $m.step \in \mathcal{A}$. Then in Line 32 we stop the upwards propagation if the new lower bound is not larger. This may seem slightly unintuitive, but it is valid because the heuristic function is admissible, and so a large lower-bound is better (that is, closer to the true cost of the remaining problem) than a smaller lower-bound. If we do have a better lower-bound, we update the node and the map (Line 33-34). Finally, we schedule all parents for updating, unless they have already been put into the queue (Line 35-37) (coalescing may cause this to happen). It is important that we do not add duplicate nodes to the queue as it would be inefficient to process the same node multiple times.

From one point of view AO* is attractive because it seeks to minimize the number of node expansions. Even so, we cannot state that it runs $O(\gamma(M) \cdot 2^{|\mathcal{A}|+\delta\cdot|\mathcal{Q}|})$ time. The Achilles' heel of AO* is the procedure `UpdateAncestors(·)` which propagates node estimates towards the start node. The problem is that we may process a large portion of the expanded search graph each time. Due to coalescing, then the number of ancestors that needs updating may be far more than a number proportional to the depth of the expanded node. Since the worst case is to update the whole graph, we easily get a $O(\gamma(M) \cdot 2^{2\cdot(|\mathcal{A}|+\delta\cdot|\mathcal{Q}|)})$ complexity bound, although this is probably not a tight bound. In this context it is worth noticing that a *non-consistent* heuristic function is actually an advantage as it implies that the updating process may be stopped early. Furthermore, the complication of the implementation that a non-consistent heuristic function gives

for A* does not apply to AO*. We summarize the exhaustive search methods in the following table:

Algorithm	Complexity	Memory
Bottom-up	$O(\gamma(M) \cdot 2^{\lvert\mathcal{A}\rvert+\delta\cdot\lvert\mathcal{Q}\rvert})$	$O(2^{\lvert\mathcal{A}\rvert+\delta\cdot\lvert\mathcal{Q}\rvert})$
Depth-first search	$O(\gamma(M) \cdot 2^{\lvert\mathcal{A}\rvert+\delta\cdot\lvert\mathcal{Q}\rvert})$	$O(2^{\lvert\mathcal{A}\rvert+\delta\cdot\lvert\mathcal{Q}\rvert})$
AO*	$O(\gamma(M) \cdot 4^{\lvert\mathcal{A}\rvert+\delta\cdot\lvert\mathcal{Q}\rvert})$	$O(2^{\lvert\mathcal{A}\rvert+\delta\cdot\lvert\mathcal{Q}\rvert})$

We can add that node expansion may be faster and that memory requirements are (considerably) lower for depth-first search. Ultimately, only experiments can answer which admissible search method that performs better, and this question remains open for troubleshooting.

4.4 Summary

In this chapter we saw how the majority of troubleshooting assumptions render the problem NP-hard. In fact, many real-world models will have to deal with three-four NP-hard problems simultaneously. Other the other hand, a few isolated problems are tractable, and even though the assumptions underlying these results are quite restrictive, we shall view them as important building blocks for heuristics when the assumptions do not apply. We then moved on to extend our toolbox with algorithms that are guaranteed to find an optimal solution such that we can easily benchmark efficient heuristics or derive new any-time heuristics.

When we are faced with a model that consists exclusively of actions, we described how depth-first search and the celebrated A* search may be used to find optimal troubleshooting sequences. For both algorithms it was important to apply coalescing to preserve an exponential time and memory complexity. We gave an in-depth account of the theoretical properties of A* and reviewed numerous extensions that seek to reduce the memory requirement or speed up the algorithm (or both). We explained how A* is in fact the optimal algorithm when it comes to minimizing the number of node expansions. However, in practice we may find that the cost of node expansion is higher in A* than in depth-first search thereby leveling the field. If we were willing to give up on admissibility, we could modify the algorithms such that they find a solution which is guaranteed to be within a predetermined bound from the optimal solution.

As soon as we introduce questions into the model, we have to use more advanced algorithms. Depth-first search could relatively easily be extended to handle troubleshooting trees with chance nodes (AND-OR graphs), albeit we had to apply coalescing a little differently. Of course, this second type of coalescing applies equally well to models containing only actions. The most complicated algorithm was AO* which appears to be the

algorithm of choice if one seeks to minimize the number of node expansions. However, we were unable to get a worst-case complexity bound that matches the bound for depth-first search or dynamic programming (bottom-up) approaches. The problem was that AO* must continuously track the currently best strategy, and this may require numerous updates to already expanded nodes as new cost estimates are propagated towards the root of the troubleshooting tree. Ultimately, only implementation experience and testing can determine which algorithm that performs better.

Chapter 5

Troubleshooting with Dependent Actions

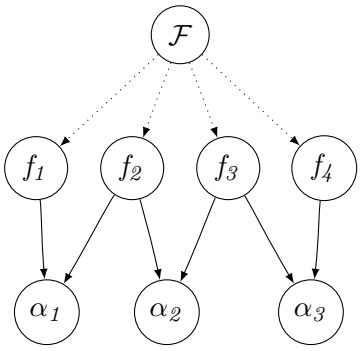
> The study of heuristics draws its inspiration from the ever-amazing observation of how much people can accomplish with that simplistic, unreliable information source known as *intuition*.
> *–Judea Pearl*

In this chapter the main topic is troubleshooting with dependent actions. Actions become dependent as soon as two actions may repair overlapping sets of faults. For example, rebooting the computer may fix mutiple problems or giving a drug to a patient may cure several diseases. In general, we can safely say that many real world troubleshooting models will contain dependent actions. We shall start with a discussion about why this topic is important, followed by a review of the classical heuristics for this problem. Then we provide theoretical insights and experimental results on using an A* search for this problem. The chapter also includes some results on dependency sets that have not been published before.

5.1 Preliminaries

When actions in a model can remedy sets of faults that overlap, we say that the model has **dependent actions**. Finding an optimal solution in models with dependent actions is of great practical importance since dependent actions can be expected to occur in many non-trivial domains.

We use the following notation. The model provides for all $\alpha \in \mathcal{A}$ and $f \in \mathcal{F}$ probabilities $P(f|\boldsymbol{\epsilon}), P(\alpha\,|\boldsymbol{\epsilon})$ and $P(\alpha\,|f)$, where $\boldsymbol{\epsilon}$ is evidence. In Figure 5.1 is shown a simple model with dependent actions. We have some **initial evidence** $\boldsymbol{\epsilon}^0$, and in the course of executing actions we collect further evidence. Recall that we write $\boldsymbol{\epsilon}^i$ to denote that the first i actions have failed ($\boldsymbol{\epsilon}^0 \subseteq \boldsymbol{\epsilon}^i$), and we have by assumption $P\big(\boldsymbol{\epsilon}^0\big) = 1$ because the device is faulty. The **presence of the fault** f is written $\mathcal{F} = f$, but we often abbreviate the event simply as f. The **set of faults that can be repaired by an action**



	f_1	f_2	f_3	f_4
$P(a_1 \mid \mathcal{F})$	1	1	0	0
$P(a_2 \mid \mathcal{F})$	0	1	1	0
$P(a_3 \mid \mathcal{F})$	0	0	1	1
$P(\mathcal{F})$	0.20	0.25	0.40	0.15
$C_{\alpha_1} = C_{\alpha_2} = C_{\alpha_3} = 1$				

Figure 5.1: Left: a simple model for a troubleshooting scenario with dependent actions. The dotted lines indicate that the faults f_1 to f_4 are states in a single fault node $\mathcal{F}$. α_1, α_2 and α_3 represent actions, and parents of an action node α are faults which may be fixed by α. Right: the quantitative part of the model.

α is denoted $fa(\alpha)$. For example, in Figure 5.1 we have $fa(\alpha_2) = \{f_2, f_3\}$. In models where actions can have $P(a \mid \boldsymbol{\epsilon}) = 1$ (**non-soft models**), $fa(\cdot)$ is a dynamic entity which we indicate by writing $fa(\cdot \mid \boldsymbol{\epsilon})$. The **set of remaining actions** is denoted $\mathcal{A}(\boldsymbol{\epsilon})$, and $\mathcal{A}(f \mid \boldsymbol{\epsilon}) \subseteq \mathcal{A}(\boldsymbol{\epsilon})$ is the set of remaining actions that can fix f.

5.2 Motivation For Action Heuristics

The remainder of this thesis is almost entirely devoted to heuristics and solutions for troubleshooting with actions only. There are at least three reasons why action heuristics and solutions are important. (we shall refer to action heuristics and solutions collectively as **action algorithms**):

1. While mixed models (containing both actions and questions) are most common in practice, we may find our selves in situations where purely action-based models are appropriate. For example, special circumstances may have rendered our ability to perform (remaining) questions impossible or irrelevant.

2. If the action algorithms are efficient and provide optimal or close-to-optimal approximations, we may use them as a solid foundation on which to build heuristics for models with questions (so-called **question heuristics**).

3. We may integrate the action algorithm into exhaustive search methods to greatly reduce the size of their search space (cf. Proposition 7 on page 79).

In most practical troubleshooting applications, however, it it often necessary to provide real-time answers to the user. Therefore the exhaustive search methods discusssed earlier quickly become unfit for such applications. Even though we have discussed how the exhaustive search methods may be converted into various heuristics, the approaches may still not be fast enough for real-time, interactive systems. Any algorithm taking more than $O(n^2)$ or $O(n^3)$ time is likely to be unsuitable for real-time applications even though troubleshooting models rarely contain more than, say, 60 actions or questions. (A seminal idea on how to "glue" several troubleshooting models together without affecting performance is given in (Skaanning and Vomlel, 2001).) To gain an understanding of how such real-time question heuristics operate, we shall briefly review the three best known algorithms (an in-depth discussion about question heuristics can also be found in (Langseth and Jensen, 2003)). They all have in common that good action sequences must be found, and in this respect we may view the algorithms more as generic *templates* operating independently of the particular action algorithm one chooses. Therefore we may also measure their complexity in terms of the number of times they need to call an action algorithm, and we shall assume that each such call takes $O(\alpha(|\mathcal{A}|))$ time. The question heuristics discussed in this section are far more focused in their search than more general search procedures because they try merely to find the best immediate decision. They have in common that they do not look very far into the future, that is, the algorithms are **myopic**.

The first approach is found in (Heckerman et al., 1995). We let $ECR\,(\mathcal{A}\,|\boldsymbol{\epsilon})$ be the **expected cost or repair for the remaining action sequence** (consisting of actions in $\mathcal{A}(\boldsymbol{\epsilon})$) found by our particular action algorithm. For each question $Q \in \mathcal{Q}$ we then define the **expected cost of repair with observation** (or **ECO**):

$$ECO\,(\mathcal{A};\, Q\,|\boldsymbol{\epsilon}) = C_Q + \sum_{q \in Q} P(Q = q\,|\boldsymbol{\epsilon}) \cdot ECR\,(\mathcal{A}\,|\boldsymbol{\epsilon} \cup \{Q = q\}) \qquad (5.1)$$

If the ECO is smaller than $ECR\,(\mathcal{A}\,|\boldsymbol{\epsilon})$ for all questions $Q \in \mathcal{Q}$, then we perform the question with the smallest ECO—otherwise we perform the best action as defined by the action algorithm. This process is continued by updating the evidence $\boldsymbol{\epsilon}$ and redoing the analysis on the remaining actions and questions. A variation would of course be to perform the best question even though not all questions have an ECO smaller than the action-based ECR. We can see that the next step can be found in $O(|\mathcal{Q}| \cdot [|\mathcal{A}| \cdot |\mathcal{F}| + \alpha(\mathcal{A})])$ time as we need to perform a partial propagation and establish an action sequence for each state of each question.

The second approach is the so-called **SACSO algorithm** (Jensen et al., 2001). The outline given here ignores some details. First, we define the **value of information** (**VOI**) for a question Q as

$$VOI(Q|\boldsymbol{\epsilon}) \quad = \quad ECR(\mathcal{A}|\boldsymbol{\epsilon}) - ECO(\mathcal{A}; Q|\boldsymbol{\epsilon}) . \tag{5.2}$$

Secondly, questions are inserted into the remaining sequence by defining a **virtual repair probability** P_Q for each question Q:

$$P_Q(\boldsymbol{\epsilon}) = \left(\max_{q \in Q} P(Q = q|\boldsymbol{\epsilon}) \right) \cdot \max_{f \in \mathcal{F}} \max_{q \in Q} \frac{P(f|\boldsymbol{\epsilon}, Q = q) - P(f|\boldsymbol{\epsilon})}{1 - P(f|\boldsymbol{\epsilon})} . \tag{5.3}$$

The aim is to treat questions almost identical to actions by finding questions that are very likely to identify a particular fault (only such questions are treated like actions). However, after we have performed an action α we usually propagate the evidence $\{\alpha = \neg a\}$ and this is not possible for questions. Instead, it is argued that the reason we perform more actions after a question must be that it failed to identify a particular fault. Therefore we should (as an approximation) insert evidence that rules out the state of the question that identified a fault. This can be done via the insertion of **likelihood evidence** (or **soft-evidence**), that rules out a certain state. These approximations allow us to find an ECR of a sequence consisting of both actions and questions (having positive and sufficiently high P_Q according to some user-defined threshold), denoted $ECR(\mathcal{A}; \mathcal{Q}|\boldsymbol{\epsilon})$. Similarly, we denote this new ECO as $ECO(\mathcal{A}; \mathcal{Q}; Q|\boldsymbol{\epsilon})$. The next step is then determined by the following procedure:

(a) Determine the set $\mathcal{Q}^{VOI}$ consisting of questions with a strictly positive $VOI(\mathcal{A}; \mathcal{Q}; Q|\boldsymbol{\epsilon})$.

(b) Let $\alpha \in \mathcal{A}$ be the best action determined by our action heuristic. Then determine the set of questions $\mathcal{Q}^{\alpha} \subseteq \mathcal{Q}^{VOI}$ for which

$$ECO(\mathcal{A}; \mathcal{Q}; Q|\boldsymbol{\epsilon}) > C_{\alpha} + P(\neg a|\boldsymbol{\epsilon}) \cdot ECO(\mathcal{A}; \mathcal{Q}; Q|\boldsymbol{\epsilon} \cup \{\neg a\}) \tag{5.4}$$

that is, find the set of questions where it appears to be beneficial to postpone the question till after the best action has been performed.

(c) If $\mathcal{Q}^{VOI} \cap \mathcal{Q}^{\alpha} = \emptyset$, then we perform α—otherwise we take the question with the highest VOI in $\mathcal{Q}^{VOI} \setminus \mathcal{Q}^{\alpha}$.

The SACSO algorithm runs in $O(|\mathcal{Q}| \cdot [(|\mathcal{A}| + |\mathcal{Q}|) \cdot |\mathcal{F}| + \alpha(\mathcal{A})])$ time as we assume that the time to find the virtual repair probabilities is included in $\alpha(\mathcal{A})$.

The last question heuristic we shall discuss is the one proposed in (Koca, 2003) and (Koca and Bilgic, 2004b). Its analysis is similar to the SACSO approach in the sense that it compares a large set of approximate sequences and greedily selects the best among them to identify the next step. However, it differs in several aspects:

(i) It computes a virtual repair probability for all questions taking into account the actions that a question indirectly affect via the associated faults. This is done via the following definition of the virtual repair probability for questions:

$$P_Q(\boldsymbol{\epsilon}) = \sum_{q \in Q} P(Q = q \,|\, \boldsymbol{\epsilon}) \cdot \left[\left(\sum_{\alpha \in \mathcal{A}(\boldsymbol{\epsilon})} \overline{P(a \,|\, \boldsymbol{\epsilon}, Q = q)} \cdot P(a \,|\, \boldsymbol{\epsilon}, Q = q) \right) - \left(\sum_{\alpha \in \mathcal{A}(\boldsymbol{\epsilon})} \overline{P(a \,|\, \boldsymbol{\epsilon})} \cdot P(a \,|\, \boldsymbol{\epsilon}) \right) \right] \tag{5.5}$$

where $\overline{P(a \,|\, \boldsymbol{\epsilon}, Q = q)}$ is the **normalized repair probability**. Therefore the formula computes the difference in average repair probability after and before a question has been asked.

(ii) They only perform a question if it improves the ECR of the remaining action sequence as well as the remaining sequence including questions.

When it comes to complexity, this more elaborate approach does not compare more sequences than the SACSO algorithm. Even so, it is clear that the action algorithm must be quite efficient if we are to get an algorithm that runs in roughly $O(n^3)$ time (n depending on $|\mathcal{A}|, |\mathcal{Q}|$, and $|\mathcal{F}|$). Therefore a major challenge is to find good approximations of (partial) strategies very efficiently. This is the main motivation for studying action heuristics or solutions in great detail.

5.3 Classical Results

The study of heuristics for dependent actions builds on the experience of troubleshooting with independent actions. We begin with the following Lemma which is valid both under the conditional costs and dependent actions assumptions:

Lemma 1 (Jensen et al. (2001))**.** *Let s be a troubleshooting sequence, and let α_x and α_{x+1} be two adjacent actions in s. If s is optimal then*

$$\begin{aligned} &C_{\alpha_x}(\boldsymbol{\epsilon}^{x-1}) + \left(1 - P\left(a_x \,|\, \boldsymbol{\epsilon}^{x-1}\right)\right) \cdot C_{\alpha_{x+1}}(\boldsymbol{\epsilon}^{x}) \leq \\ &\quad C_{\alpha_{x+1}}(\boldsymbol{\epsilon}^{x-1}) + \left(1 - P\left(a_{x+1} \,|\, \boldsymbol{\epsilon}^{x-1}\right)\right) \cdot C_{\alpha_x}(\boldsymbol{\epsilon}^{x-1}, \neg a_{a+1}) \,. \end{aligned} \tag{5.6}$$

Proof. Consider two action sequences $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ and s' where s' is equal to s execpt that actions α_x and α_{x+1} have been swapped (with $0 < x < n$). If s is optimal then we have

$$ECR\,(s) - ECR\left(s'\right) \;\; \leq \;\; 0\,.$$

Now observe that all terms except those involving α_x and α_{x+1} cancels out and we are left with

$$P\big(\boldsymbol{\epsilon}^{x-1}\big) \cdot C_{\alpha_x}(\boldsymbol{\epsilon}^{x-1}) + P(\boldsymbol{\epsilon}^{x}) \cdot C_{\alpha_{x+1}}(\boldsymbol{\epsilon}^{x}) \\ - P\big(\boldsymbol{\epsilon}^{x-1}\big) \cdot C_{\alpha_{x+1}}(\boldsymbol{\epsilon}^{x-1}) + P\big(\boldsymbol{\epsilon}^{x-1}, \neg a_{x+1}\big) \cdot C_{\alpha_x}(\boldsymbol{\epsilon}^{x}, \neg a_{x+1}) \leq 0$$

We then exploit that for all x, $P(\boldsymbol{\epsilon}^{x}) = P\big(\neg a_x \,|\boldsymbol{\epsilon}^{x-1}\big) \cdot P\big(\boldsymbol{\epsilon}^{x-1}\big)$ to get

$$P\big(\boldsymbol{\epsilon}^{x-1}\big) \cdot \Big[C_{\alpha_x}(\boldsymbol{\epsilon}^{x-1}) + P\big(\neg a_x \,|\boldsymbol{\epsilon}^{x-1}\big) \cdot C_{\alpha_{x+1}}(\boldsymbol{\epsilon}^{x}) \\ - C_{\alpha_{x+1}}(\boldsymbol{\epsilon}^{x-1}) + P\big(\neg a_{x+1} \,|\boldsymbol{\epsilon}^{x-1}\big) \cdot C_{\alpha_x}(\boldsymbol{\epsilon}^{x}, \neg a_{x+1})\Big] \leq 0$$

Since $P\big(\boldsymbol{\epsilon}^{x-1}\big) > 0$ and actions have a binary state space, we get

$$C_{\alpha_x}(\boldsymbol{\epsilon}^{x-1}) + \big(1 - P\big(a_x \,|\boldsymbol{\epsilon}^{x-1}\big)\big) \cdot C_{\alpha_{x+1}}(\boldsymbol{\epsilon}^{x}) \leq \\ C_{\alpha_{x+1}}(\boldsymbol{\epsilon}^{x-1}) + \big(1 - P\big(a_{x+1} \,|\boldsymbol{\epsilon}^{x-1}\big)\big) \cdot C_{\alpha_x}(\boldsymbol{\epsilon}^{x-1}, \neg a_{a+1})$$

as required. □

Remark 13. *Note that if we model a service call as a normal action that always fixes the problem, the above Lemma is not valid because there can be no actions that follows the the "call service" action. A discussion may be found in (Heckerman et al., 1995). In general we ignore such concerns.*

Remark 14. *We generally state results in terms of models that only contain actions. However, the results are equally valid when the sequence is embedded in a troubleshooting strategy.*

The above Lemma leads directly to the following result.

Theorem 8 (Jensen et al. (2001))**.** *Let s be a troubleshooting sequence with constant costs, and let α_x and α_{x+1} be two adjacent actions in s. If s is optimal then*

$$\frac{P\big(a_x \,|\boldsymbol{\epsilon}^{x-1}\big)}{C_{\alpha_x}} \geq \frac{P\big(a_{x+1} \,|\boldsymbol{\epsilon}^{x-1}\big)}{C_{\alpha_{x+1}}} \,. \tag{5.7}$$

This formula is the very reason that we were interested in computing virtual repair probabilities for questions. This result immediately motivates the following definitions:

Definition 26 (Efficiency of Action)**.** *Let α be an action. Then the **efficiency** of α given evidence $\boldsymbol{\epsilon}$ is define as*

$$ef(\alpha \,|\boldsymbol{\epsilon}) \;=\; \frac{P(a \,|\boldsymbol{\epsilon})}{C_\alpha} \,. \tag{5.8}$$

Definition 27 (Efficiency-Triggered Action)**.** *Assume action α_x has just been performed such that the current evidence is $\boldsymbol{\epsilon} = \{\boldsymbol{\epsilon}^{x-1}, \neg a_x\}$. Then an action $\alpha \in \mathcal{A}(\boldsymbol{\epsilon})$ for which*

$$\frac{P\big(a_x \,|\, \boldsymbol{\epsilon}^{x-1}\big)}{C_{\alpha_x}} \geq \frac{P\big(a \,|\, \boldsymbol{\epsilon}^{x-1}\big)}{C_{\alpha}} \tag{5.9}$$

*holds true is called **efficiency-triggered** by α_x given $\boldsymbol{\epsilon}^{x-1}$.*

Corollary 2. *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be an optimal troubleshooting sequence and assume we have just performed action α_x, with $0 < x < n$. Then the action that follows α_x in s must be efficiency-triggered.*

In general, there can be many efficiency-triggered actions which we may choose from without ruling out optimality. Also, the set of efficiency-triggered actions must be recomputed after each action has been performed. In turn, this implies that the above Theorem cannot in general reduce the set of potentially optimal sequences to a singleton. However, a special case appears when we consider a model containing only independent actions.

Lemma 2. *Let $\alpha, \beta \in \mathcal{A}$ be two independent actions in a model containing only actions. Then*

$$\frac{P(\alpha)}{P(\beta)} = \frac{P(\alpha \,|\, \boldsymbol{\epsilon})}{P(\beta \,|\, \boldsymbol{\epsilon})} \tag{5.10}$$

for all evidence $\boldsymbol{\epsilon}$ not involving α, β.

Proof. Consider a single-fault model with independent actions where we are updating the distribution on the fault node after receiving evidence not involving α or β. The operation correponds to lowering the probability of some states of the fault node, and then normalizing the probabilities. Since we have independent actions, the states of the fault node that have their probabilities lowered cannot intersect with $fa(\alpha)$ or $fa(\beta)$. □

The importance of the above definitions is illustrated by the following classical result, which was first brought into a troubleshooting context in (Kalagnanam and Henrion, 1990):

Theorem 9 (Kadane and Simon (1977))**.** *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be a troubleshooting sequence with independent actions. Then s is optimal if and only if*

$$\frac{P(a_i)}{C_{\alpha_i}} \geq \frac{P(a_{i+1})}{C_{\alpha_{i+1}}} \qquad \text{for } i \in \{1, \ldots, n-1\}\,. \tag{5.11}$$

Proof. Let s be optimal. Then assume the above inequalities do not hold for s. Let x be the first index where the inequality does not hold. So we must have

$$\frac{P(a_x)}{C_{\alpha_x}} < \frac{P(a_{x+1})}{C_{\alpha_{x+1}}}\,.$$

Algorithm 8 The classical P-over-C algorithm for troubleshooting with independent actions and constant cost.

```
1: function POVERC(M)
2:     Input: A troubleshooting model M = (F, A, Q, W, BN, C)
3:             with A ≠ ∅, Q = ∅, C_W = 0
4:     Propagate in the Bayesian network and compute all
5:             repair probabilities.
6:     Sort actions by descending P_α / C_α and return the sorted sequence
7: end function
```

By Lemma 2 this is equivalent to

$$\frac{P\left(a_x \,|\, \boldsymbol{\epsilon}^{x-1}\right)}{C_{\alpha_x}} < \frac{P\left(a_{x+1} \,|\, \boldsymbol{\epsilon}^{x-1}\right)}{C_{\alpha_{x+1}}} \,.$$

But then by Theorem 8 s cannot be optimal, which is a contradiction. So the inequalities must hold. □

Because of the above results, we shall henceforth abbreviate the **initial repair probability** $P(a)$ as P_α. The theorem leads directly for the so-called **P-over-C algorithm** described in Algorithm 8 for troubleshooting with independent actions and constant costs. The algorithm runs in $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time due to the comparison-based sorting. Normally, a full propagation would take $O(|\mathcal{A}| \cdot |\mathcal{F}|)$ time, but the independence of actions implies that this can be done in $O(|\mathcal{A}|)$ time.

The following results enable an easy way to compute the ECR for models with independent actions (cf. also (Heckerman et al., 1995)).

Proposition 8. *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be a troubleshooting sequence with independent actions and constant costs. Then the ECR of s may be computed as*

$$ECR\,(s) = \sum_{i=1}^{n} C_{\alpha_i} \cdot \left(1 - \sum_{j=1}^{i-1} P_{\alpha_j}\right) \,, \tag{5.12}$$

where $1 - \sum_{j=1}^{i-1} P_{\alpha_j} = P\left(\boldsymbol{\epsilon}^{i-1}\right)$.

Proof. This is basically Proposition 1, so all that is left is the evidence computation. We use induction in the evidence set $\boldsymbol{\epsilon}$. Clearly $P(\neg a) = 1 - P(a)$. Then consider the case where we are given evidence $\boldsymbol{\epsilon}$ and enlarge this set with an unperformed action α. We have

$$P(\neg a \,|\, \boldsymbol{\epsilon}) = 1 - P(a \,|\, \boldsymbol{\epsilon}) = 1 - P(\boldsymbol{\epsilon})^{-1} \cdot P_\alpha = P(\boldsymbol{\epsilon})^{-1} \cdot (P(\boldsymbol{\epsilon}) - P_\alpha)$$

Algorithm 9 The updating P-over-C algorithm for models with dependent actions

function UPDATINGPOVERC(M)
  Input: A troubleshooting model $M = (\mathcal{F}, \mathcal{A}, \mathcal{Q}, W, BN, \mathcal{C})$
    with $\mathcal{A} \neq \emptyset$, $\mathcal{Q} = \emptyset$, $C_W = 0$
  Set $s = \langle\rangle$
  for $i = 1$ to $|\mathcal{A}|$ **do**
    Propagate in the Bayesian network and compute all
      repair probabilities.
    Let $\alpha = \arg\max_{\alpha \in \mathcal{A}(\boldsymbol{\epsilon})} ef(\alpha \,|\boldsymbol{\epsilon})$
    Set $s = s + \alpha$
    Set $\boldsymbol{\epsilon} = \boldsymbol{\epsilon} \cup \{\neg a\}$
  end for
  return s
end function

where the second step follows from the single-fault assumption and because actions are independent (if there are more faults associated to a single action we can always combine them into one fault.) We then apply the induction hypothesis and get

$$P(\neg a \,|\boldsymbol{\epsilon}) \cdot P(\boldsymbol{\epsilon}) = P(\boldsymbol{\epsilon}) - P_\alpha = 1 - \sum_{\beta \in \boldsymbol{\epsilon}} P_\beta - P_\alpha$$

and the result follows. □

Remark 15. *Even though the actions must be independent in the above Proposition, they may well be non-perfect, that is, we may have $P(a \,|f) < 1$ for the faults f that an action α can address.*

For models with dependent actions we do not have such a simple formula. However, we may use the following recursive formulation to simplify computations

$$P(\neg a, \boldsymbol{\epsilon}) = P(\neg a|\boldsymbol{\epsilon}) \cdot P(\boldsymbol{\epsilon}) = (1 - P(a \,|\boldsymbol{\epsilon})) \cdot P(\boldsymbol{\epsilon}) \tag{5.13}$$

thereby taking advantage of the fact that we have already computed $P(\boldsymbol{\epsilon})$. We illustrate this by an example.

Example 8 (A greedy algorithm for dependent actions)**.** *A commonly used heuristic for dependent actions is the **updating P-over-C algorithm** described in Algorithm 9. The algorithm greedily picks the next action based on the current efficiency of actions. Now consider the model with dependent actions from Figure 5.1:*

	f_1	f_2	f_3	f_4
$P(a_1 \mid \mathcal{F})$	1	1	0	0
$P(a_2 \mid \mathcal{F})$	0	1	1	0
$P(a_3 \mid \mathcal{F})$	0	0	1	1
$P(\mathcal{F})$	0.20	0.25	0.40	0.15
$C_{\alpha_1} = C_{\alpha_2} = C_{\alpha_3} = 1$				

We have $ef(\alpha_1) = 0.45$, $ef(\alpha_3) = 0.65$, *and* $ef(\alpha_3) = 0.55$, *and so the updating P-over-C algorithm chooses to perform* α_2 *first. Then we have* $ef(\alpha_1 \mid \neg a_2) = \frac{4}{7}$ *and* $ef(\alpha_3 \mid \neg a_2) = \frac{3}{7}$, *so the next action chosen will be* α_1 *and we end up with the sequence* $\langle \alpha_2, \alpha_1, \alpha_3 \rangle$ *which has ECR*

$$\begin{aligned} ECR(\langle \alpha_2, \alpha_1, \alpha_3 \rangle) &= 1 + (1 - 0.65) \cdot 1 + (1 - 0.65) \cdot (1 - \frac{4}{7}) \cdot 1 \\ &= 1 + 0.35 + 0.35 \cdot \frac{3}{7} = 1.5 \,. \end{aligned}$$

Now consider the sequence $\langle \alpha_3, \alpha_1 \rangle$ *:*

$$ECR(\langle \alpha_3, \alpha_1 \rangle) = 1 + (1 - 0.55) \cdot 1 = 1.45$$

which shows that the greedy strategy is not optimal when dealing with dependent actions. A final remark: notice how easy it is to compute the updated repair probabilities when the model have perfect actions (like the one in this example). We simple normalize the remaining fault probabilities and take the sum of those that belong to a particular action.

More advanced heuristics for dependent actions are described in (Langseth and Jensen, 2001). The key idea is to take the value of information of performing an action into account. To approximate this VOI one may use formulas involving the entropy of normalized efficiencies or the updating P-over-C algorithm.

5.4 Heuristic Functions for Troubleshooting

In the previous chapter we gave a detailed description of various admissible search methods. They all had in common that they could take advantage of an admissible heuristic function $h(\cdot)$ which optimistically estimates the cost of solving the remaining problem for any node n in the search graph. The notion of heuristic functions was kept abstract without mentioning any concrete functions. In this section we shall therefore discuss the heuristic functions that have been suggested in the troubleshooting literature.

First we need a little notation. We aim at defining a lower bound on the ECR for every node n in a troubleshoting tree. The evidence gathered at

this node is denoted $\boldsymbol{\epsilon}^n$. A partial strategy rooted at n is denoted s^n and has expected cost $ECR\,(s^n)$. The optimal strategy rooted at n is denoted s^{n*} and we write $ECR^*(n)$ for its expected cost when we do not want to make explicit how this strategy looks like. $ECR\,(\boldsymbol{\epsilon}^n)$ is defined as the cost of the path from the root s to n. We then generalize the definition of expected cost of repair:

Definition 28 (Conditional Expected Cost of Strategy)**.** *Let n be a non-leaf node defining the root of a (possibly partial) troubleshooting strategy s^n. Assume furthermore we have received additional evidence $\boldsymbol{\eta}$ where $\boldsymbol{\eta} \cap \boldsymbol{\epsilon}^n = \emptyset$. Then the* ***conditional expected cost of repair*** *of s^n given $\boldsymbol{\eta}$ is defined as*

$$ECR\,(s^n\,|\boldsymbol{\eta}) \;=\; \sum_{\ell \in \mathcal{L}(s^n)} P\Big(\boldsymbol{\epsilon}^\ell\,|\boldsymbol{\epsilon}^n \cup \boldsymbol{\eta}\Big) \cdot \mathcal{C}(\boldsymbol{\epsilon}^\ell) \tag{5.14}$$

where

$$\mathcal{C}(\boldsymbol{\epsilon}^\ell) \;=\; \sum_{\alpha \in \boldsymbol{\epsilon}^\ell \cap \mathcal{A}} C_\alpha + \sum_{Q \in \boldsymbol{\epsilon}^\ell \cap \mathcal{Q}} C_Q\,. \tag{5.15}$$

We also write $ECR^*(n\,|\boldsymbol{\eta})$ for the optimal cost when the optimal strategy is implicit. This makes it possible to express the conditional expected cost of repair of an otherwise optimal strategy s^{n*} under additional evidence. For example, we may calculate the ECR of s^{n*} given evidence $\boldsymbol{\eta}$ and find that it is not an optimal strategy any more, that is, it may happen that $ECR\,(s^{n*}\,|\boldsymbol{\eta}) > ECR^*(n\,|\boldsymbol{\eta})$. (Vomlelová and Vomlel, 2003) have then suggested the following heuristic function for use in troubleshooting:

Definition 29. *Let n be a non-leaf node in a troubleshooting strategy. The function $\underline{ECR}(s^n)$ is defined as*

$$\underline{ECR}(s^n) \;=\; P(\boldsymbol{\epsilon}^n) \cdot \underline{ECR}_h(s^n) \tag{5.16}$$

where

$$\underline{ECR}_h(s^n) \;=\; \sum_{f \in \mathcal{F}} P(f|\boldsymbol{\epsilon}^n) \cdot ECR^*(n\,|f)\,. \tag{5.17}$$

Remark 16. *In (Vomlelová and Vomlel, 2003) the factor $P(\boldsymbol{\epsilon}^n)$ is left out. However, the factor ensures that the decomposition of the evaluation function takes the simple form*

$$f(n) \;=\; \underbrace{ECR\,(\boldsymbol{\epsilon}^n)}_{g(n)} + \underbrace{\underline{ECR}(s^n)}_{h(n)}\,.$$

Alternatively, we could have excluded the conditioning of $P(f)$ on $\boldsymbol{\epsilon}^n$ in Equation 5.17 to avoid the $P(\boldsymbol{\epsilon}^n)$ factor. However, from an implementation perspective it is easier to work with the conditional probabiity $P(f|\boldsymbol{\epsilon}^n)$ than the joint probability $P(f, \boldsymbol{\epsilon}^n)$. We can say the above formulation is most ideal for A whereas $\underline{ECR}_h(\cdot)$ is most suited for AO* (cf. the use of* `UpdateAncestors(·)`*).*

The optimal cost $ECR^*(n\,|f)$ is easy to calculate in a single-fault model: the optimal sequence is found by ordering the actions in $\mathcal{A}\,(f|\boldsymbol{\epsilon}^n)$ with respect to descending efficiency (because instantiating the fault node renders the actions independent and questions irrelevant). The next example illustrates such a computation.

Example 9 (Computing ECR^*)**.** *Let n be a non-leaf node in a troubleshooting strategy. Assume the fault f can be repaired by two actions α_1 and α_2 and that $P(a_1\,|f) = 0.9$ and $P(a_2\,|f) = 0.8$. Furthermore, let both actions have cost 1. Since instantiating the fault node renders the actions conditionally independent, $P(a|\boldsymbol{\epsilon}^n \cup f) = P(a|f)$ (for all $\boldsymbol{\epsilon}^n \in \mathcal{E}$) and the efficiencies of the two actions are 0.9 and 0.8, respectively. We get*

$$\begin{aligned} ECR^*(n\,|f) &= ECR\,(\langle\alpha_1, \alpha_2\rangle) \\ &= C_{\alpha_1} + P(\neg a_1\,|f) \cdot C_{\alpha_2} \\ &= 1 + 0.1 \cdot 1 = 1.1\,. \end{aligned}$$

Not only is $\underline{ECR}(\cdot)$ easy to compute, it also has the following property:

Theorem 10 (Vomlelová and Vomlel (2003))**.** *The heuristic function $\underline{ECR}(\cdot)$ is admissible, that is,*

$$\underline{ECR}(s^n) \;\leq\; ECR\,(s^{n*}) \tag{5.18}$$

for all non-leaf nodes n.

Proof. We have by definition

$$\begin{aligned} ECR\,(s^{n*}) &= \sum_{\ell \in \mathcal{L}(s^{n*})} P\Big(\boldsymbol{\epsilon}^\ell\,|\boldsymbol{\epsilon}^n\Big) \cdot \mathcal{C}(\boldsymbol{\epsilon}^\ell) \\ &= \sum_{\ell \in \mathcal{L}(s^{n*})} \sum_{f \in \mathcal{F}} P\Big(\boldsymbol{\epsilon}^\ell\,|\boldsymbol{\epsilon}^n \cup f\Big) \cdot P(f|\boldsymbol{\epsilon}^n) \cdot \mathcal{C}(\boldsymbol{\epsilon}^\ell) \\ &= \sum_{f \in \mathcal{F}} P(f|\boldsymbol{\epsilon}^n) \cdot \sum_{\ell \in \mathcal{L}(s^{n*})} P\Big(\boldsymbol{\epsilon}^\ell\,|\boldsymbol{\epsilon}^n \cup f\Big) \cdot \mathcal{C}(\boldsymbol{\epsilon}^\ell) \\ &= \sum_{f \in \mathcal{F}} P(f|\boldsymbol{\epsilon}^n) \cdot ECR\,(s^{n*}|f) \end{aligned}$$

and since $ECR^*(n\,|f) \leq ECR\,(s^{n*}\,|f)$ for all $f \in \mathcal{F}$, the result follows. □

The heuristic function $\underline{ECR}(\cdot)$ is valid in models that contains both questions and actions. We shall now investigate stronger property of monotonicity, but only for models without questions. As we learned in Chapter 4, this is mostly of theoretical interest. First we need to make the monotonicity condition explicit in our troubleshooting notation.

Remark 17. *For the remainder of this section, we let α_n denote the performed action on the edge from a node n to a successor node m in the search graph.*

Proposition 9. *Assume we have a troubleshooting model without questions. Let m be a successor node of a n in a troubleshooting strategy. Then monotonicity*

$$h(n) \leq c(n,m) + h(m)$$

of the heuristic function $\underline{ECR}(\cdot)$ is equivalent to

$$\underline{ECR}_h(s^n) \quad \leq \quad C_{\alpha_n} + P(\neg a_n \,|\boldsymbol{\epsilon}^n) \cdot \underline{ECR}_h(s^m)\,.$$

Proof. The A* algorithm works by continuously expanding a frontier node n for which the value of the evaluation function

$$f(n) \quad = \quad g(n) + h(n),$$

is minimal until finally a goal node t is expanded. We can rewrite this as

$$f(n) \quad = \quad \underbrace{ECR\,(\boldsymbol{\epsilon}^n)}_{g(n)} + \underbrace{P(\boldsymbol{\epsilon}^n) \cdot \underline{ECR}_h(s^n)}_{h(n)}$$

Now consider the estimate of a node m following n:

$$\begin{aligned} f(m) \quad &= \quad ECR\,(\boldsymbol{\epsilon}^m) + P(\boldsymbol{\epsilon}^m) \cdot \underline{ECR}_h(s^m) \\ &= \quad \underbrace{\underbrace{ECR\,(\boldsymbol{\epsilon}^n)}_{g(n)} + \underbrace{P(\boldsymbol{\epsilon}^n) \cdot C_{\alpha_n}}_{c(n,m)}}_{g(m)} + \underbrace{P(\boldsymbol{\epsilon}^m) \cdot \underline{ECR}_h(s^m)}_{h(m)}\,, \end{aligned}$$

where we have performed α_n to get from n to m. Therefore $c(n,m) = P(\boldsymbol{\epsilon}^n) \cdot C_{\alpha_n}$ and we have $P(\boldsymbol{\epsilon}^m) = P(\neg a_n \,|\boldsymbol{\epsilon}^n) \cdot P(\boldsymbol{\epsilon}^n)$, and so the common factor $P(\boldsymbol{\epsilon}^n)$ cancels out. □

This leads us to the following result.

Theorem 11. *Assume we have a troubleshooting model without questions. Then the heuristic function $\underline{ECR}(\cdot)$ is monotone.*

Proof. The idea is to express $\underline{ECR}_h(s^m)$ in terms of $\underline{ECR}_h(s^n)$. To do that we consider the complement of the set $fa(\alpha_n)$ which is the set of all faults that α_n cannot fix. For each $f \in \mathcal{F} \setminus fa(\alpha_n)$ Bayes' rule (conditioned) yields

$$P(f|\boldsymbol{\epsilon}^m) \quad = \quad \frac{1 \cdot P(f|\boldsymbol{\epsilon}^n)}{P(\neg a_n \,|\boldsymbol{\epsilon}^n)},$$

because $P(\neg a_n \,|\boldsymbol{\epsilon}^n \cup f) \equiv 1$. If we restrict $\underline{ECR}_h(\cdot)$ to a subset of faults $X \subseteq \mathcal{F}$, we shall abuse notation and write it

$$\underline{ECR}_h(s^n \,|X) = \sum_{f \in X} P(f|\boldsymbol{\epsilon}) \cdot ECR^*(s^n \,|f)\,.$$

In particular, we must have

$$\underline{ECR}_h(s^n) \;=\; \underline{ECR}_h(s^n\,|\mathcal{F}\setminus fa(\alpha_n)) + \underline{ECR}_h(s^n\,|fa(\alpha_n))\,. \quad (5.19)$$

We can furthermore define

$$\Delta\mathcal{F} \;=\; \underline{ECR}_h(s^m\,|\mathcal{F}\setminus fa(\alpha_n)) - \underline{ECR}_h(s^n\,|\mathcal{F}\setminus fa(\alpha_n))\,,$$

which is an extra cost because all faults in $\mathcal{F}\setminus fa(\alpha_n)$ are more likely. Similarly

$$\Delta fa(\alpha_n) = \underline{ECR}_h(s^m\,|fa(\alpha_n)) - \underline{ECR}_h(s^n\,|fa(\alpha_n))\,,$$

is the cost lost or gained because α_n has been performed and can no longer repair the faults $fa(\alpha_n)$. We can then express $\underline{ECR}_h(s^m)$ by

$$\underline{ECR}_h(s^m) = \underline{ECR}_h(s^n) + \Delta fa(\alpha_n) + \Delta\mathcal{F}, \quad (5.20)$$

The constant $ECR^*(\cdot)$ factors implies

$$\Delta\mathcal{F} = \sum_{f\in\mathcal{F}\setminus fa(\alpha_n)} [P(f|\boldsymbol{\epsilon}^m) - P(f|\boldsymbol{\epsilon}^n)]\cdot ECR^*(s^n\,|f).$$

Exploiting Bayes' rule (as explained above) and Equation 5.19 we get

$$\begin{aligned}\Delta\mathcal{F} &= \left[\frac{1}{P(\neg a_n\,|\boldsymbol{\epsilon}^n)} - 1\right]\cdot \underline{ECR}_h(s^n\,|\mathcal{F}\setminus fa(\alpha_n))\\ &= \left[\frac{1}{P(\neg a_n\,|\boldsymbol{\epsilon}^n)} - 1\right]\cdot\Big[\underline{ECR}_h(s^n) - \underline{ECR}_h(s^n\,|fa(\alpha_n))\Big].\end{aligned}$$

Inserting into Equation 5.20 yields

$$\begin{aligned}\underline{ECR}_h(s^m) &= \underline{ECR}_h(s^n) + \underline{ECR}_h(s^m\,|fa(\alpha_n)) - \underline{ECR}_h(s^n\,|fa(\alpha_n))\\ &\quad + \left[\frac{1}{P(\neg a_n\,|\boldsymbol{\epsilon}^n)} - 1\right]\cdot\Big[\underline{ECR}_h(s^n) - \underline{ECR}_h(s^n\,|fa(\alpha_n))\Big]\end{aligned}$$

$$= \frac{\underline{ECR}_h(s^n)}{P(\neg a_n\,|\boldsymbol{\epsilon}^n)} + \underline{ECR}_h(s^m\,|fa(\alpha_n)) - \frac{1}{P(\neg a_n\,|\boldsymbol{\epsilon}^n)}\cdot\underline{ECR}_h(s^n\,|fa(\alpha_n))\,,$$

and we rearrange the equation into

$$\begin{aligned}\underline{ECR}_h(s^n) &= P(\neg a_n\,|\boldsymbol{\epsilon}^n)\cdot\underline{ECR}(s^m)\\ &\quad + \underbrace{\underline{ECR}_h(s^n\,|fa(\alpha_n)) - P(\neg a_n\,|\boldsymbol{\epsilon}^n)\cdot\underline{ECR}_h(s^m\,|fa(\alpha_n))}_{\Delta}\,.\end{aligned}$$

By Proposition 9, we have to prove $\Delta \le C_{\alpha_n}$. Because of Bayes' rule and the conditional independence in the single-fault model we have

$$\begin{aligned}P(\neg a_n\,|\boldsymbol{\epsilon}^n)\cdot P(f|\boldsymbol{\epsilon}^m) &= P(\neg a_n\,|\boldsymbol{\epsilon}^n)\cdot\frac{P(\neg a_n\,|f)\cdot P(f|\boldsymbol{\epsilon}^n)}{P(\neg a_n\,|\boldsymbol{\epsilon}^n)}\\ &= P(\neg a_n\,|f)\cdot P(f|\boldsymbol{\epsilon}^n)\,.\end{aligned}$$

So we get

$$\Delta = \sum_{f \in fa(\alpha_n)} P(f|\boldsymbol{\epsilon}^n) \cdot \underbrace{[ECR^*(n\,|f) - P(\neg a_n\,|f) \cdot ECR^*(m\,|\cup f)]}_{\delta} \ .$$

Because of the single-fault assumption, we only need to prove that $\delta \leq C_{\alpha_n}$. We now index the actions in $\mathcal{A}\,(f|\boldsymbol{\epsilon}^n)$ as follows:

$$\frac{P(\beta_i\,|f)}{C_{\beta_i}} \ \geq \ \frac{P(\beta_{i+1}\,|f)}{C_{\beta_{i+1}}} \ \ \forall i.$$

In this ordering, we have $\alpha_n = \beta_x$. The inequalities generalizes to

$$C_{\beta_i} \leq \frac{P(\beta_i\,|f)}{P(\beta_j\,|f)} \cdot C_{\beta_j} \ \ \forall j > i \ . \tag{5.21}$$

In particular, this is true for $j = x$ which we shall exploit later. Assume we have N dependent actions in $\mathcal{A}\,(f|\boldsymbol{\epsilon}^n)$. The first term of δ is then

$$ECR^*(n\,|f) = ECR^*(\langle\beta_1, \ldots, \beta_N\rangle) = C_{\beta_1} + \sum_{i=2}^{N} C_{\beta_i} \cdot \prod_{j=1}^{i-1} P(\neg b_j\,|f) \ . \tag{5.22}$$

Assume that $x > 1$ (we shall deal with $x = 1$ later), then the second term of δ is

$$P(\neg a_n\,|f) \cdot ECR^*(m\,|f) \ \ = \ \ P(\neg a_n\,|f) \cdot ECR^*(\langle \ldots, \beta_{x-1}, \beta_{x+1}, \ldots\rangle)$$

$$= P(\neg a_n\,|f) \cdot \left[C_{\beta_1} + \sum_{i=2}^{x-1} C_{\beta_i} \cdot \prod_{j=1}^{i-1} P(\neg b_j\,|f) \ + \ \frac{\sum_{i=x+1}^{N} C_{\beta_i} \cdot \prod_{j=1}^{i-1} P(\neg b_j\,|f)}{P(\neg a_n\,|f)} \right] .$$

We see that the last term is also represented in Equation 5.22 and therefore cancels out. We get

$$\begin{aligned} \delta \ \ = \ \ & C_{\beta_1} \cdot [1 - P(\neg a_n\,|f)] + [1 - P(\neg a_n\,|f)] \cdot \sum_{i=2}^{x-1} C_{\beta_i} \cdot \prod_{j=1}^{i-1} P(\neg b_j\,|f) \\ & + C_{\alpha_n} \cdot \prod_{j=1}^{x-1} P(\neg b_j\,|f), \end{aligned}$$

where the last term is a leftover from Equation 5.22. Using $P(\neg a|\boldsymbol{\epsilon}) = 1 - P(a|\boldsymbol{\epsilon})$ and Equation 5.21 we get

$$
\begin{aligned}
\delta &= C_{\beta_1} \cdot P(a_n \,|f) + P(a_n \,|f) \cdot \sum_{i=2}^{x-1} C_{\beta_i} \cdot \prod_{j=1}^{i-1} P(\neg b_j \,|f) + \; C_{\alpha_n} \cdot \prod_{j=1}^{x-1} P(\neg b_j \,|f) \\
&\leq \frac{P(b_1 \,|f)}{P(a_n \,|f)} \cdot C_{\alpha_n} \cdot P(a_n \,|f) + P(a_n \,|f) \cdot \sum_{i=2}^{x-1} \frac{P(b_i \,|f)}{P(a_n \,|f)} \cdot C_{\alpha_n} \cdot \prod_{j=1}^{i-1} P(\neg b_j \,|f) \\
&\quad + C_{\alpha_n} \cdot \prod_{j=1}^{x-1} P(\neg b_j \,|f) \\
&= C_{\alpha_n} \cdot \left[P(b_1 \,|f) + \sum_{i=2}^{x-1} P(b_i \,|f) \cdot \prod_{j=1}^{i-1} P(\neg b_j \,|f) + \prod_{j=1}^{x-1} P(\neg b_j \,|f) \right] \qquad (5.23) \\
&= C_{\alpha_n} \cdot \left[1 - P(\neg b_1 \,|f) + (1 - P(\neg b_2 \,|f)) \cdot P(\neg b_1 \,|f) + \cdots + \prod_{j=1}^{x-1} P(\neg b_j \,|f) \right] \\
&= C_{\alpha_n} \cdot \left[1 - P(\neg b_2 \,|f) \cdot P(\neg b_1 \,|f) + \cdots + \prod_{j=1}^{x-1} P(\neg b_j \,|f) \right] \\
&= C_{\alpha_n} \cdot 1 \,,
\end{aligned}
$$

as required. This is not surprising if we look at the expression inside the parenthesis of Equation 5.23: the corresponding events are "β_1 fixes f, β_2 fixes f if β_1 did not fix f" etc. up to "none of the actions fixed f". These events form a sample space.

When $x = 1$, then $\delta = C_{\alpha_n} - P(\neg a_n \,|f) \cdot C_{\alpha_n}$, so in all cases $\delta \leq C_{\alpha_n}$ which completes the proof. □

The following example shows that $\underline{ECR}(\cdot)$ is not monotone when the model includes questions.

Example 10 (Questions do not imply monotonicity)**.** *Now consider a model with one question Q, two actions α_1 and α_2 and two faults f_1 and f_2 that can only be repaired by α_1 and α_2, respectively. Let $C_Q = 1$, $C_{\alpha_1} = 1$, and $C_{\alpha_2} = 10$. Let $P(a_1 \,|f_1) = P(a_2 \,|f_2) = 1$. Initially we have $P(f_1) = P(f_2) = 0.5$, and finally let $P(Q = q \,|f_1) = 1$, $P(Q = q \,|f_2) = 0$ and so $P(Q = q) = 0.5$ When $\boldsymbol{\epsilon} = \emptyset$ we have for the root node s*

$$
\underline{ECR}(s^s) \;=\; P(f_1) \cdot C_{\alpha_1} + P(f_2) \cdot C_{\alpha_2} = 0.5 \cdot 1 + 0.5 \cdot 10 = 5.5
$$

After observing $Q = q$ we get to node n^q with estimate

$$
\begin{aligned}
\underline{ECR}(s^{n^q}) &= P(Q = q) \cdot (P(f_1 \,|\, Q = q) \cdot C_{\alpha_1} + P(f_2 \,|\, Q = q) \cdot C_{\alpha_2}) \\
&= 0.5 \cdot (1 \cdot 1 + 0 \cdot 10) = 0.5
\end{aligned}
$$

and since $5.5 > 1 + 0.5$, the function cannot be monotone. Notice that the full strategy starting with observing Q has an ECR equal to 6.5.

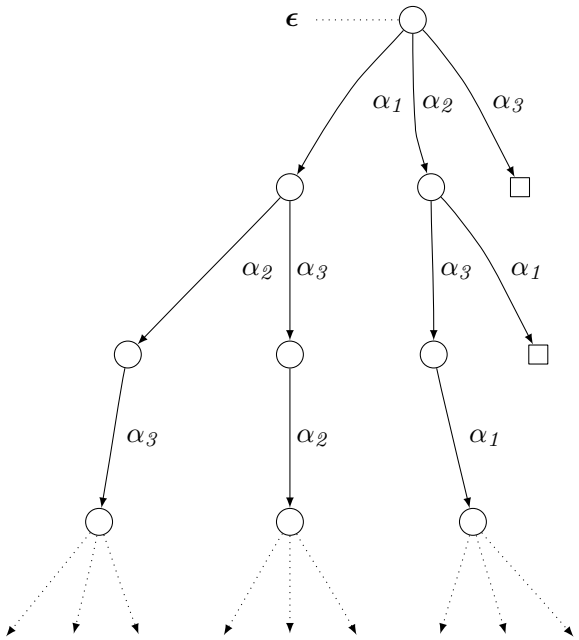


Figure 5.2: An overview of the pruning process for any subset of three actions. At the root of the subtree we have evidence $\boldsymbol{\epsilon}$ and the actions are sorted with respect to efficiency, and we have $ef(\alpha_1|\boldsymbol{\epsilon}) > ef(\alpha_2|\boldsymbol{\epsilon}) > ef(\alpha_3|\boldsymbol{\epsilon})$. Theorem 8 implies that we can prune the nodes ending in a square, and so we are left with only three possible sequences ($\langle\alpha_1, \alpha_2, \alpha_3\rangle$, $\langle\alpha_1, \alpha_3, \alpha_2\rangle$, and $\langle\alpha_2, \alpha_3, \alpha_1\rangle$). After A* has discovered the last node in these three sequences, the three paths are subject to coalescing.

If we are not interested in admissibility, we may speed up the search by transforming $\underline{ECR}(\cdot)$ into a non-admissible, but more accurate heuristic function. This is discussed in (Warnquist and Nyberg, 2008). For models with questions only, (Raghavan et al., 1999a) and (Raghavan et al., 1999c) provides several heuristic functions.

5.5 A* with Efficiency-based Pruning

We now turn our attention to some empirical results concerning the effectiveness of a pruning method based on triggered actions (cf. Theorem 8 and Definition 27). Specifically, we have implemented the A* algorithm (Algorithm 3 on page 69) using the heuristic function described above, and we additionally prune decision nodes for actions that are not triggered. We call this **efficiency-based pruning**.

In Figure 5.2 it is illustrated how the theorem can be used for pruning. If we have the order $ef(\alpha_1) > ef(\alpha_2) > ef(\alpha_3)$ at the root, we know that α_3 should never be the first action. Furthermore, after performing α_2, we know that α_1 should never be the second action. In summary, the theorem is very easy to exploit during the expansion of a node by keeping the actions sorted with respect to efficiency and by passing that information from the parent node.

Remark 18. *It might be slitghly inaccurate to re-use the term "pruning" here. Normally we may think of pruning as the process of determining that a newly generated node is not needed based on some properties of the generated node. However, for "efficiency-based pruning" we do not actually need to generate a node to figure out that it is superfluous.*

Apart from efficiency-based pruning our implementation also perform upper-bound pruning on newly *generated* nodes. For depth-first search it is customary to perform an initial search using some greedy heuristic such that paths that are far from being optimal can be pruned early. For A* this is usually not used because A* does not expand any unnecessary nodes. However, A* may very well *explore* (and store) many nodes that are not optimal. Therefore, if we have a good initial upper bound $\overline{ECR}$ of ECR^*, we can immediately throw such nodes away (Like in depth-first search we prune any node n if $f(n) \geq \overline{ECR}$) . This can be important because the frontier is usually quite large compared to the set of expanded nodes. Of course, whenever we find a goal node with lower ECR than $\overline{ECR}$ we update $\overline{ECR}$ to this new value.

Whether such a close bound $\overline{ECR}$ is easy to obtain depends somewhat on the model in question, but by choosing a close bound, we can ensure a conservative comparison of the effect of efficiency-based pruning (because using an upper-bound will speed up the search). In our case we are able to calculate this upper bound by using the updating P-over-C algorithm (Algorithm 9 on page 91). In practice this gave us bounds that came within one percent from ECR^*.

Remark 19. *The fact that the naive greedy algorithm comes quite close to the optimal value is an important observation in its own right. It strongly suggests that it is possible to prove that a polynomial algorithm can return a solution with strong guarantees on the distance to the optimal ECR. This observation is the first benefit we gain from solution methods.*

For the experiments we have generated a number of random models with an increasing number of actions. For a certain size of $|\mathcal{A}|$ we take $|\mathcal{F}| = |\mathcal{A}|$ and pick the associated faults $fa(\alpha)$ for an action at random with some dispersion on the number of associated faults. We then measure the degree of dependency between the actions as the average size of $fa(\alpha)$ over

Table 5.1: Experimental results for A* with efficiency-based pruning. Time is measures in seconds, the "Pruned" column indicates the number of pruned nodes, and the "Tree" column is the size of the search tree. A bar ("-") indicates that the models could not be solved due to memory requirements. "Relative Time" is the time relative to the fastest method (see Table 5.3).

Method	***A* + EP***			
Model / Average Dep.	Time	Pruned	Tree	Relative Time
15 / 2.2	0.58	69k	17k	1.41
16 / 2.5	1.89	189k	45k	1.24
17 / 2.88	3.77	267k	76k	3.04
18 / 2.22	19.19	2735k	467k	2.64
19 / 2.32	17.97	1614k	361k	1.88
20 / 2.8	17.94	1219k	294k	2.26
21 / 2.76	47.23	4768k	969k	2.06
22 / 2.82	-	-	-	-
23 / 3.22	-	-	-	-
24 / 3.42	-	-	-	-

all actions. Probabilities and costs are also generated randomly. In this manner we generated 10 models which we have used for comparing three versions of A* :

(a) A* with efficiency-based pruning (A* + EP)

(b) A* with coalescing and bounding (A* + C + B)

(c) A* with coalescing, bounding, and efficiency-based pruning (A* + EP + C + B).

The results are shown in Table 5.1 to 5.3. We can see that the models have between 15 and 24 actions and that the average dependency lies between 2.2 and 3.42.

Remark 20. *All the experiments reported in this thesis were carried out on an 32-bit 2.13 GHz Intel Pentium Centrino with 2 GB of memory. Furthermore, the 35,000 lines of code needed to do the experiments of this thesis were written in C++.*

In the first test (Table 5.1) we see that A* with efficiency-based pruning can solve models with up to 21 actions. Larger models remain unsolvable due to memory requirements. The running time compared to the best method (method (c)) seems to be somewhere between 2 and 3 times

Table 5.2: Experimental results for A* with coalescing and upper-bound. Time is measures in seconds, the "Pruned" column indicates the number of pruned nodes, and the "Tree" column is the size of the search tree. "Relative Time" is the time relative to the fastest method (see Table 5.3).

Method	***A** + *C* + *B***			
Model / Average Dep.	Time	Pruned	Tree	Relative Time
15 / 2.2	1.44	89k	19k	3.51
16 / 2.5	3.41	188k	44k	2.23
17 / 2.88	4.16	246k	52K	3.35
18 / 2.22	31.66	1616k	438k	4.35
19 / 2.32	30.27	1437k	295k	3.17
20 / 2.8	28.94	1291k	261k	3.64
21 / 2.76	77.17	3474k	650k	3.37
22 / 2.82	246.2	9931k	1824k	3.56
23 / 3.22	367.78	13485k	2387k	3.51
24 / 3.42	646.28	20533k	4025k	6.73

as high on average. The number of nodes that are pruned is large compared to the size of the search graph when an optimal solution have been found, but this is not so surprising when coalescing is not done. When we ran A* without pruning, it was about 30-40 times slower, and so viewed in isolation, efficiency-based pruning is a big win.

However, when we compare with method (b) (Table 5.2), we see that these two enhancements can handle larger models. The expense is that the running time is increased. It may seem surprising that the running time increases so much, since it should be relatively cheap to lookup a node in a coalescing map. The reason is that before we can utilize a lookup in the map, we have to generate the node (and compute its f-value) which is somewhat expensive. In general, method (b) tells us that efficiency-based pruning cannot stand alone and that coalescing is a more fundamental property.

Table 5.3 describes the results of method (c) which combines the enhancements of the two first methods. In general it is about 3 to 4 times as fast as method (b), and at the same time method 3 uses the least memory. Since the search graph is somewhat smaller than for method (b), then we could have expected that the number of pruned nodes should increase. However, the reason is simply that efficiency-based pruning is quite effective in pruning nodes early, which in turn leaves fewer pruning opportunities later in the search.

A natural question to ask is if Theorem 8 can be extended to make efficiency-based pruning even better? We could consider three adjacent actions instead of two, which would then lead to 5 inequalities instead of 1.

Table 5.3: Experimental results for A* with coalescing, upper-bound and efficiency-based pruning. Time is measures in seconds, the "Pruned" column indicates the number of pruned nodes, and the "Tree" column is the size of the search tree.

Method	***A** + *EP* + *C* + *B***		
Model / Average Dep.	Time	Pruned	Tree
15 / 2.2	0.41	63k	8k
16 / 2.5	1.53	131k	17k
17 / 2.88	1.24	168k	19k
18 / 2.22	7.28	1077k	128k
19 / 2.32	9.55	989k	109k
20 / 2.8	7.95	897k	87k
21 / 2.76	22.88	2490k	248k
22 / 2.82	69.17	7759k	708k
23 / 3.22	104.72	10343k	969k
24 / 3.42	96.02	12437k	977k

To get an overview of these inequalities, we have collected them in Figure 5.3. If we consider an arbitrary subset of three actions α_1, α_2, and α_3, we would normally need to compare six different sequences. However, if we have calculated the efficiencies of the three actions at the local root node with evidence ϵ, Theorem 8 leaves us with only three possible candidates. After the three sequences are expanded, the paths are coalesced into a single node in the search graph (See Figure 5.4).

Now imagine that A* is about to expand α_3 in the sequence $\langle \alpha_1, \alpha_2, \alpha_3 \rangle$. We determine if the current node expansion is optimal by comparing it with the ECR of the sequence $\langle \alpha_2, \alpha_3, \alpha_1 \rangle$. (There is no need for comparing $\langle \alpha_1, \alpha_2, \alpha_3 \rangle$ with $\langle \alpha_1, \alpha_3, \alpha_2 \rangle$ since Theorem 8 has pruned the latter.) If we expand the sequence $\langle \alpha_2, \alpha_3, \alpha_1 \rangle$ first, the analysis is similar and we compare with the best of the two other sequences (again, the best sequence is found by applying Theorem 8). There is no way to avoid calculating the full ECR of both sequences, and we have to traverse the search graph down to the local root and up to the first node of the second path. Furthermore, this traversal means that we have to store child pointers in all nodes, and we also need to keep all expanded nodes in memory. This more than doubles the memory requirement.

1. $ECR_{xyz} \leq ECR_{xzy} : \dfrac{P(a_z \,|\neg a_x)}{C_z} \leq \dfrac{P(a_y \,|\neg a_x)}{C_y}$

2. $ECR_{xyz} \leq ECR_{yxz} : \dfrac{P(a_y)}{C_y} \leq \dfrac{P(a_x)}{C_x}$

3. $ECR_{xyz} \leq ECR_{yzx}$:
   $C_x \cdot (1 - P_{y|x} \cdot P_x) - C_z \cdot P(a_x \,|\neg a_y) \cdot P_y \leq C_y \cdot P(a_x)$

4. $ECR_{xyz} \leq ECR_{zyx}$:
   $C_x \cdot (1 - P(a_z \,|\neg a_y) P_y) + C_y \cdot (P_x - P_z) \leq C_z \cdot (1 - P_{y|x} \cdot P_x)$

5. $ECR_{xyz} \leq ECR_{zxy}$:
   $C_x \cdot P(a_z) + C_y \cdot P(a_z \,|\neg a_x) \cdot P_x \leq C_z \cdot (1 - P_{x|y} \cdot P_y)$

Figure 5.3: The five inequalities arising from the six permutations of three actions α_x, α_y, and α_z. $P_{x|y}$ is shorthand for $P(\neg a_x \,|\neg a_y)$.

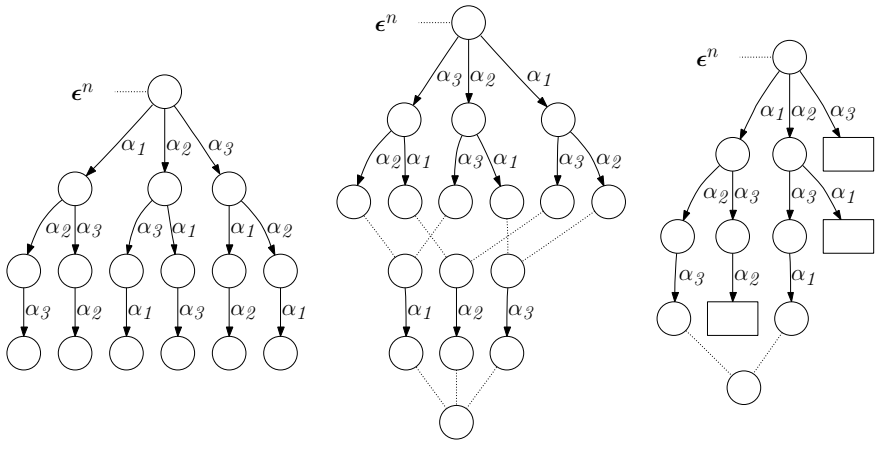


Figure 5.4: The effect of coalescing and efficiency-based pruning in A* on a subset of three actions $\{\alpha_1, \alpha_2, \alpha_3\}$. Left: the search graph without coalescing. Middle: the search graph with coalescing. Right: the search graph when applying both pruning and coalescing.

In conclusion the method turns out to slow down A* . In a model with 19 action and 19 faults, Theorem 8 pruned 989k nodes whereas the pruning over three sebsequent actions prevented a mere 3556 expansions out of about 41k possible. Theorem 11 also explains why the effect of the pruning is so small: if A* is guided by a monotone heuristic function and expands a node, then the optimal path to that node has been found. This means that the sub-trees outlined with dotted edges in Figure 5.2 are never explored unless we really need to explore them. Should we discover a non-optimal sequence first, that path is not explored further until coalescing happens.

5.6 A* with Dependency Sets

In this section we shall abandon admissibility and discuss heuristics based on the notion of dependency sets. Informally, a **dependency set** is a subset of $\mathcal{A}$ where we can alter the efficiency ordering among actions by performing an action in this set. On the other hand, the relative ordering of efficiencies of actions outside the dependency set is not alterred by performing an action in the dependency set. This leads us to the following definitions:

Definition 30. *A **dependency graph** for a troubleshooting model given evidence $\boldsymbol{\epsilon}$ is the undirected graph with a vertex for each action $\alpha \in \mathcal{A}(\boldsymbol{\epsilon})$ and an edge between two vertices α_1 and α_2 if $fa(\alpha_1 \,|\boldsymbol{\epsilon}) \cap fa(\alpha_2 \,|\boldsymbol{\epsilon}) \neq \emptyset$.*

Definition 31. *A **dependency set** for a troubleshooting model given evidence $\boldsymbol{\epsilon}$ is a connectivity component in the dependency graph given $\boldsymbol{\epsilon}$.*

When we look at each dependency set in isolation, such a set induces a troubleshooting model by considering the actions and associated faults of the dependency set. Therefore we may talk about optimal sequences of such induced models.

Definition 32. *A **dependency set leader** for a troubleshooting model given evidence $\boldsymbol{\epsilon}$ is the first action of an optimal sequence in the model induced by a dependency set given $\boldsymbol{\epsilon}$.*

Dependency sets are important because it is believed that the order of actions in the same dependency set does not change when actions outside the set are performed (Koca, 2003). This intuitive property has been used to conjecture that the following divide-and-conquer algorithm preserves admissibility:

Conjecture 1 (Koca and Bilgic (2004a)). *Suppose we are able to calculate the dependency set leaders. Then the globally optimal sequence is given by the following algorithm:*

1. *Construct the dependency sets and retrieve the set leaders.*
2. *Calculate $ef(\cdot|\boldsymbol{\epsilon})$ for all set leaders.*
3. *Select the set leader with the highest $ef(\cdot|\boldsymbol{\epsilon})$ and perform it.*
4. *If it fails, update the probabilities, and continue in step (2).*

Since it takes exponential time to find the set leaders in each dependency set, a model that can be partioned into just a few dependency sets may be much easier to solve than the initial model. Unfortunately, the following example illustrates that the conjecture is not true.

Example 11 (Dependency sets break admissibility). *We consider the following model with four actions and five faults:*

	f_1	f_2	f_3	f_4	f_5
$P(a_1\|\mathcal{F})$	1	1	0	0	0
$P(a_2\|\mathcal{F})$	0	1	1	0	0
$P(a_3\|\mathcal{F})$	0	0	1	1	0
$P(b\|\mathcal{F})$	0	0	0	0	1
$P(\mathcal{F})$	0.10	0.15	0.15	0.1	0.5
$C_{\alpha_1} = C_{\alpha_2} = C_{\alpha_3} = 1$, $C_\beta = 2$					

This model has two dependency sets, one with three actions $\{\alpha_1, \alpha_2, \alpha_3\}$ and one with one action $\{\beta\}$. We can see that all actions are perfect. Initially we have $P_{\alpha_1} = 0.25, P_{\alpha_2} = 0.30, P_{\alpha_3} = 0.25$, and $P_\beta = 0.5$, and the efficiencies are $ef(\alpha_1) = ef(\alpha_3) = ef(\beta) = 0.25$ and $ef(\alpha_2) = 0.30$.

The conjecture then states that we must find the best sequence in each dependency set. For the set consisting of β, this is trivial, and for the dependency set with the α's we have six possibilities. The only serious contenders is the greedy $s^1 = \langle\alpha_2, \alpha_1, \alpha_3\rangle$ and the non-greedy $s^2 = \langle\alpha_1, \alpha_3\rangle$. In both cases, the order of α_1 and α_3 is immaterial as they have the same efficiency. We get (by using Equation 5.13)

$$\begin{aligned} ECR(\langle\alpha_2, \alpha_1, \alpha_3\rangle) &= 1 + 0.70 \cdot 1 + 0.6 \cdot 1 = 2.3 \\ ECR(\langle\alpha_1, \alpha_3\rangle) &= 1 + 0.75 \cdot 1 = 1.75 \end{aligned}$$

The greedy algorithm is thereby not optimal when considering this dependency set in isolation. But we shall see that it becomes the best way to perform the actions in when combined with the action β. The conjecture states that we have to compare the two most efficient set leaders. We look at the merging with s^2: α_1,α_3 and β

are all equally efficient and hence all possible (final) sequences will have the same ECR. For example,

$$ECR\left(\langle \alpha_1, \beta, \alpha_3 \rangle\right) \quad = \quad 1 + 0.75 \cdot 2 + 0.25 \cdot 1 = 2.75$$

Now, if we merge with s^1 instead, then α_2 is most efficient and hereafter β is the most efficient; finally α_1 and α_3 have the same efficiency. Thereby we get

$$ECR\left(\langle \alpha_2, \beta, \alpha_1, \alpha_3 \rangle\right) \quad = \quad 1 + 0.70 \cdot 2 + 0.20 \cdot 1 + 0.10 \cdot 1 = 2.7$$

Hence the conjecture is false. One may also ask if the conjecture holds if the costs are all the same. It turns out that the conjecture is still false in this case. If we split the action β into β_1 and β_2 where $ef(\beta_1) = ef(\beta_2) = ef(\beta) = 0.25$, then all actions have the same cost (for this to work, we should also split the fault f_5 into two faults). We then have three dependency sets, and we get

$$ECR\left(\langle \alpha_1, \beta_1, \beta_2, \alpha_3 \rangle\right) = 1 + 0.75 \cdot 1 + 0.50 \cdot 1 + 0.25 \cdot 1 = 2.50$$
$$ECR\left(\langle \alpha_2, \beta_1, \beta_2, \alpha_1, \alpha_3 \rangle\right) = 1 + 0.70 \cdot 1 + 0.45 \cdot 1 + 0.2 \cdot 1 + 0.1 \cdot 1 = 2.45$$

and so there is little hope that the conjecture holds for any special case of dependency sets.

Now, since the conjecture is false, then we cannot apply the algorithm and keep admissibility; however, as we shall see, it appears that it can often be a good heuristic to use. We still believe it might be possible to find a sufficiently narrow definition of dependency sets where admissibility is preserved. A starting point could be to investigate **nested dependency sets** where for any pair of actions α, β in a dependency set, we have either

$$fa(\alpha) \subset fa(\beta) \quad \text{or} \quad fa(\beta) \subset fa(\alpha) \tag{5.24}$$

but not both. We shall, however, not follow this line of thought further.

Instead we shall use the notion of dependency sets to simplify the search space of A* . We call this resulting algorithm for the **hybrid approach**. Figure 5.5 shows an example of the search tree explored by A* in our hybrid approach. Near the root node, A* is often forced to create a branch for each successor node. However, as we get closer to the goal nodes, branching is more likely to be avoided. The branching can be avoided because we assume the depedency set conjecture is true.

The hybrid approach then simply works by finding an optimal sequence in dependency sets of a fairly small size. For this work we have restricted us to sets of a size smaller than four (later we shall discuss an extension). At any point before expanding a node, if the most efficient action belongs to a dependency set of such a small size, we find the first action in that dependency set. If the dependency set consists of one or two actions, this calculation is trivial. If the dependency set has three actions, we find the

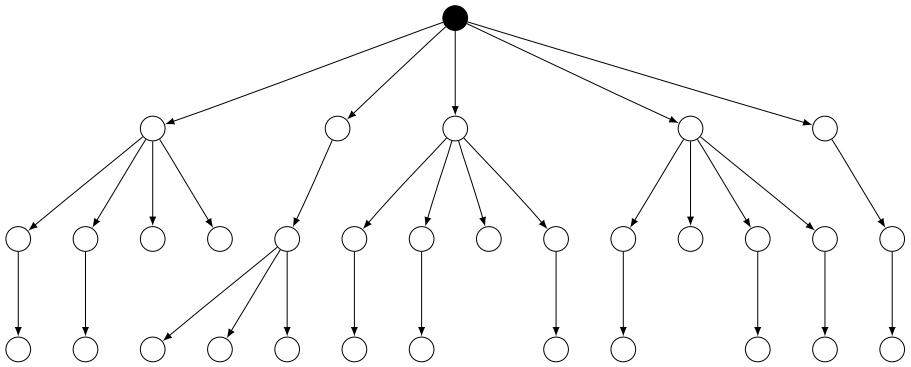


Figure 5.5: An example of what the search tree looks like in the hybrid approach. For some nodes, the normal A* branching is avoided, and near goal nodes this branching is almost avoided for all nodes. We can see that it might happen that the algorithm has to investigate all successors of a node even though the path down to that node was explored without branching.

first by comparing the three candidate sequences as we discussed in Section 5.5. Otherwise we simply expand the node as usual by generating all successors.

The results are shown in Figure 5.6. We can see that the hybrid approach is somewhat slower for models with an average dependency between 2 and 3. This is because the hybrid approach spends time investigating the size of the dependency set of the most efficient action, but it rarely gets to exploit the benefits of a small dependency set. For an average dependency between 2.1 and 1.6 the hybrid approach becomes superior, and below 1.6 it becomes very fast. It should be noted that the hybrid approach only failed to find an optimal solution one or two times out of the 21 models, and that it is very close to the optimal value even in those cases.

The plot also shows a clear trend: A* seems to perform better on models with larger dependency among actions. We belive there can be two explanations for this:

(i) The heuristic function becomes increasingly more accurate as the dependency increases.

(ii) The entropy of action efficiencies decreases (perhaps by accident) as the dependency increases, making the distribution over all possible ECR-values wider. In turn, a model with a wider distribution over ECR-values is easier to solve for A* as it becomes easier to reject unpromising paths.

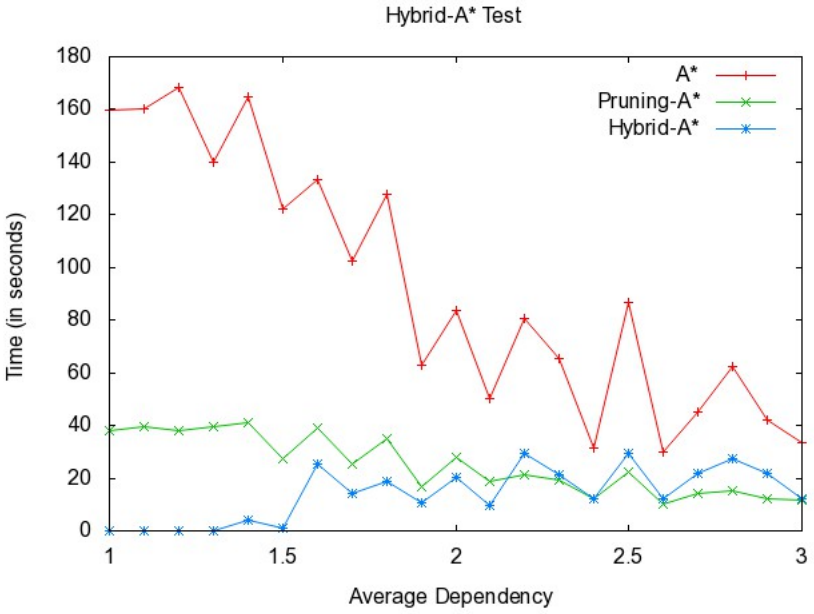


Figure 5.6: Results for the hybrid approach in models with 20 actions and 20 faults. The average dependency ranges between 3 and 1, and each action is usually associated with 1 to 3 faults. The time is measured in seconds. "A* " is A* with coalescing, "pruning-A* " is "A* " plus efficiency-based pruning and "hybrid-A* " is the hybrid approach based on "pruning-A* ". For an average dependency below 2.2 we see that the hybrid method wins.

Remark 21. *At the time this work was carried out, we were unaware that the dependency set conjecture was false. The few discrepancies that were experienced during the tests were attributed to floating point inaccuracies. For example, floating point comparisons are inherently error prone on the Intel architecture due to the fact that floating point registers use 80 bits precision whereas a* `double` *is usually 64 bits (Monniaux, 2008). However, we now know that these discrepancies constituted counter examples.*

Even though the hybrid A* approach seems very promising, the results are limiting in the sense that it only solves small dependency sets. We still need to determine how large a dependency set that it pays off to solve. One may expect that it will be most beneficial to solve small dependency sets by brute-force whereas dependency sets of medium size can be solved by a recursive call to hybrid-A* . To shed some more light on this issue, we report the results obtained by a DAT3-group which was under our supervision (Cohen et al., 2008). Their results may be viewed in Figure 5.7.

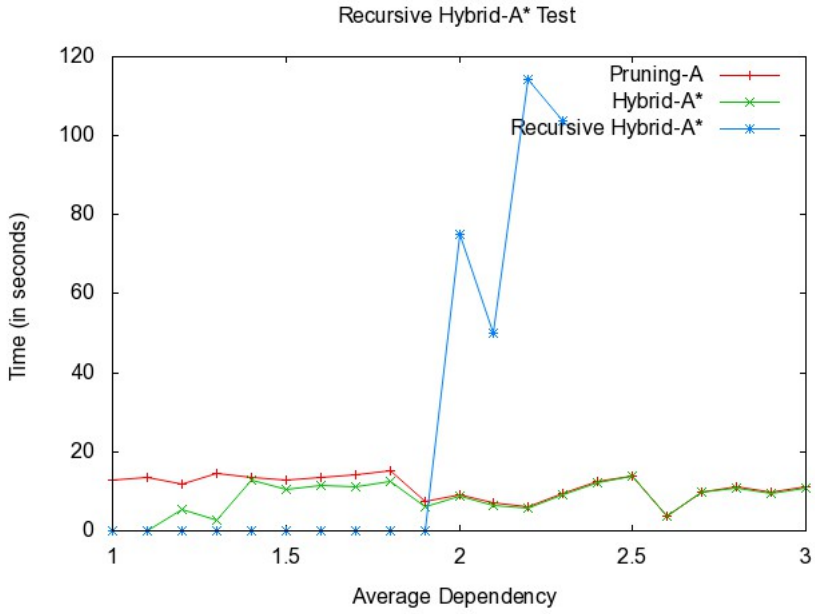


Figure 5.7: Results for the recursive hybrid approach in models with 20 actions and 20 faults (Cohen et al., 2008). The average dependency ranges between 3 and 1, and each action is usually associated with 1 to 3 faults. Time is measured in seconds. "pruning-A* " is "A* " plus efficiency-based pruning, "hybrid-A* " is the hybrid approach based on "pruning-A* ", and "recursive-hybrid-A* " recursively solves all dependency sets. For an average dependency below 2.2 we see that the hybrid method wins.

The figure compares A* with efficiency-based pruning with hybrid A* and a fully recursive hybrid-A* . First notice that the results may differ because the implementation and hardware platform is different. Nevertheless, the results clearly show that dependency sets can be solved very efficiently by the recursive approach for models with an average dependency below 2. As always, its not completely easy to extrapolate this benchmark to models of larger sizes, that is, it may be that for models with (say) 40 actions, only a smaller average dependency can be handled efficiently.

The results also show that recursive calculations become very slow at some point. This is arguably because many sub-problems can end up being solved multiple times. By carefully sharing the coalescing map between recursive invocations, this can probably be avoided. Their experiments also indicated that it was beneficial to solve as large dependency sets as possible. In any case, it seems that giving up on admissibility helps, but that models with a high degree of dependency are still difficult to approximate via the notion of dependency sets.

5.7 Summary

In this chapter we studied solutions and heuristics for models with actions only under the assumption of independent and dependent actions. We argued that the main motivation for looking at actions in isolation is that virtually all real-time question heuristics build on such action-based algorithms. At the same time we reviewed the question heuristics proposed in the literature.

We then gave theoretical reasons for defining the efficiency of actions and described common heuristics like the P-over-C algorithm and variants of it. In doing so, we also explained the classical result that the P-over-C algorithm is optimal for models with independent actions.

A central topic was the heuristic function proposed by (Vomlelová and Vomlel, 2003) which fulfill interesting theoretical properties. This heuristic function was then used in empirical evaluations of the A^* algorithm. The first results were concerned with the benefit of efficiency-based pruning which appears to work well in isolation for asymmetrical decision problems like troubleshooting, albeit the same asymmetry allows us to apply coalescing. Together with coalescing, the pruning is less impressive, and we analysed the reasons for this.

The final topic was troubleshooting with dependency sets. On the theoretical side we unfortunately found that the conjecture by (Koca and Bilgic, 2004a) was not true. However, the experiments revealed that it is nevertheless a very good approximation, often preserving admissibility. Thus, by exploiting this approximation we were able to solve many models with a low-to-medium degree of dependency among the actions almost instantly. In general the experiments left the impression that models with dependent actions are easy to approximate well and therefore we believe it should be possible to prove good worst-case quality bounds on a polynomial algorithm. Of course, this is only speculation and it remains an important open problem to verify this conjecture.

Chapter 6

Troubleshooting With Cost Clusters

> To iterate is human, to recurse divine.
> *–L. Peter Deutsch*

In this chapter we shall start our investigation of troubleshooting with conditional costs. Even though the introduction of conditional costs in general leads to an intractable problem, our aim is to investigate a simplified, but highly relevant, special scenario. The scenario is characterized by the following assumptions: (a) there are no questions, (b) all actions are independent, (c) the set of actions $\mathcal{A}$ can be partitioned into a set of clusters where where each cluster must first be opened to perform actions in the cluster, and (d) the system test may be performed for free at any time, regardless of what clusters might be open. This scenario is interesting because it models real-life situations where we need to take apart and assemble the equipment during the troubleshooting process. For example, if we are repairing a car that cannot start, we may open the hood of the car to exchange the spark plugs and test if the car is working without closing the hood; if exchanging the spark plugs did not fix the car, we can then look at the battery without paying the cost of opening the hood once again. Similar examples can be for troubleshooting a jet-engine or some other complicated piece of industrial equipment. We start with a short overview of previous research on this topic, and then we give a full formal treatment of the subject.

6.1 Related Work

Let us briefly summarize other approaches to conditional costs that resemble the idea of cost-clusters. We review the literature in chronological order.

The cost cluster framework originates from (Langseth and Jensen, 2001). They consider simple cost-cluster models where a nested cluster is allowed on the top-level cluster, but no deeper layered clusters are allowed. We call this a **flat cost-cluster model**, and an example is given in Figure 6.1. The

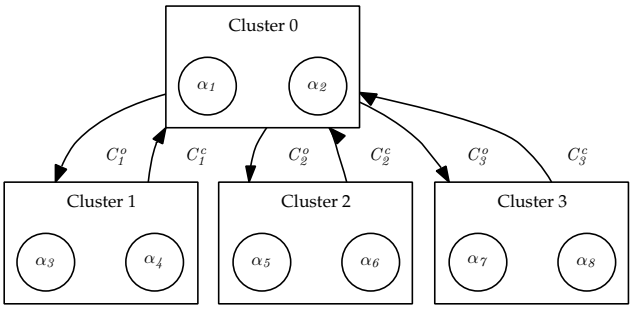


Figure 6.1: Example of the flat cost cluster model. To open a cluster $\mathcal{K}_i$ we pay the cost C_i^o, and to close a cluster we pay the cost C_i^c.

idea is that in order to access an action in a bottom level cluster $\mathcal{K}_i$, you need to pay an additional cost C_i^o and to close the cluster, you have to pay an additional cost C_i^c. Thereby it is possible to model the repair of complex man-made devices where you need to take apart some of the equipment to perform certain actions.

They are the first to introduce the following ideas: (a) that multiple actions might need to be performed before leaving a cluster, (b) that the cost to enter and leave a cluster should be modelled separately (we shall later see that this level of model detail is not needed), and (c) that a model can be **with inside information** or **without inside information** meaning that the system can be tested when a a cluster is open or that it has to be closed before testing the system, respectively. Of course, we can have models that mix clusters with and without inside information, but we shall not consider this situation. They furthermore describe heuristics for both types of problems. Later in this chapter we present a proof of correctness of their algorithm for the problem with inside information. We furthermore extend the model to a tree of clusters and give an $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time algorithm that is proved optimal. A model without inside information is closely related to troubleshooting with post-poned system test (the topic of Chapter 7) and this domain has recently been proven NP-hard (Lín, 2011).

Almost simultaneously, (Skaanning and Vomlel, 2001) discusses the use of troubleshooting with simultaneous models. The idea is to introduce special virtual "submodel" actions that effectively encapsulates a complete troubleshooting model. The virtual actions then get a cost equal to the ECR of the attached model, and conditional probabilities are assigned to it as with normal actions. Furthermore, one can associate a cost for entering and leaving the submodel. Thus, if we model each cluster as a troubleshooting model and create virtual submodel actions for each nested cluster, we essentially have a cost cluster model.

The model hierarchy has as its main motivation to isolate computational

requirements to a single model whereby real-time results can be achieved for very large models. However, it seems to come at the expense of a somewhat rough approximation in dealing with (a) the efficiency of the virtual actions, and (b) the determination of the number of actions to solve before leaving a cluster. It seems that inside information is assumed such that the system can be tested at all times. Since the models may contain dependent actions and questions, this approach is more general than the pure cost-cluster frameworks.

In (Koca and Bilgic, 2004b) two cost matrices encode the cost of steps conditional on the previous steps. The first matrix encodes the reduction in cost of an action α induced by the previous step whereas the second matrix encodes the reduction in cost induced by performing an action *anytime* before α. Since costs can be negative, this approach can emulate a cost-cluster framework. Like the method of (Skaanning and Vomlel, 2001), their approach is slightly more general in the sense that DAG structures can be modelled. Their method does handle the possibility of performing several actions before leaving a cluster, though this happens somewhat implicitly.

(Warnquist et al., 2008) also describe a slightly more general cost cluster framework than (Langseth and Jensen, 2001), but they do not address the issue of finding an efficient algorithm. They allow for a general DAG-based description of cost clusters and then define a heuristic based on a limited-breadth depth-first search.

Finally, in version 2 of GeNIe (GeNIe, 2010), the tool has added support for "observation costs" in their diagnosis system. The tool allows for simple cost for observations and conditional costs of questions based on the configuration of the parent nodes. Furthermore, questions can be grouped such that we can model general cost clusters (but with questions only).

We find it reassuring that all literature in the domain of conditional costs accepts the concept of clusters as a very practical modelling technique.

6.2 Preliminaries

Let us now discuss troubleshooting notation and definitions for the domain of cost clusters. As stated earlier, we exclusively deal with cost clusters under the inside information assumption.

We assume that the actions $\mathcal{A}$ can be partitioned into $\ell + 1$ clusters $\mathcal{K}$,$\mathcal{K}_1, \ldots, \mathcal{K}_\ell$ where cluster $\mathcal{K}$ is the **top-level cluster** and the remaining are **bottom-level clusters**. The **cost of opening** a cluster $\mathcal{K}_i$ is C_i^o and the **cost of closing** it again is C_i^c. An example of such a model was given in Figure 6.1. We define $C_{\mathcal{K}_i} = C_i^o + C_i^c$, and an action α belongs to cluster $\mathcal{K}(\alpha)$.

During the course of troubleshooting we gather **evidence** $\boldsymbol{\epsilon}^i$ meaning that the first i actions failed to solve the problem, and we have by assump-

tion $P(\boldsymbol{\epsilon}^0) = 1$ because the device is faulty. We also write $\boldsymbol{\epsilon}^{x:y}$ as shorthand for $\bigcup_{i=x}^{y}\{\neg a_i\}$. $\mathcal{FA}(\boldsymbol{\epsilon})$ is the set of **free actions** consisting of all actions (excluding those already performed) from open clusters given evidence $\boldsymbol{\epsilon}$. $\mathcal{CA}(\boldsymbol{\epsilon})$ is the set of **confined actions** consisting of all actions from closed clusters given evidence $\boldsymbol{\epsilon}$. Note that we have $\mathcal{FA}(\boldsymbol{\epsilon}) \cup \mathcal{CA}(\boldsymbol{\epsilon}) = \mathcal{A}(\boldsymbol{\epsilon})$ and $\mathcal{FA}(\boldsymbol{\epsilon}) \cap \mathcal{CA}(\boldsymbol{\epsilon}) = \emptyset$ for all evidence $\boldsymbol{\epsilon}$. By performing an action α from $\mathcal{CA}(\boldsymbol{\epsilon})$ we pay the cost $C_{\mathcal{K}(\alpha)}$ because at this point we are certain that we must both open and close the cluster. In that case α is called an **opening action** (for $\mathcal{K}(\alpha)$), and all remaining actions of $\mathcal{K}(\alpha)$ are **released** by removing them from $\mathcal{CA}(\boldsymbol{\epsilon})$ and adding them to $\mathcal{FA}(\boldsymbol{\epsilon})$. The **conditional cost** $C_\alpha(\boldsymbol{\epsilon})$ of an action α given evidence $\boldsymbol{\epsilon}$ is given by $C_\alpha + C_{\mathcal{K}(\alpha)}$ if $\alpha \in \mathcal{CA}(\boldsymbol{\epsilon})$ and by C_α if $\alpha \in \mathcal{FA}(\boldsymbol{\epsilon})$.

A **troubleshooting sequence** is a sequence of actions $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ prescribing the process of repeatedly performing the next action until the problem is fixed or the last action has been performed. We shall write $s[k, m]$ for the subsequence $\langle \alpha_k, \ldots, \alpha_m \rangle$ and $s(k, m)$ for the subsequence $\langle \alpha_{k+1}, \ldots, \alpha_{m-1} \rangle$. The index of an opening action in a troubleshooting sequence s is called an **opening index**, and the **set of all opening indices** for s is denoted $\mathcal{Z}$ with $\mathcal{Z} \subseteq \{1, \ldots, n\}, |\mathcal{Z}| = \ell$. To measure the quality of a given sequence we use the following definition which takes into account the conditional costs.

Definition 33. *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be a troubleshooting sequence. Then the **expected cost of repair** (ECR) of s is given by*

$$ECR(s) \;=\; \sum_{i=1}^{n} P(\boldsymbol{\epsilon}^{i-1}) \cdot C_{\alpha_i}(\boldsymbol{\epsilon}^{i-1}) \,. \tag{6.1}$$

As always, our optimization problem is to find a troubleshooting sequence with minimal ECR. Since we have independent actions, we may simplify computations and notation somewhat because of the following result (cf. Proposition 8 on page 90).

Proposition 10. *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be a troubleshooting sequence. Then the ECR of s may be computed as*

$$ECR(s) = \sum_{i=1}^{n} C_{\alpha_i}(\boldsymbol{\epsilon}^{i-1}) \cdot \left(1 - \sum_{j=1}^{i-1} P_{\alpha_j}\right) \,, \tag{6.2}$$

where $1 - \sum_{j=1}^{i-1} P_{\alpha_j} = P(\boldsymbol{\epsilon}^{i-1})$.

Thus, due to our assumptions, we may completely ignore $\mathcal{F}$, $P(f)$, and $P(\alpha|f)$ once the repair probabilities have been computed. Therefore, we

mainly use P_α in the rest of this chapter. Using the set of opening indices $\mathcal{Z}$, we can rewrite the definition of ECR of a sequence s to

$$ECR(s) = \sum_{i=1} C_{\alpha_i} \cdot \left(1 - \sum_{j=1}^{i-1} P_{\alpha_j}\right) + \sum_{z \in \mathcal{Z}} C_{\mathcal{K}(\alpha_z)} \cdot \left(1 - \sum_{j=1}^{z-1} P_{\alpha_j}\right) \quad (6.3)$$

where we have decomposed the terms into those that rely on the cost of performing actions and those that rely on the cost of opening and closing a cluster. We define the **efficiency** of an action α given evidence $\boldsymbol{\epsilon}$ as $ef(\alpha\,|\boldsymbol{\epsilon}) = P_\alpha / C_\alpha(\boldsymbol{\epsilon})$, and we write $ef(\alpha)$ for the unconditional efficiency P_α / C_α. Finally, the **cluster efficiency** of an opening action is $cef(\alpha) = \frac{P_\alpha}{C_\alpha + C_{\mathcal{K}(\alpha)}}$.

Lemma 3. *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be an optimal troubleshooting sequence with opening indices $z_i \in \mathcal{Z}$. Then the $\ell+1$ subsequences $s[\alpha_1, \alpha_{z_1})$, $s[\alpha_{z_i}, \alpha_{z_{i+1}}) \, \forall i \in \{1, \ldots, \ell - 1\}$, and $s[\alpha_{z_\ell}, \alpha_n]$ are ordered with respect to descending efficiency.*

Proof. Between opening indices the costs are not conditional, and so we must sort by descending $ef(\cdot)$ to be optimal. □

We have now established that given the opening index for each cluster, it is a simple task of merging ordered sequences to establish an optimal sequence. The difficult part is to determine the opening indices which we may choose in at most $\binom{|\mathcal{A}|}{\ell}$ ways.

Given that without cost clusters we simply need to sort the action with respect to descending $\frac{P_\alpha}{C_\alpha}$-values, we might wonder if that could still guarantee an optimal sequence. The follwing example shows that it does not.

Example 12 (Sorting by $ef(\cdot)$)**.** *Consider the following model*

	P_α	C_α	$ef(\alpha)$	$cef(\alpha)$
α_1	0.5	1	0.5	0.5
α_2	0.2	2	0.1	0.1
β	0.3	1	0.3	0.097

where $\alpha_i \in \mathcal{K}$ (the top-level cluster) and $\beta \in \mathcal{K}_2$ with $C_{\mathcal{K}_2} = 2.1$. Then compare

$$\begin{aligned} ECR(\langle \alpha_1, \alpha_2, \beta \rangle) &= 1 + 0.5 \cdot 2 + 0.3 \cdot 3.1 = 2.93 \\ ECR(\langle \alpha_1, \beta, \alpha_2 \rangle) &= 1 + 0.5 \cdot 3.1 + 0.2 \cdot 2 = 2.95 \end{aligned}$$

which shows that following an P-over-C ordering is not optimal.

The above example does suggest that we should instead sort by cluster efficiency which take into account the conditional costs. The following example shows that this is not viable either.

Example 13 (Sorting by $cef(\cdot)$)**.** *We consider the following model*

	P_α	C_α	$ef(\alpha)$	$cef(\alpha)$
α_1	0.4	1	0.4	0.4
α_2	0.1	1	0.1	0.1
β_1	0.3	1	0.3	0.097
β_2	0.2	1	0.2	0.065

and so $C_{\mathcal{K}_2} = 2.1$*. We then compare*

$$\begin{aligned} ECR(\langle \alpha_1, \alpha_2, \beta_1, \beta_2 \rangle) &= 1 + 0.6 \cdot 1 + 0.5 \cdot 3.1 + 0.2 \cdot 1 = 3.35 \\ ECR(\langle \alpha_1, \beta_1, \beta_2, \alpha_2 \rangle) &= 1 + 0.6 \cdot 3.1 + 0.3 \cdot 1 + 0.1 \cdot 1 = 3.26 \end{aligned}$$

which shows that it is not optimal to perform α_2 *before opening cluster* $\mathcal{K}_2$*.*

It appears then that there is no trivial algorithm that can efficiently find an optimal sequence. However, in the next sections we shall see that a minor extension to the P-over-C algorithm suggested by (Langseth and Jensen, 2001) is in fact optimal for the flat cost-cluster model.

6.3 The Extended P-over-C Algorithm

The standard "P-over-C" algorithm works by sorting the actions based on descending efficiency (cf. Algorithm 8 on page 90). The extended algorithm works in a similar manner, but it also considers the efficiency of a cluster: if a cluster is more efficient than all remaining actions and clusters, we should perform some actions from that cluster first. This leads to the following definition.

Definition 34 (Efficiency of Cluster)**.** *The* ***efficiency of a cluster*** $\mathcal{K}$ *is defined as*

$$ef(\mathcal{K}) = \max_{\mathcal{M} \subseteq \mathcal{K}} \frac{\sum_{\alpha \in \mathcal{M}} P_\alpha}{C_{\mathcal{K}} + \sum_{\alpha \in \mathcal{M}} C_\alpha} \tag{6.4}$$

and the largest set $\mathcal{M} \subseteq \mathcal{K}$ *that maximizes the efficiency is called the* ***maximizing set*** *of* $\mathcal{K}$*. The sequence of actions found by sorting the actions of the maximizing set by descending efficiency is called the* ***maximizing sequence*** *of* $\mathcal{K}$*.*

It turns out that it is quite easy to calculate the efficiency of a cluster even though we maximize over an exponential number of subsets. The following result is a slightly more informative version of the one from (Langseth and Jensen, 2001):

Lemma 4. *Let* $\mathcal{K}$ *be a cluster. Then the maximizing set* $\mathcal{M}$ *can be found by including the most efficient actions of* $\mathcal{K}$ *until* $ef(\mathcal{K})$ *starts decreasing. Furthermore, all actions* α *in the maximizing set* $\mathcal{M}$ *have* $ef(\alpha) \geq ef(\mathcal{K})$ *and all actions* $\beta \in \mathcal{K} \setminus \mathcal{M}$ *have* $ef(\beta) < ef(\mathcal{K})$*.*

Algorithm 10 The extended P-over-C algorithm (Langseth and Jensen, 2001)

1: **function** EXTENDEDPOVERC($\mathcal{K}, \mathcal{K}_1, \ldots, \mathcal{K}_\ell$)
2: Sort actions of $\mathcal{K}$ and all $\mathcal{K}_i$ by descending $ef(\cdot)$
3: Calculate $ef(\mathcal{K}_i)$ and maximizing sets $\mathcal{M}_i$ for all $i \in \{1, \ldots, \ell\}$
4: Let $\mathcal{K}_{closed} = \{\mathcal{K}_i \mid i \in \{1, \ldots, \ell\}\}$
5: Let $\mathcal{A} = \{\alpha \mid \alpha \in \mathcal{K}$ or $\alpha \in \mathcal{K}_i \setminus \mathcal{M}_i$ for some $i\}$
6: Let $s = \langle\rangle$
7: **repeat**
8: Let β be the most efficient action in $\mathcal{A}$ or cluster in $\mathcal{K}_{closed}$
9: **if** β is an action **then**
10: Add action β to s
11: Set $\mathcal{A} = \mathcal{A} \setminus \{\beta\}$
12: **else**
13: Add all actions of the maximizing set
14: of cluster β to s in order of descending efficiency
15: Set $\mathcal{K}_{closed} = \mathcal{K}_{closed} \setminus \{\beta\}$
16: **end if**
17: **until** $\mathcal{K}_{closed} = \emptyset$ and $\mathcal{A} = \emptyset$
18: **return** s
19: **end function**

The algorithm is described in Algorithm 10. It greedily computes the most efficient cluster (or action) and performs the actions in the maximizing set (or action). We can see that line 2 takes at most $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time. Once the actions have been sorted, line 3-6 takes at most $O(|\mathcal{A}|)$ time. The loop in line 7-20 can be implemented to run in at most $O(|\mathcal{A}| \cdot \lg(\ell + 1))$ time by using a priority queue for the most efficient element of $\mathcal{A}$ and the most efficient element of each cluster. Thus the algorithm has $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ worst case running time. In the next section we prove that the algorithm returns an optimal sequence.

Example 14 (The extended P-over-C algorithm)**.** *We consider a model with three clusters, where $\mathcal{K}$ is the root cluster and $\mathcal{K}_\beta$ and $\mathcal{K}_\gamma$ are the bottom-level clusters. We have $C_{\mathcal{K}_\beta} = 2$ and $C_{\mathcal{K}_\gamma} = 1$, and the following model parameters:*

	P_α	C_α	$ef(\alpha)$	*cluster*	$ef(\mathcal{K})$
α_1	*0.14*	*1*	*0.14*	$\mathcal{K}$	
α_2	*0.11*	*1*	*0.11*	$\mathcal{K}$	
β_1	*0.20*	*1*	*0.067*	$\mathcal{K}_\beta$	*0.075*
β_2	*0.10*	*1*	*0.033*	$\mathcal{K}_\beta$	
γ_1	*0.25*	*1*	*0.125*	$\mathcal{K}_\gamma$	*0.15*
γ_2	*0.20*	*1*	*0.10*	$\mathcal{K}_\gamma$	

The maximizing set for $\mathcal{K}_\beta$ is $\{\beta_1, \beta_2\}$ and for $\mathcal{K}_\gamma$ it is $\{\gamma_1, \gamma_2\}$, and from this the cluster efficiencies have been calculated. Algorithm 10 returns the sequence $s = \langle \gamma_1, \gamma_2, \alpha_1, \alpha_2, , \beta_1, \beta_2 \rangle$ which has ECR

$$ECR\,(s) = 2 + 0.75 + 0.55 + 0.41 + 0.30 \cdot 3 + 0.10 = 4.71\ .$$

If we followed the simple P-over-C algorithm we would get the sequence $s^2 = \langle \alpha_1, \gamma_1, \alpha_2, \gamma_2, \beta_1, \beta_2 \rangle$ with ECR

$$ECR\left(s^2\right) = 1 + 0.86 \cdot 2 + 0.61 + 0.50 + 0.30 \cdot 3 + 0.10 = 4.83\ .$$

6.4 Correctness of The Extended P-Over-C Algorithm

We have earlier seen how easy it was to prove the P-over-C algorithm correct. As we shall see in this section, it turns out to be somewhat harder to give a similar proof for the extended P-over-C algorithm. We start with the following lemma.

Lemma 5. *Let a, b, c, and d be strictly positive numbers. Then*

$$\frac{a+b}{c+d} \otimes \frac{a}{c} \quad \text{if and only if} \quad \frac{b}{d} \otimes \frac{a}{c} \tag{6.5}$$

for any weak order $\otimes$ (e.g. $\geq$ and $\leq$).

Proof.

$$\frac{a+b}{c+d} \otimes \frac{a}{c} \Leftrightarrow a \cdot c + b \cdot c \otimes a \cdot c + a \cdot d \Leftrightarrow \frac{b}{d} \otimes \frac{a}{c}$$

□

The proof of Lemma 4 is then given below.

Proof. Let $\mathcal{M}$ consist of actions in $\mathcal{K}$ such that $ef(\mathcal{M})$ is maximized. Then $ef(\mathcal{M})$ equals

$$\begin{aligned} \frac{\sum_{\alpha \in \mathcal{M}} P_\alpha}{C_\mathcal{K} + \sum_{\alpha \in \mathcal{M}} C_\alpha} &= \frac{\sum_{\alpha \in \mathcal{M} \setminus \{\beta\}} P_\alpha + P_\beta}{C_\mathcal{K} + \sum_{\alpha \in \mathcal{M} \setminus \{\beta\}} C_\alpha + C_\beta} \\ &= \frac{S_P + P_\beta}{S_C + C_\beta} = \frac{P}{C} \end{aligned}$$

where β is chosen arbitrarily. Let furthermore $\gamma \in \mathcal{K} \setminus \mathcal{M}$. We shall prove

$$\frac{P_\beta}{C_\beta} \geq \frac{P}{C} > \frac{P_\gamma}{C_\gamma}$$

which implies the lemma. We first prove the left-most inequality. Because $ef(\mathcal{M})$ is maximal we have

$$\frac{S_P + P_\beta}{S_C + C_\beta} \geq \frac{S_P}{S_C} \text{ which is equivalent to } \frac{P_\beta}{C_\beta} \geq \frac{\mathrm{S}_P}{\mathrm{S}_C}$$

which again is equivalent to

$$\frac{P_\beta}{C_\beta} \geq \frac{S_P + P_\beta}{S_C + C_\beta}\,.$$

The second inequality is proved similarly. □

We have already established that actions between opening indices are sorted with respect to efficiency in an optimal sequence. When we look at actions around opening indices we get the following result.

Lemma 6. *Let $s = \langle \ldots, \alpha_x, \alpha_{x+1}, \ldots \rangle$ be an optimal troubleshooting sequence, and let $\mathcal{Z}$ be the opening indices of s. Then*

$$\begin{array}{lll} cef(\alpha_x) & \geq ef(\alpha_{x+1}) & \text{if } x \in \mathcal{Z},\ \alpha_{x+1} \in \mathcal{FA}(\boldsymbol{\epsilon}^{x-1}) \\ ef(\alpha_x) & \geq cef(\alpha_{x+1}) & \text{if } \alpha_x \in \mathcal{FA}(\boldsymbol{\epsilon}^{x-1}), x+1 \in \mathcal{Z} \\ cef(\alpha_x) & \geq cef(\alpha_{x+1}) & \text{if } x \in \mathcal{Z},\ x+1 \in \mathcal{Z} \end{array} \tag{6.6}$$

Proof. Apply Lemma 1 on page 87 and do some pencil pushing. For example, case 1: $x \in \mathcal{Z}$ and $\alpha_{x+1} \in \mathcal{FA}(\boldsymbol{\epsilon}^{x-1})$. In this case we have

$$\begin{array}{l} C_{\alpha_x} + C_{\mathcal{K}(\alpha_x)} + \big(1 - P\big(\alpha_x \,|\boldsymbol{\epsilon}^{x-1}\big)\big) \cdot C_{\alpha_{x+1}} \leq \\ \qquad C_{\alpha_{x+1}} + \big(1 - P\big(\alpha_{x+1} \,|\boldsymbol{\epsilon}^{x-1}\big)\big) \cdot \big(C_{\alpha_x} + C_{\mathcal{K}(\alpha_x)}\big) \\ \Updownarrow \\ P\big(\alpha_{x+1} \,|\boldsymbol{\epsilon}^{x-1}\big)\,\big[C_{\alpha_x} + C_{\mathcal{K}(\alpha_x)}\big] \leq P\big(\alpha_x \,|\boldsymbol{\epsilon}^{x-1}\big)\, C_{\alpha_{x+1}} \\ \Updownarrow \\ \qquad ef(\alpha_{x+1}) \leq cef(\alpha_x) \end{array}$$

because $P(\alpha_x) \geq P(\alpha_{x+1}) \Leftrightarrow P(\alpha_x \,|\boldsymbol{\epsilon}) \geq P(\alpha_{x+1} \,|\boldsymbol{\epsilon})$ for independent actions. □

If we, for example, compare with Example 12, we see that the second sequence could immediately have been discarded by use of the above Lemma. The following definitions allow us to raise the abstraction level somewhat.

Definition 35 (Efficiency of Subsequence)**.** *Let $s[x, y]$ be a subsequence of a troubleshooting sequence s. Then the **efficiency** of $s[x, y]$ is given by*

$$ef(s[x, y]) \quad = \quad \frac{\sum_{i=x}^{y} P_{\alpha_i}}{\sum_{i=x}^{y} C_{\alpha_i}(\boldsymbol{\epsilon}^{i-1})} \tag{6.7}$$

Definition 36 (Regular Subsequence)**.** *Let $s = \langle \ldots, \alpha_x, \ldots, \alpha_y, \ldots \rangle$ be a troubleshooting sequence. If all actions of the subsequence $s[x, y]$ belong to the same cluster, we say that the subsequence is **regular**. If furthermore $s[x, y]$ is as long as possible while not breaking regularity, we say that the subsequence is a **maximal regular subsequence**.*

Remark 22. *Any troubleshooting sequence can be partitioned into a sequence of regular subsequences, and if all the subsequences are maximal, this partition is unique.*

We then have the following quite intuitive result.

Lemma 7. *Let s be an optimal troubleshooting sequence, and let $s[x, x + k]$ and $s[y, y + \ell]$ (with $y = x + k + 1$) be two adjacent regular subsequences such that $\mathcal{K}(\alpha_x) \neq \mathcal{K}(\alpha_y)$ or such that neither x nor y is an opening index. Then*

$$ef(s[x, x + k]) \quad \geq \quad ef(s[y, y + \ell]) \tag{6.8}$$

Proof. We consider the sequence

$$s^2 = \langle \ldots, \alpha_{x-1}, \alpha_y, \ldots, \alpha_{y+\ell}, \alpha_x, \ldots, \alpha_{x+k}, \ldots \rangle$$

which is equal to s except that the two regular sequences have been swapped. Since s is optimal we have $ECR(s) - ECR\left(s^2\right) \leq 0$. Because the subsequences are regular and belong to different clusters or do not contain opening indices, the costs are the same in the two sequences in both s and s^2. Therefore, we get that the terms of $ECR(s) - ECR\left(s^2\right)$ equal

$$\begin{gathered}
C_{\alpha_x}(\boldsymbol{\epsilon}^{x-1}) \cdot \left[P\left(\boldsymbol{\epsilon}^{x-1}\right) - P\left(\boldsymbol{\epsilon}^{x-1}, \boldsymbol{\epsilon}^{y:y+\ell}\right)\right] \\
\vdots \\
C_{\alpha_{x+k}}(\boldsymbol{\epsilon}^{x+k-1}) \cdot \left[P\left(\boldsymbol{\epsilon}^{x+k-1}\right) - P\left(\boldsymbol{\epsilon}^{x+k-1}, \boldsymbol{\epsilon}^{y:y+\ell}\right)\right] \\
C_{\alpha_y}(\boldsymbol{\epsilon}^{y-1}) \cdot \left[P\left(\boldsymbol{\epsilon}^{y-1}\right) - P\left(\boldsymbol{\epsilon}^{x-1}\right)\right] \\
\vdots \\
C_{\alpha_{y+\ell}}(\boldsymbol{\epsilon}^{y+\ell-1}) \cdot \left[P\left(\boldsymbol{\epsilon}^{y+\ell-1}\right) - P\left(\boldsymbol{\epsilon}^{x-1}, \boldsymbol{\epsilon}^{y:y+\ell-1}\right)\right]
\end{gathered}$$

since the remaining terms cancel out. Now observe that

$$\begin{aligned}
P\left(\boldsymbol{\epsilon}^{x+i-1}\right) - P\left(\boldsymbol{\epsilon}^{x+i-1}, \boldsymbol{\epsilon}^{y:y+\ell}\right) &= 1 - \sum_{j=1}^{x+i-1} P_{\alpha_j} - \left[1 - \sum_{j=1}^{x+i-1} P_{\alpha_j} - \sum_{j=y}^{y+\ell} P_{\alpha_j}\right] \\
&= \sum_{j=y}^{y+\ell} P_{\alpha_j}
\end{aligned}$$

and, similarly,

$$\begin{aligned} P\left(\boldsymbol{\epsilon}^{y+i-1}\right) - P\left(\boldsymbol{\epsilon}^{x-1}, \boldsymbol{\epsilon}^{y:y+i-1}\right) &= 1 - \sum_{j=1}^{y+i-1} P_{\alpha_j} - \left[1 - \sum_{j=1}^{x-1} P_{\alpha_j} - \sum_{j=y}^{y+i-1} P_{\alpha_j}\right] \\ &= -\sum_{j=x}^{x+k} P_{\alpha_j} \, . \end{aligned}$$

So $ECR\,(s) - ECR\left(s^2\right) \leq 0$ is equivalent to

$$\left[\sum_{i=x}^{x+k} C_{\alpha_i}(\boldsymbol{\epsilon}^{i-1})\right] \cdot \sum_{j=y}^{y+\ell} P_{\alpha_j} \leq \left[\sum_{i=y}^{y+\ell} C_{\alpha_i}(\boldsymbol{\epsilon}^{i-1})\right] \cdot \sum_{j=x}^{x+k} P_{\alpha_j}$$

which yields the result. □

Lemma 8. *There exists an optimal troubleshooting sequence s where for each opening index $x \in \mathcal{Z}$, there is a maximal regular subsequence $s[x, x+j]$ ($j \geq 0$) that contains the maximizing sequence for cluster $\mathcal{K}(\alpha_x)$.*

Proof. Let s be an optimal troubleshooting sequence, and let x be an opening index. Let $s[x, x+j]$ be a maximal regular subsequence and assume that it does not contain the maximizing set. Then there exists $\alpha_y \in \mathcal{K}(\alpha_x)$ with $y > x + j + 1$ such that

$$ef(\alpha_y) > ef(s[x, x+j])$$

Observe that the subsequence $s[x, y-1]$ can be partitioned into $m > 1$, say, maximal regular subsequences $s_1, \ldots, s_m$ with $s_1 = s[x, x+j]$. By Lemma 7 we have

$$ef(\alpha_y) > ef(s_1) \geq ef(s_2) \geq \cdots \geq ef(s_m) \geq ef(\alpha_y)$$

where the last inequality follows by the fact that α_y is not an opening action (so we avoid $\geq cef(\alpha_y)$). This situation is clearly impossible. Therefore $s[x, x+j]$ must contain the maximizing set. By Lemma 3, it must also contain a maximizing sequence. □

Remark 23. *In the above proof there is a technicality that we did not consider: there might be equality between the efficiency of an action in the maximizing sequence, the efficiency of the maximizing sequence, and one or more free actions. This problem can always be solved by rearranging the actions, and so for all proofs we shall ignore such details for the sake of clarity.*

Finally, we have the following theorem:

Theorem 12. *Algorithm 10 returns an optimal troubleshooting sequence.*

Proof. By Lemma 7 we know that an optimal sequence can be partitioned into a sequence of maximal regular subsequences which is sorted by descending efficiency. If we consider Lemma 8 too, then we know that we should open the clusters in order of highest efficiency and perform at least all actions in their maximizing sequences as computed by Lemma 4. By Lemma 3 we know that the order of actions in the maximizing sequences is the optimal one. By Lemma 7 we also know that all free actions α with $ef(\alpha) > ef(\mathcal{K})$ must be performed before opening the cluster, and all free actions with $ef(\alpha) < ef(\mathcal{K})$ must be performed after opening the cluster and performing all the actions in its maximizing sequence. □

6.5 The Tree Cost Cluster Model

In this section we shall investigate an extension of the flat cluster model where the clusters can be arranged as a tree. We call such a model for a **tree cost cluster model**, and an example is given in Figure 6.2. In the tree cost cluster model, the ECR does not admit the simple decomposition of Equation 6.3. The complication is that several clusters might need to be opened before performing an action in a deeply nested cluster. We therefore call troubleshooting sequences in the tree cost cluster model for **tree troubleshooting sequences**. Unfortunately, it is easy to construct examples that show that Algorithm 10 will not yield optimal tree troubleshooting sequences.

The so-called "updating P-over-C" algorithm is like the P-over-C algorithm, but the efficiencies are recalculated whenever an action has been performed (see Algorithm 9 on page 91). This leads, for example, to a better heuristic in troubleshooting models with dependent actions. In a similar manner, we may define an "updating extended P-over-C" algorithm based on Algorithm 10 where efficiencies of actions and clusters are recalculated (this is an $O(|\mathcal{A}|^2)$ time algorithm). Unfortunately, the following example shows that neither Algorithm 10 nor its derived updating algorithm guarantees optimality for tree cost cluster models.

Example 15 (The extended P-over-C algorithm is not optimal)**.** *Consider a model with three clusters $\mathcal{K} = \{\alpha\}, \mathcal{K}_\beta = \{\beta_1, \beta_2\}$, and $\mathcal{K}_\gamma = \{\gamma_1, \gamma_2\}$ with $C_{\mathcal{K}_\beta}(\boldsymbol{\epsilon}^0) = 1$ and $C_{\mathcal{K}_\gamma}(\boldsymbol{\epsilon}^0) = 2$, and so $pa(\mathcal{K}_\gamma) = \mathcal{K}_\beta$ and $pa(\mathcal{K}_\beta) = \mathcal{K}$. The remaining model parameters are as follows:*

	P_α	C_α	$ef(\alpha)$	$ef(\mathcal{K})$	$cef(\alpha)$
α	0.2	4	0.05	0.05	0.05
β_1	0.2	1	0.2	0.1	0.1
β_2	0.1	1	0.1		0.05
γ_1	0.3	1	0.3	0.14	0.1
γ_2	0.2	1	0.2		0.066

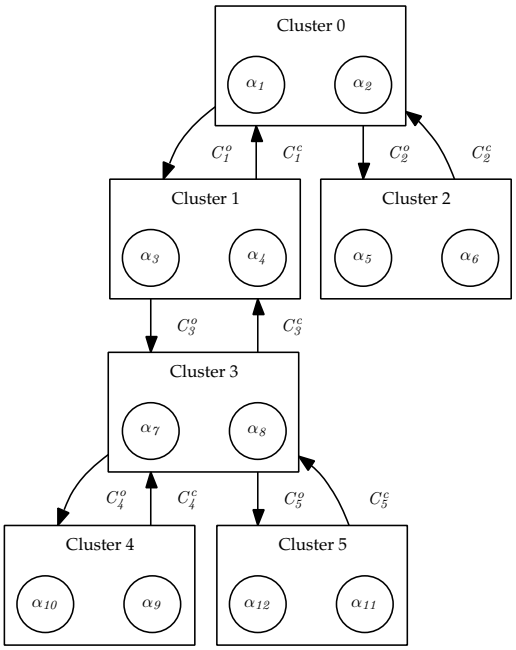


Figure 6.2: Example of a tree cost cluster model. To open a cluster $\mathcal{K}_i$ when the parent cluster is open we pay the cost C_i^o and to close a cluster given that all children clusters are closed we pay the cost C_i^c.

where we have computed the efficiency of a cluster by computing a maximizing set by considering all released actions. Lemma 4 makes it easy to calculate the set, E.g., for $\mathcal{K}_\gamma$ the maximizing set equals $\{\gamma_1, \gamma_2\}$.

If the strategy is to open a cluster with highest efficiency, then we should open $\mathcal{K}_\gamma$ first. After opening we should simply take the actions by descending efficiency (Lemma 3) which leads to the sequence $s^1 = \langle \gamma_1, \gamma_2, \beta_1, \beta_2, \alpha \rangle$. However, the optimal sequence is $s^2 = \langle \beta_1, \gamma_1, \gamma_2, \beta_2, \alpha \rangle$:

$$\begin{aligned} ECR\left(s^1\right) &= 3 + 0.7 + 0.5 + 0.3 + 0.2 \cdot 4 = 5.3 \\ ECR\left(s^2\right) &= 2 + 0.8 \cdot 2 + 0.5 + 0.3 + 0.2 \cdot 4 = 5.2 \end{aligned}$$

Remark 24. *The updating (non-extended) P-over-C algorithm leads to the sequence s^2 above, but this strategy is not optimal in general either since that is not even true for flat cluster models.*

Let us discuss the complexity for troubleshooting with the tree cost cluster model if we use brute-force. First observe that we have at most $\binom{|\mathcal{A}|}{\ell}$ ways of assigning the ℓ closed clusters to the $|\mathcal{A}|$ possible opening indices. Given the opening indices of the clusters, we can then construct a tree troubleshooting sequence by iterating in the following manner:

(a) pick the free actions in order of descending efficiency until an opening action occurs,

(b) then pick the opening action which is the most efficient action of all actions in newly opened clusters.

The correctness of this procedure relies on the fact that Lemma 3 generalizes to tree troubleshooting sequences. This is seen to be the case as all the subsequences consist of free actions and so there are no conditional costs involved. By sorting the free actions and the actions of each cluster by descending efficiency initially, and by using a priority queue for the free actions, then the sequence can be constructed in $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time. By using a matrix to compute the conditional cost, we can compute the ECR of this sequence in $O(|\mathcal{A}|)$ time. Therefore the problem of troubleshooting tree cost clusters can take at most $O(|\mathcal{A}|^{\ell+1} \cdot \lg |\mathcal{A}|)$. However, we shall in the following present a new algorithm that solves the tree cost cluster model in $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time.

First we need some additional definitions. The **conditional cost** $C_\alpha(\boldsymbol{\epsilon})$ of $\alpha \in \mathcal{K}_i$ will now depend on how many clusters that have been opened on the path from the root $\mathcal{K}$ to $\mathcal{K}_i$. We therefore let $\mathcal{AK}(\mathcal{K}_i|\boldsymbol{\epsilon})$ denote the **set of ancestor clusters** that needs to be opened on the path from the root $\mathcal{K}$ to $\mathcal{K}_i$ given evidence $\boldsymbol{\epsilon}$. We then define

$$C_{\mathcal{K}_i}(\boldsymbol{\epsilon}) = \sum_{\mathcal{K} \in \mathcal{AK}(\mathcal{K}_i|\boldsymbol{\epsilon})} C_{\mathcal{K}}, \qquad C_\alpha(\boldsymbol{\epsilon}) = C_\alpha + C_{\mathcal{K}(\alpha)}(\boldsymbol{\epsilon}) \tag{6.9}$$

Given this, Definition 33 is still valid for tree troubleshooting sequences. A single action is called an **atomic action**. A **compound action** consists of opening a cluster $\mathcal{K}$ and a sequence of actions in which each action may be either atomic or compound. We use A and B as notation for compound actions. Note that we shall not distinguish syntactically between atomic and compound actions inside a compound action. Also note that a compound action corresponds to a subsequence where the first action is an opening action, and the efficiency of a compound action is simply defined as the efficiency of the corresponding subsequence. If $\mathcal{T}$ is a tree cost cluster model and $\mathcal{K}$ is an arbitrary cluster in $\mathcal{T}$, then the **subtree model induced by** $\mathcal{K}$, denoted $\mathcal{T}_\mathcal{K}$, is a new tree cost cluster model containing exactly the clusters in the subtree rooted at $\mathcal{K}$, and with $\mathcal{K}$ as the open root cluster. If the induced subtree model is a flat cluster model, we call it a **flat subtree model**.

Algorithm 11 The bottom-up P-over-C algorithm

```
1: function BOTTOMUPPOVERC(T)
2:     Input: a tree cost cluster model T with root K
3:     Compute the model K↑ induced by absorbtion
4:         into K (see Definition 38)
5:     Let s = ⟨⟩
6:     while K↑ ≠ ∅ do
7:         Let A be the most efficient action in K↑
8:         if A is an atomic action then
9:             Add action A to s
10:        else
11:            Add all actions of A to s in the order prescribed by β
12:        end if
13:        Set K↑ = K↑ \ {A}
14:    end while
15:    Return s
16: end function
```

Definition 37 (Absorbtion)**.** *Let $\mathcal{T}_\mathcal{K} = \{\mathcal{K}, \mathcal{K}_1, \ldots, \mathcal{K}_\ell\}$ be a flat subtree model. Then the **absorbtion** of $\mathcal{K}_1, \ldots, \mathcal{K}_\ell$ into $\mathcal{K}$ is a new cluster $\mathcal{K}^\uparrow$ containing*

1. *for each child cluster $\mathcal{K}_i$, a compound action A_i induced by the maximizing sequence for $\mathcal{K}_i$, and*
2. *all remaining actions from $\mathcal{K}$,$\mathcal{K}_1$,...,$\mathcal{K}_\ell$.*

An example is given in Figure 6.3. Note that in $\mathcal{K}^\uparrow$ all the actions in a child cluster $\mathcal{K}_i$ that are not contained in the newly generated compound action will have a lower efficiency than the compound action for $\mathcal{K}_i$ (but it may be higher than the efficiencies of other compound actions).

Definition 38 (Model Induced by Absorbtion)**.** *Let $\mathcal{T}$ be a tree cost cluster model, and let $\mathcal{K}$ be any cluster in $\mathcal{T}$. Then $\mathcal{T}_\mathcal{K}$ may be transformed into a single cluster $\mathcal{K}^\uparrow$ by repeated absorbtion into the root cluster of flat subtree models. The resulting cluster $\mathcal{K}^\uparrow$ is called the **model induced by absorbtion** into $\mathcal{K}$.*

Remark 25. *By construction, the compound actions in a model induced by absorbtion into the root cluster $\mathcal{K}$ will only contain actions from the subtrees rooted at a child of $\mathcal{K}$.*

With these definitions we can now present Algorithm 11. The algorithm works in a bottom-up fashion, basically merging leaf clusters into their parents (absorbtion) until the tree is reduced to a single cluster. Then an optimal sequence is constructed by unfolding compound actions (and performing the contained actions) when they are most efficient. The following example illustrates how the algorithm works.

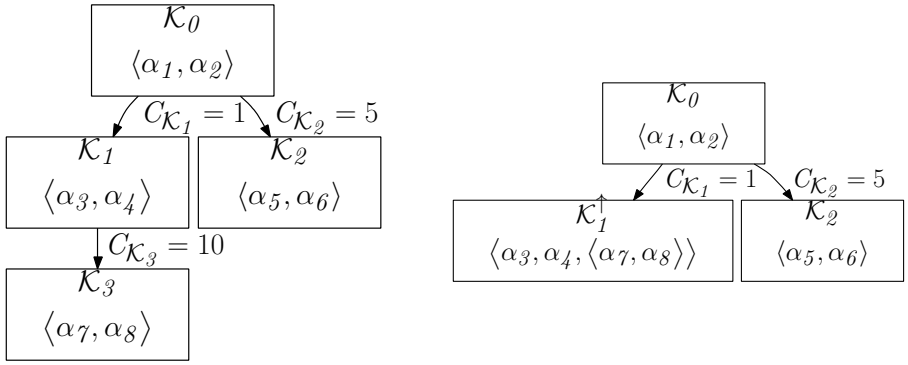


Figure 6.3: Left: the initial tree cost cluster model of Example 16. Right: the tree cost cluster model resulting after absorbtion into $\mathcal{K}_1$.

Example 16 (The bottom-up P-over-C algorithm)**.** *We consider the following model where the initial cluster structure is given in Figure 6.3 on the left:*

	P_α	C_α	$ef(\alpha)$
α_1	0.1	1	0.1
α_2	0.05	1	0.05
α_3	0.2	1	0.2
α_4	0.15	2	0.075
α_5	0.1	1	0.1
α_6	0.05	2	0.025
α_7	0.25	1	0.25
α_8	0.1	1	0.1

We start with the cluster $\mathcal{K}_3$ which has the maximizing sequence $\langle \alpha_7, \alpha_8 \rangle$ since $\frac{0.25}{11} < \frac{0.35}{12} \approx 0.029$. This leads to the model in Figure 6.3 on the right where the formed compound action is the least efficient in cluster $\mathcal{K}_1^{\uparrow}$. Similarly, the we have $\frac{0.1}{6} < \frac{0.15}{8} \approx 0.019$ so the compound action $\langle \alpha_5, \alpha_6 \rangle$ is merged into cluster $\mathcal{K}_0$ (see Figure 6.4 on the left). Finally, the maximizing sequence of $\mathcal{K}_1^{\uparrow}$ is simply α_3 and the resulting tree troubleshooting sequence is found by simple merging by efficiency. In this case the order of α_1 and α_3 is immaterial.

Before we prove the correctness of the algorithm, we shall discuss its complexity.

Lemma 9. *Algorithm 11 can be made to run in $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time.*

Proof. First we sort the actions of all clusters in the tree $\mathcal{T}$—this takes at most $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time. During absorbtion, it is important to avoid merging all actions of the child clusters into the parent cluster. Instead, we merge only the compound actions into the parent cluster (this takes $O(\ell \cdot \lg |\mathcal{A}|)$

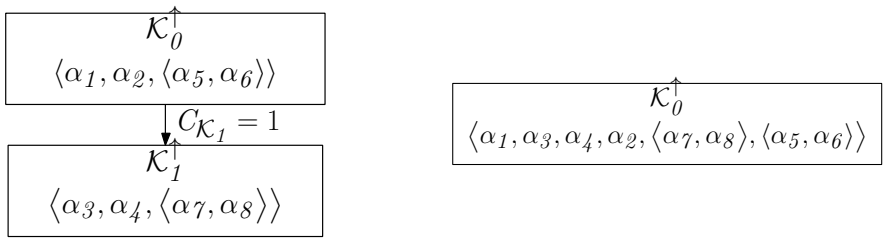


Figure 6.4: Left: the tree cost cluster model of Example 16 after further absorbtion of $\mathcal{K}_2$ into $\mathcal{K}_0$. Right: the final model after no more absorbtion is possible. The resulting sequence is an optimal tree troubleshooting sequence.

time overall) and create a priority queue holding the *most efficient* remaining action of each child cluster. When creating a compound action for a parent cluster, we then use actions from the priority queue as needed, and update the priority queue whenever an action is taken out (this requires moving the most efficient action (if any) from the list of sorted actions of the cluster of the action taken out of the queue). Since an action taken out of this priority queue never enters the queue again, creating all the compound actions can never take more than $O(|\mathcal{A}| \cdot \lg \ell)$ time. As the absorbtion process moves towards the root, we are forced to merge priority queues from different subtrees. We ignore that Fibonacci heaps can be merged in $O(1)$ time such that an implementation is not forced to use these, and instead we assume an $O(\lg \ell)$ merging procedure. Since we need to merge at most ℓ times, the result follows. □

In the following we shall prove that Algorithm 11 computes an optimal tree troubleshooting sequence. The first two lemmas are minor generalizations of previous lemmas, and the proofs are almost identical. The first one we have already discussed.

Lemma 10. *Lemma 3 generalizes to tree troubleshooting sequences, that is, actions between opening indices must be sorted by effiency in an optimal tree troubleshooting sequence.*

Lemma 11. *Lemma 7 generalizes to subsequences of actions that consists of (i) only free actions, or (ii) actions from the same subtree. That is, case (ii) means that the result holds for adjacent subsequences containing actions from different subtrees.*

Proof. Again the two adjacent subsequences will have identical costs even though we swap them. On the other hand, this is not the case if the subsequence contains actions from overlapping subtrees. □

Next we shall investigate the special properties of the compound actions generated by the absorbtion process.

Definition 39 (Maximizing Compound Action)**.** *Let $\mathcal{T}$ be a tree cost cluster model, and let $\mathcal{K}$ be any non-leaf cluster in $\mathcal{T}$. A **maximizing compound action** $\hat{A}$ **for** $\mathcal{K}$ **in** $\mathcal{T}$ is defined as any most efficient compound action in the model induced by absorbtion into $\mathcal{K}$.*

Lemma 12. *Let $\mathcal{T}$ be a tree cost cluster model, and let $\mathcal{K}$ be any non-leaf cluster in $\mathcal{T}$. Let $\mathcal{T}_{\mathcal{K}}$ be the subtree model induced by $\mathcal{K}$, and let $\hat{A}$ be a maximizing compound action for $\mathcal{K}$ in $\mathcal{T}$. Then*

$$ef(\hat{A}) \quad \geq \quad ef(B) \tag{6.10}$$

where B is any possible compound action in $\mathcal{T}_{\mathcal{K}}$ not including atomic actions from $\mathcal{K}$.

Proof. We proceed by induction over all tree structures. Basis is a flat cluster model $\mathcal{T} = \{\mathcal{K}, \mathcal{K}_1, \ldots, \mathcal{K}_\ell\}$ with compound actions $\hat{B_1}, \ldots, \hat{B_\ell}$ of $\mathcal{K}$ and $\hat{A} = \max_i \hat{B_i}$. Let B be any compound action including actions from clusters in $\mathcal{T} \setminus \{\mathcal{K}\}$, and assume that $ef(B) > ef(\hat{A})$. We shall use the fact

$$\min_i^n \frac{P_i}{C_i} \leq \frac{\sum_i^n P_i}{\sum_i^n C_i} \leq \max_i^n \frac{P_i}{C_i} \tag{6.11}$$

(which is also known as Cauchy's third inequality (Steele, 2004)). Then by Equation 6.11, B cannot be formed by any combination of the $\hat{B_i}$'s as this would not increase the efficiency. Therefore B must be formed by either a strict subset or a strict superset of one of the $\hat{B_i}$'s. If B is a subset of any $\hat{B_i}$, then the maximality of $\hat{B_i}$ leads to a contradiction. If B is a superset of any $\hat{B_i}$, then it will include subsets of actions from a set of clusters with subscripts $\mathcal{I} \subseteq \{1, \ldots, \ell\}$. Let us denote the subsets from each $\mathcal{K}_i$ as B_i. We then have

$$ef(B) = \frac{\sum_{i \in \mathcal{I}} P_{B_i}}{\sum_{i \in \mathcal{I}} C_{B_i}} \leq \max_{i \in \mathcal{I}} \frac{P_{B_i}}{C_{B_i}} \leq \max_{i \in \{1,\ldots,\ell\}} \frac{P_{\hat{B_i}}}{C_{\hat{B_i}}} = ef(\hat{A})$$

where the first inequality follows by Equation 6.11, the second follows by the definition of compound actions formed during absorbtion, and the last equality is by definition of a maximizing compound action. Since the sets B_i were chosen arbitrarily, we get a contradiction. Hence in all cases $ef(\hat{A}) \geq ef(B)$.

Induction step: we assume the Lemma is true for all children $\mathcal{K}_i, \ldots, \mathcal{K}_\ell$ of an arbitrary cluster $\mathcal{K}$ where the children have maximizing compound actions $\hat{B_1}, \ldots, \hat{B_\ell}$. A similar argument as above then shows that the lemma is true for $\mathcal{K}$ as well. □

Lemma 13. *Let $\mathcal{T}$ be a tree cost cluster model with root cluster $\mathcal{K}$. Then there exists an optimal tree troubleshooting sequence s that contains (as subsequences) all the compound actions of the model induced by absorbtion into $\mathcal{K}$. Furthermore, the compound actions in s are ordered by descending efficiency (in each cluster).*

Proof. Let $s = \langle \alpha_1, \ldots, \alpha_x, \ldots, \alpha_{x+k}, \ldots \rangle$ be an optimal tree troubleshooting sequence and let α_x be an opening action, and let $s[x, x+k], k \geq 0$ be the sequence of maximal length of actions from the same subtree. Let furthermore $s[x, x+k]$ be the first subsequence that contradicts the lemma, that is, $s[x, x+k]$ does not contain the compound action $\hat{A}$ for the cluster $\mathcal{K}(\alpha_x)$. Then there exists an atomic action $\alpha_y \in \hat{A}$ (with $y > x+k+1$) such that $\alpha_y \notin s[x, x+k]$. We then have

$$ef(\alpha_y) > ef(\hat{A}) > ef(s[x, x+k])$$

because all atomic actions in a compound action are more efficient than the compound action itself, and because $\hat{A}$ is the most efficient compound action in the subtree rooted at $\mathcal{K}(\alpha_x)$ (Lemma 12). We can then partition the actions between α_{x+k} and α_y into $m > 1$, say, subsequences (of maximal length) $s_1, \ldots, s_m$. If one (or more) of these subsequence is more efficient than α_y, we immediately get a contradiction to optimality of s because such a subsequence can be moved before $s[x, x+k]$ (Lemma 11). So we can assume that all the m subsequences are less efficient than α_y. Then by successive application of Lemma 11 we can decrease the ECR by moving α_y to position $x+k+1$. However, this again contradicts that s was optimal. Hence $s[x, x+k]$ must contain $\hat{A}$.

By Lemma 11 it follows that the order of the compound actions must be by descending efficiency. □

Theorem 13. *The bottom-up P-over-C algorithm (Algorithm 11) can return an optimal troubleshooting sequence in $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time.*

Proof. By Lemma 13 we only need to establish the order of the free actions between compound actions. By Lemma 11 it follows that any compound action is preceeded by more efficient free actions and followed by less efficient free actions. Then by Lemma 9 the result follows. □

6.6 Remarks on Open Problems

First we shall consider how we might extend the tree cost cluster model. It is natural to ask if the efficient algorithm of the tree cost cluster model can be generalised to handle a DAG structure over the clusters. We do not have an answer to that problem, but let us discuss how we may exploit the bottom-up P-over-C algorithm for sufficiently simple DAG structures. For

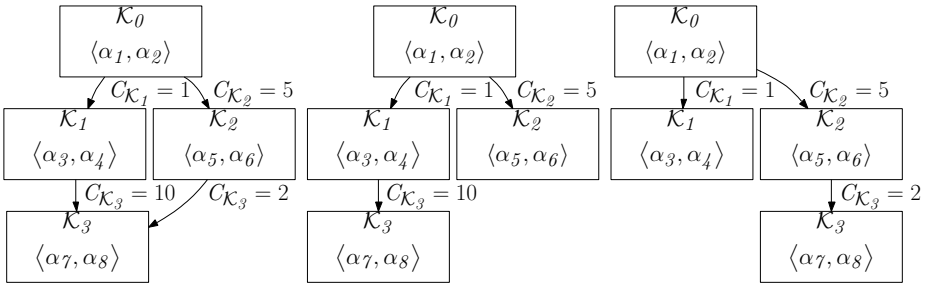


Figure 6.5: Left: an example of a DAG cost cluster model. Middle and Right: the two tree cost cluster models induced by the DAG model.

example, imagine a model structure as the one depicted in Figure 6.5 on the left. In an optimal troubleshooting sequence we know that exactly one path to the bottom cluster must be taken. So we can convert the DAG into a set of trees among which the optimal sequence of opening indices must be present, and then we can apply the bottom-up P-over-C algorithm to solve each induced tree model in isolation. The tree cost cluster models induced by the DAG are also shown in the figure.

This scheme should work well if the number of clusters with more than one parent cluster is low, and we believe many real-world domains are sufficiently simple in their decomposition structure to make the technique applicable. However, it appears that the number of trees induced by a DAG grows exponentially in the number of clusters with more than one parent. As discussed in Chapter 4, the problem is in fact NP-hard. On the other hand, it seems intuitive that the tree on the left in Figure 6.5 cannot hold the optimal solution as the cost of opening and closing cluster $\mathcal{K}_3$ is larger than the cost of accessing both cluster $\mathcal{K}_2$ and $\mathcal{K}_3$ in the tree on the right. Therefore, it would be interesting (and of practical importance) to have theoretical results that reduce the number of induced tree cost cluster models that need to be considered.

Let us now look at a different topic. As we have discussed earlier, troubleshooting with independent faults can be accomplished in $O(|\mathcal{F}| \cdot \lg |\mathcal{F}|)$ in models with clusters (Srinivas, 1995). It is assumed that there is an associated action with each fault which can inspect and repair the fault. (As noted by (Koca, 2003) we need not necessarily have a one-to-one correspondence between faults and actions as we may simply form a new **compound fault** consisting of all the faults that can be repaired by an action to re-gain a one-to-one correpondence). Instead of sorting actions by efficiency, we simply sort the desceding efficiency of the faults, that is,

$$\frac{P(f_i)}{C_{\alpha_i} \cdot (1 - P(f_i))} \quad \geq \quad \frac{P(f_{i+1})}{C_{\alpha_{i+1}} \cdot (1 - P(f_{i+1}))} \tag{6.12}$$

is true for all adjacent faults in an optimal sequence of faults to inspect. As soon as we introduce cost clusters we may try to find something similar to Lemma 4 to compute a **maximizing compound fault**. If we take the same approach as Lemma 4, then we have to compute

$$\arg\max_{\mathcal{M}\subseteq\mathcal{K}} \frac{1 - \prod_{f\in\mathcal{M}} P(\neg f)}{\left(C_{\mathcal{K}} + \sum_{\alpha\in\mathcal{A}(\mathcal{M})} C_\alpha\right) \cdot \prod_{f\in\mathcal{M}} P(\neg f)} \tag{6.13}$$

since the probability of a compound fault is taken as 1 minus the probability that no atomic faults are present. We regard it as non-trivial to compute this set. In any case, we leave the whole problem of defining algorithms for cost clusters of independent faults as an interesting open problem.

6.7 Summary

In this chapter we continued our discussion on action-based algorithms. This time we investigated the domain of conditional costs in the form of various cost cluster models. The actions are grouped into clusters where actions in a particular cluster share the same enabling cost.

This appears to be a very narrow domain, but when we investigated earlier ideas for conditional cost models, they all supported cost clusters in some form or another. Under the assumption of independent actions and inside information, we could define the expected cost of repair almost identical to previous definitions for action sequences, the important difference being that the cost of opening and closing a cluster must now be taken into account. We then gave a full theoretical account of the extended P-over-C algorithm (Langseth and Jensen, 2001) for troubleshooting with the flat cost model. It turns out that the extended P-over-C algorithm is indeed optimal in this domain.

Finally, we generalized the flat cost cluster model to the tree cost cluster model. This lead to a new recursive merging procedure, the bottom-up P-over-C algorithm, which to our great satisfaction also turned out to be optimal and to be optimally efficient (that is, it runs in $O(|\mathcal{A}| \cdot \lg|\mathcal{A}|)$ time). We explained how models where the clusters form a DAG may be solved by generating a set of tree cost cluster models. This approach seems to scale poorly, and it remains an open problem if efficient, optimal algorithms exist for special cases of this domain. Another interesting open problem is whether there exists an efficient solution method for cost clusters of independent faults.

Chapter 7

Troubleshooting with Postponed System Test

> I tell you Egon, it [work] is the most useless waste of time that I have ever experienced.
> *–Yvonne (Olsen Banden)*

Just like Yvonne, most people have experienced that there is a limit to how much work that one wants to do compared to how much personal benefit one gains. When facing an NP-hard problem, computer scientists face the same dilemma. We may spend inordinate amounts of computing power only to improve the solution marginally. In this section we shall therefore discuss a variety of heuristics and solutions for troubleshooting with postponed system test. When this work was done, we conjectured that the problem was NP-hard, and this was recently proven to be the case (Lín, 2011). It also turns out that this problem is closely related to troubleshooting with cost clusters without inside information which we briefly discuss.

7.1 Preliminaries

In traditional troubleshooting it has been assumed that each action is followed by a test that determines if the entire system is working. In traditional single-fault troubleshooting it has generally been assumed that the cost of this test C_W is so low that it should not even be modelled (a noticeable exception is (Heckerman et al., 1995)). The reason that this is problematic is simply that the system test cost can be so high that it should be postponed until we have performed more actions. For this chapter we continue to make the strict assumptions that (a) there are no questions, and (b) action are independent, but relax the assumption that the cost of the system test C_W is zero.

Before we can define our new optimization problem formally, we need some notation and terminology. Since we have independent actions, we

stick with the notation P_α for the repair probability established in the previous chapters. For each $V \subseteq \mathcal{A}$ we can form a **compound action** A with the following properties

$$P_A = \sum_{\alpha \in V} P_\alpha, \qquad C_A = \sum_{\alpha \in V} C_\alpha, \qquad |A| = |V| \,. \tag{7.1}$$

To make the distinction clear, we call actions $\alpha \in \mathcal{A}$ for **atomic actions**. To make it clear that the cost of an action includes the cost of testing the system C_W, we write C_{A+W} for $C_A + C_W$ and so C_A never includes the cost of testing the system. We can see that a partition of $\mathcal{A}$ into k subsets $\{V_1, \ldots, V_k\}$ induces a new model with independent actions $\{A_1, \ldots, A_k\}$. We shall also say that $\{A_1, \ldots, A_k\}$ is a **partition** of $\mathcal{A}$ whenever the corresponding sets of atomic actions form a partition of $\mathcal{A}$. If we furthermore arrange the compound actions into a sequence $\langle A_1, \ldots, A_k \rangle$, we call it for an **ordered partition** of $\mathcal{A}$.

From now on we shall not distinguish between the set of actions and the compound action; that is, we use A to mean a set of atomic actions and write $A = \{\alpha_1, \ldots, \alpha_{|A|}\}$—in essence this corresponds to the fact that the order of actions inside a compound action is irrelevant because we do not perform the system test until all actions have been carried out. (Note that this is different from the compound actions of the previous chapter where the order was important.)

A **compound troubleshooting sequence** is a sequence of compound actions $s = \langle A_1, \ldots, A_\ell \rangle$ prescribing the process of repeatedly performing the next action until the problem is fixed or the last action has been performed and such that $\{A_1, \ldots, A_\ell\}$ is a partition of $\mathcal{A}$. When each compound action contains only one action, we say that the sequence is **atomic**. A subsequence of s, $s[k, m] = \langle A_k, \ldots, A_m \rangle$ with $k \leq m + 1$, is called a **partial troubleshooting sequence** if $k > 1$ or $m < \ell$. If $k > m$, the subsequence is an **empty subsequence** and if $k = 1$, we may simply write $s[m]$.

Definition 40 (Expected Cost of Compound Troubleshooting Sequence)**.** *Let $s = \langle A_1, \ldots, A_\ell \rangle$ be a compound troubleshooting sequence. Then the **expected cost of repair** (ECR) of s is the mean of the cost until an action succeeds or all actions have been performed:*

$$ECR\,(s) \;=\; \sum_{i=1}^{\ell} C_{A_i+W} \cdot P\big(\boldsymbol{\epsilon}^{i-1}\big) \,. \tag{7.2}$$

As usual the problem is then to find a troubleshooting sequence with minimal $ECR\,(\cdot)$ over all possible sequences. We now define efficiency slightly differently.

Definition 41 (Efficiency of Compound Action)**.** *The efficiency of a compound action A is*

$$ef(A) \quad = \quad \frac{P_A}{C_{A+W}} \,. \tag{7.3}$$

We can extend Theorem 8 on page 88 without any problem to cover compound action sequences.

Corollary 3. *Let $s = \langle A_1, \ldots, A_\ell \rangle$ be a compound troubleshooting sequence. Then a necessary condition for s to be optimal, even when $C_W > 0$, is that*

$$ef(A_i) \geq ef(A_{i+1}) \qquad \text{for } i \in \{1, \ldots, \ell - 1\} \,. \tag{7.4}$$

Proof. Because a troubleshooting sequence partitions $\mathcal{A}$, the compound actions can be seen as virtual actions which are also independent. Then apply Lemma 1 on page 87 and simplify due to unconditional costs and independent actions. □

We may also compute the ECR of any compound troubleshooting sequence very easily:

Corollary 4. *Let $s = \langle A_1, \ldots, A_\ell \rangle$ be a compound troubleshooting sequence. Then the ECR of s may be computed as*

$$ECR\,(s) = \sum_{i=1}^{\ell} C_{A_i+W} \cdot \left(1 - \sum_{j=1}^{i-1} P_{A_j} \right) \,, \tag{7.5}$$

where $1 - \sum_{j=1}^{i-1} P_{A_j} = P\big(\boldsymbol{\epsilon}^{i-1}\big)$.

The following example shows that Theorem 9 on page 89 does not extend to arbitrary troubleshooting sequences when $C_W > 0$.

Example 17 (The P-over-C algorithm is not optimal)**.** *Consider a model with four atomic actions α_1, α_2, α_3 and α_4. The other properties of the model are as follows:*

	P_α	C_α	$C_{\alpha+W}$	$ef(\alpha)$
α_1	0.24	1	2	0.12
α_2	0.42	3	4	0.105
α_3	0.2	1	2	0.1
α_4	0.14	19	20	0.007

So $C_W = 1$. We have

$$ECR\,(\langle \alpha_1, \alpha_2, \alpha_3, \alpha_4 \rangle) = 2 + 0.76 \cdot 4 + 0.34 \cdot 2 + 0.14 \cdot 20 = 8.52$$

$$ECR\,(\langle \{\alpha_1, \alpha_2\}, \alpha_3, \alpha_4 \rangle) = 5 + 0.34 \cdot 2 + 0.14 \cdot 20 = 8.48$$

thus proving that it is more efficient to form the compound action $A = \{\alpha_1, \alpha_2\}$.

Algorithm 12 Exhaustive Search With Compound Actions

```
1: function GENERATEALLSEQUENCES(𝒜, &𝒮)
2:     Let all = ∅
3:     for i = 1 to |𝒜| do
4:         for all (|𝒜| choose i) ways to construct the first compound action A do
5:             if A ∉ 𝒮 then
6:                 Compute A from scratch
7:                 Set 𝒮 = 𝒮 ∪ {A}
8:             end if
9:             Let after = GENERATEALLSEQUENCES(𝒜 \ A, 𝒮)
10:            for s = ⟨A_1, ..., A_ℓ⟩ ∈ after do
11:                if ef(A_1) ≤ ef(A) then
12:                    Let s' = ⟨A, A_1, ..., A_ℓ⟩
13:                    Set all = all ∪ {s'}
14:                end if
15:            end for
16:        end for
17:    end for
18:    return all
19: end function
20: function OPTIMALCOMPOUNDACTIONSEQUENCE(𝒜)
21:    Let 𝒮 = ∅
22:    Let all = GENERATEALLSEQUENCES(𝒜, 𝒮)
23:    return arg min_{s∈all} ECR(s)
24: end function
```

To be able to test heuristic algorithms it is necessary with an algorithm that is guaranteed to find an optimal sequence. Exhaustive searches are always able to do this, albeit the algorithm might take exponential or super-exponential time. Algorithm 12 shows an exhaustive search with pruning (line 12, applying Theorem 3) and memoization (line 6-9).

Let us discuss the complexity of this algorithm. The algorithm recursively generates all possible sequences of compound actions. The number of such sequences is $n! \cdot 2^{n-1}$ because for each of the $n!$ permutations of the n atomic actions we can form 2^{n-1} different partitions. Thus the running time and memory requirement of the algorithm is $O(n! \cdot 2^n)$. Note, however, that this is not a tight upper bound since a troubleshooting sequence $s = \langle A_1, \ldots, A_\ell \rangle$ can be constructed by $\prod_{i=1}^{\ell} |A_i|!$ different orderings. Later we show that the complexity of a simple exhaustive search can be reduced to $O(n^3 \cdot n!)$.

In case someone else should be interested in deriving a tighter bound on the complexity, we now give a more formal analysis which may be used for inspiration. The time is dominated by `GenerateAllSequences(·)` which again executes in time proportional to the total number of sequences. This number may be described implicitly by the recurrence

$$\begin{aligned} T(n) &= 1 + \binom{n}{1} \cdot T(n-1) + \binom{n}{2} \cdot T(n-2) + \cdots + \binom{n}{n-1} \cdot T(1) \\ &= 1 + \sum_{i=1}^{n-1} \binom{n}{i} \cdot T(n-i) = 1 + \sum_{i=1}^{n-1} \binom{n}{i} \cdot T(i) \end{aligned} \tag{7.6}$$

By the substitution method we can prove the solution is $T(n) = O(n! \cdot 2^n)$. If we define

$$f(i) = \binom{n}{i} \cdot i! \cdot 2^i$$

then we can first determine which term in the sum in Equation 7.6 that dominates the others. This is done by solving for i in $f(i) < f(i+1)$, and the solution is $i < n - \frac{1}{2}$ so the last term dominates the others. Secondly we determine how much the $(n-1)$-th term dominates the others by computing $f(n-1) - f(i)$. We get

$$f(n-1) - f(i) = n! \cdot 2^i \cdot \left[2^{n-1-i} - \left(\prod_{x=2}^{n-1} x \right)^{-1} \right] \tag{7.7}$$

We can now write Equation 7.6 as

$$T(n) = 1 + \sum_{i=1}^{n-1} \binom{n}{n-1} \cdot (n-1)! \cdot 2^{n-1} - \sum_{i=1}^{n-2} f(n-1) - f(i) \tag{7.8}$$

and by some pencil pushing we get the desired result $T(n) = O(n! \cdot 2^n)$.

7.2 Two Simple Greedy Heuristics

In this section we shall develop two simple greedy algorithms. Furthermore, we also give examples showing that none of the algorithms are guaranteed to find an optimal troubleshooting sequence.

For the first algorithm we consider how the ECR of a troubleshooting sequence can be improved by merging two neighbouring actions into one compound action. We have the following result.

Theorem 14. *Let the two troubleshooting sequences s and s' be given as*

$$\begin{aligned} s &= \langle \ldots, \alpha_x, \alpha_{x+1}, \ldots \rangle \\ s' &= \langle \ldots, A, \ldots \rangle \end{aligned}$$

with $A = \{\alpha_x, \alpha_{x+1}\}$ and everything else being the same. Then s' has a strictly lower ECR than s if and only if

$$C_W > \frac{C_{\alpha_{x+1}} \cdot P_{\alpha_x}}{1 - \sum_{j=1}^{x} P_{\alpha_j}} . \tag{7.9}$$

Proof. When the sequence with the new compound action is better, we must have

$$ECR\left(s'\right) - ECR\left(s\right) \leq 0 .$$

We observe that most of the terms cancel each other out and we are left with

$$\begin{aligned} & C_{A+W} \cdot P\left(\boldsymbol{\epsilon}^{x-1}\right) - \left[C_{\alpha_x+W} \cdot P\left(\boldsymbol{\epsilon}^{x-1}\right) + C_{\alpha_{x+1}+W} \cdot P(\boldsymbol{\epsilon}^{x})\right] < 0 \\ & \qquad \Updownarrow \\ & C_{A+W} < C_{\alpha_x} + C_W + (C_{\alpha_{x+1}} + C_W) \cdot \frac{P(\boldsymbol{\epsilon}^{x})}{P(\boldsymbol{\epsilon}^{x-1})} \\ & \qquad \Updownarrow \\ & C_{\alpha_{x+1}} \cdot \left(1 - \frac{P(\boldsymbol{\epsilon}^{x})}{P(\boldsymbol{\epsilon}^{x-1})}\right) < C_W \cdot \frac{P(\boldsymbol{\epsilon}^{x})}{P(\boldsymbol{\epsilon}^{x-1})} \\ & \qquad \Updownarrow \\ & C_W > C_{\alpha_{x+1}} \cdot \left(\frac{P\left(\boldsymbol{\epsilon}^{x-1}\right)}{P(\boldsymbol{\epsilon}^{x})} - 1\right) \\ & \qquad \Updownarrow \\ & C_W > C_{\alpha_{x+1}} \cdot \left(\frac{P_{\alpha_x}}{1 - \sum_{j=1}^{x} P_{\alpha_j}}\right) \end{aligned}$$

□

Theorem 14 immediately suggests a procedure where we greedily merge adjacent actions until the ECR is no longer improved. We start with an atomic troubleshooting sequence and form compound actions by repeated application of Theorem 14. Justified by Theorem 9, there are two obvious choices of the initial sorting: with respect to descending efficiency or descending $\frac{P_\alpha}{C_\alpha}$-value. We have summarized this procedure in Algorithm 13. (The special function $1\mathrm{st}\,\mathrm{arg}(\cdot)$ returns the first index for which the boolean argument returns true.) As specified, the algorithm runs in $O(|\mathcal{A}|^2)$ time. However, if the sum in line 5 in `CompoundAction`$(\cdot)$ are precomputed, the total cost of all calls to `CompoundAction`$(\cdot)$ can be made linear. The running time of the algorithm is then dominated by the initial sorting of the actions leading to an $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ worst case running time.

Algorithm 13 Computing Compound Actions Greedily

function COMPOUNDACTION($\langle \alpha_1, \ldots, \alpha_n \rangle, x$)
  if $x = n$ **then**
    return $\{\alpha_n\}$
  end if
  Let $m = \overset{n}{\underset{i=x}{1\text{st}\arg}} \left(C_W \leq \frac{C_{\alpha_{i+1}} \cdot P_{\alpha_i}}{1 - \sum_{j=1}^{i} P_{\alpha_j}} \right)$
  return $\{\alpha_x, \ldots, \alpha_m\}$
end function
function GREEDYCOMPOUNDACTIONSEQUENCE($\mathcal{A}$)
  Let $s' = \langle \alpha_1, \ldots, \alpha_n \rangle$ such that $ef(\alpha_i) \geq ef(\alpha_{i+1})$ or $\frac{P_{\alpha_i}}{C_{\alpha_i}} \geq \frac{P_{\alpha_{i+1}}}{C_{\alpha_{i+1}}}$.
  Let $s = \langle\rangle$
  Let $i = 1$
  while $i \leq n$ **do**
    Let $A =$ COMPOUNDACTION(s', i)
    Set $s = s + A$ ▷ Concatenate seqences
    Set $i = i + |A|$
  end while
  return s
end function

Is Algorithm 13 guaranteed to find an optimal troubleshooting sequence? The example below shows that this is not the case (of course).

Example 18 (Algorithm 13)**.** *We consider the following model:*

	P_α	C_α	$C_{\alpha+W}$	$ef(\alpha)$	$\frac{P_\alpha}{C_\alpha}$
α_1	*0.61*	*1*	*11*	*0.055*	*0.61*
α_2	*0.21*	*5*	*15*	*0.014*	*0.042*
α_3	*0.18*	*3*	*13*	*0.013*	*0.06*

So $C_W = 10$. Using the $ef(\cdot)$ order Algorithm 13, then first computes

$$10 < \frac{5 \cdot 0.61}{1 - 0.61} \approx 7.8 \,.$$

Since this is false, it continues with the second test

$$10 < \frac{3 \cdot 0.21}{1 - 0.61 - 0.21} = 3.5$$

which means that the algorithm suggest the sequence consisting of a single compound action $\{\alpha_1, \alpha_2, \alpha_3\}$. If we repeat the calculations using the $\frac{P_\alpha}{C_\alpha}$-order, we get the same result. The ECR is easily seen to be 19*, but*

$$ECR\left(\langle \alpha_1, \{\alpha_2, \alpha_3\} \rangle\right) = 11 + 0.39 \cdot 18 = 18.02 \,.$$

Algorithm 14 Greedy Algorithm with Max-Efficient Actions

```
1: function GREEDYCOMPOUNDACTIONSEQUENCE2(𝒜)
2:     Let s' = ⟨α_1, ..., α_n⟩ such that P_{α_i}/C_{α_i} ≥ P_{α_{i+1}}/C_{α_{i+1}}
3:     Let s = ⟨⟩
4:     Repeatedly add the most efficient compound action to s
5:     return s
6: end function
```

Remark 26. *We shall give many examples in this chapter showing that the algorithms are not optimal. Even though this is obvious when the problem is NP-hard, we do so to give some insights about the cases where admissibility is lost.*

Let us now consider another greedy heuristic. The algorithm is based on the idea that we should repeatedly form the most efficient compound action until all atomic actions have been assigned a compound action. To do this, we need a simple way of computing the most efficient compound action from a set of atomic actions. To this aim we shall reuse a minor variation of Lemma 4 on page 118.

Lemma 14. *Let $V \subseteq \mathcal{A}$ be given. Let A be a compound action consisting of actions in V with $ef(A)$ maximal and with $|A|$ minimal over those actions with maximal efficiency. Then A can be found by the following procedure: define $ef(A) = 0$ when $A = \emptyset$; then repeatedly add an action $\alpha \in V$ to A with the highest $\frac{P_\alpha}{C_\alpha}$-value until $ef(A)$ does not increase.*

Proof. The proof is virtually identical to the proof of Lemma 4 on page 118. □

Remark 27. *One might wonder why we do not simply reuse Lemma 4 on page 118. In case several actions have the same efficiency, the set $\mathcal{M}$ that maximizes the efficiency of the compound action need not be unique. In such case we shall prefer any smallest of such maximizing sets $\mathcal{M}$ because it might be non-optimal to choose a larger set. To see this, then consider an action α which could have been part of the maximizing set: if α is included in $\mathcal{M}$, then the cost of α will be paid with a lower probability than if α is performed immediately after testing the system. Thus, splitting such an atomic action out of the compound action could be beneficial. On the other hand, the next C_W cost will be multiplied with a larger probability. In certain case we may use Theorem 14 on page 139, for example, if the maximizing compound action could consume all remaining atomic actions. In the end, the chance that this situation arises is probably low for real-world models.*

Algorithm 14 shows a greedy algorithm based on Lemma 14. Like Algorithm 13 we can precompute the sums used in the efficiency and get a worst-case running time of $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$. The following example illustrates how Algorithm 14 works.

Example 19 (Algorithm 14)**.** *Consider the following model*

	P_α	C_α	$C_{\alpha+W}$	$\frac{P_\alpha}{C_\alpha}$
α_1	0.15	1	2	0.150
α_2	0.35	2	3	0.175
α_3	0.50	3	4	0.167

where the ordering considered is $\langle \alpha_2, \alpha_3, \alpha_1 \rangle$. Algorithm 14 computes the following values to determine the first compound action:

$$\begin{aligned} ef(\{\alpha_2\}) &= \frac{0.35}{3} = 0.117 \\ ef(\{\alpha_2, \alpha_3\}) &= \frac{0.35 + 0.5}{6} = 0.1416 \\ ef(\{\alpha_2, \alpha_3, \alpha_1\}) &= \frac{1}{7} = 0.1429 \end{aligned}$$

Thus the suggested sequence is one single compound action with ECR 7. However,

$$ECR\left(\langle \{\alpha_2, \alpha_3\}, \alpha_1 \rangle\right) = 6 + 0.15 \cdot 2 = 6.3$$

which shows the algorithm is not optimal.

Having established the heuristic nature of both greedy algorithms, we would like to know if they at least find a local optimum. We use the following definition of a local optimum.

Definition 42 (Consistent Compound Action Sequence)**.** *Let $s = \langle \alpha_1, ..., \alpha_n \rangle$ be a sequence of atomic actions, and let $s^c = \langle B_1, \ldots, B_m \rangle$ be a compound action sequence over the same set of atomic actions as s. Then s^c is said to be **consistent** with s if $\forall\, k < l$*

$$\alpha_i \in B_k \text{ and } \alpha_j \in B_l \implies i < j. \tag{7.10}$$

In other words, s^c can be seen as an ordered partition of s.

If an atomic action sequence s is given, we call the set of all compound action sequences consistent with s for $\mathcal{CS}(s)$. A local optimum is then a compound action sequence $s^c \in \mathcal{CS}(s)$ with minimal ECR. We then say the (ordered) partition induced by s^c is **optimal**.

From Example 18 and 19 it follows that neither Algorithm 13 nor Algorithm 14 guarantees an optimal partition.

7.3 An Algorithm For Optimal Partitioning

We have now seen how the greedy algorithms fail to provide optimal sequences even under the restriction that the solution must be consistent with a single seed-sequence s of atomic actions. However, since there are $2^{|\mathcal{A}|-1}$ ways to partition the actions of s, a brute force approach is quite impractical even for models of moderate size. By exploiting Theorem 15 below we can construct an $O(|\mathcal{A}|^3)$ time dynamic programming algorithm.

Lemma 15. *Let $k \geq 0$ and let $s^c = \langle A_1, \ldots, A_\ell \rangle$ be a partial troubleshooting sequence that is performed after the first k actions have been performed. Then the contribution of s^c to the total ECR is given by*

$$ECR_{(k)}(s^c) \;=\; \sum_{i=1}^{\ell} P\left(\boldsymbol{\epsilon}^{k+i-1}\right) \cdot C_{A_i+W} \tag{7.11}$$

where

$$P\left(\boldsymbol{\epsilon}^{k+i-1}\right) \;=\; 1 - \sum_{j=1}^{k} P_{\alpha_j} - \sum_{j=1}^{i-1} P_{A_j}\ . \tag{7.12}$$

Proof. Follows directly from Corollary 4. □

The lemma ensures that the following definition is sound. We shall call $ECR_{(k)}(\cdot)$ for the **ECR of a partial troubleshooting sequence**.

Definition 43 (ECR of Optimal Action Sequence)**.** *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be an atomic troubleshooting sequence. For any partial sequence $s[k, m]$ with $1 \leq k \leq m + 1$ the ECR of an optimal compound action sequence consistent with $s[k, m]$ is defined by*

$$ECR^*(s[k,m]) = \begin{cases} \min\limits_{s^c \in \mathcal{CS}(s[k,m])} ECR_{(k-1)}(s^c) & \text{if } k \leq m \\ 0 & \text{if } k = m+1 \end{cases}\ . \tag{7.13}$$

Theorem 15. *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be an atomic troubleshooting sequence. Then for $1 \leq k \leq m$, $ECR^*(s[k, m])$ can be calculate either as*

$$\min_{i=k}^{m} \left(ECR_{(k-1)}(\langle\{\alpha_k, \ldots, \alpha_i\}\rangle) + ECR^*(s[i+1, m]) \right) \tag{7.14}$$

or as

$$\min_{i=k}^{m} \left(ECR^*(s[k, i-1]) + ECR_{(i-1)}(\langle\{\alpha_i, \ldots, \alpha_m\}\rangle) \right). \tag{7.15}$$

Proof. Since the two cases are symmetric (that is, one reuses the ECR for the remaining subsequence and the other reuses the ECR for the already performed subsequence), then we need only consider Equation 7.14. We use induction in $m - k$. Basis step: $m - k = 0$. Since there is only one compound sequence consistent with $\langle \alpha_k \rangle$, we get

$$ECR^*(s[k,m]) = \min\left(ECR_{(k-1)}(\langle\{\alpha_k\}\rangle) + 0\right) = ECR_{(k-1)}(\langle\{\alpha_k\}\rangle)$$

so the theorem holds in this case. Induction step: assume the theorem holds for $m - k \leq x$ and let $m - k = x + 1$. Then let

$$s^* = \underset{s^c \in \mathcal{CS}(s[k,m])}{\arg\min} ECR_{(k-1)}(s^c)$$

for which we must prove

$$ECR^*(s^*) = \min_{i=k}^{m} \left(ECR_{(k-1)}(\langle\{\alpha_k, \ldots, \alpha_i\}\rangle) + ECR^*(s[i+1,m]) \right).$$

By Lemma 15 it is correct to split the ECR into two terms corresponding to an ordered partition of s^*. Furthermore, by the induction hypothesis we can compute $ECR^*(s[i+1,m])$ for any i since $m-(i+1) \leq x$. Then consider that s^* must start with a compound action consisting of $y \in \{1, 2, \ldots, x+1\}$ actions. These choices of the first action correspond exactly to the expressions that we minimize over, and the result follows. □

Theorem 15 shows that the problem of finding an optimal partition of a seed sequence s requires optimal partitions of smaller partial sequences (the **optimal substructure property**) and that these smaller partial sequences are needed by the computation of several bigger sequences (the **overlapping subproblems property**). These two properties enables an *efficient* dynamic programming algorithm, that is, an algorithm that only computes solutions to subproblems once by storing the results in a table. Algorithm 15 uses this approach to find the optimal partition of a seed-sequence s. The algorithm runs in $\Theta(|\mathcal{A}|^3)$ time and uses $\Theta(|\mathcal{A}|^2)$ space. The correctness of the algorithm follows from the above theorem. The example below describes the steps of the algorithm in detail.

Example 20 (Algorithm 15)**.** *We consider the model from Example 17 on page 137 where we shall find the best partitioning of the subsequence $\langle \alpha_1, \alpha_2, \alpha_3 \rangle$. The main work is done by the procedure* `FillCell(·)` *which updates two matrices storing the ECR and the index that induces the optimal partitioning, respectively. Assume the seed-sequence is $s = \langle \alpha_1, \alpha_2, \alpha_3, \alpha_4 \rangle$, then we may visualize the matrices as one by the following:*

Algorithm 15 Optimal Partition into Compound Actions

```
 1: procedure FillCell(row, col, s, C_W, &ecrMatrix, &prevMatrix)
Require: row > col
 2:     Let N = row − col
 3:     Let A = ⟨α_col, ..., α_col+N⟩ from s
 4:     Let ecr = P(ε^(col−1)) · C_A
 5:     Set ecrMatrix[row, col] = ecr
 6:     Set prevMatrix[row, col] = col
 7:     for prev = col + 1 to row − 1 do
 8:         Let ecr′ = ecrMatrix[prev, col] + ecrMatrix[row, prev]
 9:         if ecr′ < ecr then
10:             Set ecrMatrix[row, col] = ecr′
11:             Set prevMatrix[row, col] = prevMatrix[row, prev]
12:         end if
13:     end for
14: end procedure
15: function OptimalPartition(s = ⟨α_1, ..., α_n⟩, C_W)
16:     Let N = |s|
17:     Let ecrMatrix = {0}
18:     Let prevMatrix = {0}
19:     for row = 1 to N do
20:         for col = row − 1 down to 1 do
21:             FillCell(row, col, s, C_W, ecrMatrix, prevMatrix)
22:         end for
23:     end for
24:     Let s* = ⟨⟩
25:     Let prev = N
26:     repeat
27:         Let i = prevMatrix[prev, 1]
28:         Let A = ⟨α_i+1, ..., α_prev⟩
29:         Set s* = ⟨A, s*⟩
30:     until prev < 1
31:     Assert(ECR(s*) = ecrMatrix[N, 1])
32:     return s*
33: end function
```

$$\begin{pmatrix} (0,0) & (0,0) & (0,0) & (0,0) & (0,0) \\ (2,1) & (0,0) & (0,0) & (0,0) & (0,0) \\ (5,1) & (3.04,2) & (0,0) & (0,0) & (0,0) \\ (0,0) & (0,0) & (0,0) & (0,0) & (0,0) \\ (0,0) & (0,0) & (0,0) & (0,0) & (0,0) \end{pmatrix}$$

Here the procedure is paused after the third row has been calculated. Entries in

the diagonal and above are all zero, and so are entries that have not been calculated yet. In the first column is stored the optimal solution when going from having performed zero actions to $1, 2, 3, 4$ *actions. In the second column is stored the optimal solution when going from having performed* 1 *action to* $2, 3, 4$ *actions, etc. At index* $(3, 1)$ *is currently stored the optimal solution over the two first actions. This solution has an ECR of* 5 *and the sequence of compound actions that has this ECR is simply* $\langle A_1 \rangle = \langle \alpha_1, \alpha_2 \rangle$ *because the index 1 tells us that we must jump to the first row.*

The matrices are update one row at a time stating from the right. We shall now describe how the fourth row is updated. This happens in line 20-22 in Algorithm 15.

$Row = 4, col = 3$*: in* `FillCell` *we first form the compound action* $A = \langle \alpha_3 \rangle$ *and calculate its contribution the ECR (line 2-4). The for-loop in line is then not entered, and the matrices now looks as follows:*

$$\begin{pmatrix} (0,0) & (0,0) & (0,0) & (0,0) & (0,0) \\ (2,1) & (0,0) & (0,0) & (0,0) & (0,0) \\ (5,1) & \textit{(3.04,2)} & (0,0) & (0,0) & (0,0) \\ (0,0) & (0,0) & \textit{(0.68,3)} & (0,0) & (0,0) \\ (0,0) & (0,0) & (0,0) & (0,0) & (0,0) \end{pmatrix}$$

$Row = 4, col = 2$*: as usual* `FillCell` *first computes the contribution to the ECR when going directly from index 2 to 4 (which corresponds to performing the compound action* $\langle \alpha_2, \alpha_3 \rangle$*). The for-loop in line 7-13 is then entered one time. The loop body investigates alternative ways to perform* $\langle \alpha_2, \alpha_3 \rangle$*; in this case there is only one alternative: to perform* α_2 *and* α_3 *independently. In line 8 the ECR contribution of this path is calculated by looking up already calculated values; in this case the values highlighted in italics above are added. This turns out to be the better than performing both actions before the system test and therefore we also propagate the index in line 11. The matrices now looks this way:*

$$\begin{pmatrix} (0,0) & (0,0) & (0,0) & (0,0) & (0,0) \\ \textit{(2,1)} & (0,0) & (0,0) & (0,0) & (0,0) \\ \textbf{\textit{(5,1)}} & (3.04,2) & (0,0) & (0,0) & (0,0) \\ (0,0) & \textit{(3.80,3)} & \textbf{\textit{(0.68,3)}} & (0,0) & (0,0) \\ (0,0) & (0,0) & (0,0) & (0,0) & (0,0) \end{pmatrix}$$

Notice that the for-loop in line 7-13 only investigates a linear number of alternative paths even though an exponential number of such paths exists. However, this is enough to find the optimal path since we have already computed all relevant optimal sub-paths and can exploit them in the loop.

$Row = 4, col = 1$*: this time* `FillCell` *computes the cost of the direct path (as usual), and then compare this cost with sum of the non-bold-font italics cells and the sum of the bold-font italics cells (see matrices above). The non-bold-font cells correspond to the sequence* $\langle \alpha_1, \alpha_2, \alpha_3 \rangle$ *and the bold-font cells correspond to*

Algorithm 16 Optimized Exhaustive Search

```
1: function OPTIMALCOMPOUNDACTIONSEQUENCE2(𝒜)
2:     Let bestEcr = ∞
3:     Let bestSeq = ⟨⟩
4:     for all n! permutations of 𝒜  s do
5:         Let s′ = OPTIMALPARTITIONING(s)
6:         if ECR(s′) < bestEcr then
7:             Set bestEcr = ECR(s′)
8:             Set bestSeq = s′
9:         end if
10:    end for
11:    return bestSeq
12: end function
```

the sequence $\langle A, \alpha_3 \rangle$*. The latter sequence has the lowest ECR, so we update the matrices as follows:*

$$\begin{pmatrix} (0,0) & (0,0) & (0,0) & (0,0) & (0,0) \\ (2,1) & (0,0) & (0,0) & (0,0) & (0,0) \\ (5,1) & (3.04,2) & (0,0) & (0,0) & (0,0) \\ (5.68,3) & (3.80,3) & (0.68,3) & (0,0) & (0,0) \\ (0,0) & (0,0) & (0,0) & (0,0) & (0,0) \end{pmatrix}$$

From this we conclude that the optimal way to perform the first three actions is $\langle A, \alpha_3 \rangle$*.*

Algorithm 15 now also means that we can make a more efficient exhaustive search for the best troubleshooting sequence. The algorithm runs in $\Theta(n^3 \cdot n!)$ time and is shown in Algorithm 16.

It should not be too difficult to extend Theorem 15 such it can be used for computing the optimal partitioning even if the model has dependent actions. In fact, we may see the theorem as a consequence of Proposition 7 on page 79 and the additivity of the ECR of action sequences.

Corollary 5. *Algorithm 15 also computes an optimal partitioning of a sequence of dependent actions if we compute the probability of the evidence via the updating P-over-C algorithm (Algorithm 9 on page 91).*

Furthermore, we can immediately construct a new heuristic based on Algorithm 15. The most obvious is to take a seed-sequence where the actions are sorted by efficiency or P-over-C—this is exactly what Algorithm 17 does. The following example shows a case where Algorithm 17 does not find an optimal troubleshooting sequence.

Algorithm 17 Action Sequence with Optimal Partition

1: **function** COMPOUNDACTIONSEQUENCE($\mathcal{A}$)
2: Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ such that $ef(\alpha_i) \geq ef(\alpha_{i+1})$ or $\frac{P_{\alpha_i}}{C_{\alpha_i}} \geq \frac{P_{\alpha_{i+1}}}{C_{\alpha_{i+1}}}$
3: Let s = OPTIMALPARTITION(s)
4: **return** s
5: **end function**

Example 21 (Algorithm 17 is not optimal)**.** *Consider the model from Example 19, but take $C_W = 2$:*

	P_α	C_α	$C_{\alpha+W}$	$ef(\alpha)$	$\frac{P_\alpha}{C_\alpha}$
α_1	0.15	1	3	0.05	0.150
α_2	0.35	2	4	0.0875	0.175
α_3	0.50	3	5	0.1	0.167

For the $ef(\cdot)$ ordering, the algorithm constructs the following tables—here visualized as one table with a pair of values $(ECR^*(s[k,m]), i)$ where $i-1$ is the split that minimized the (partial) ECR:

$$\begin{pmatrix} (0,0) & (0,0) & (0,0) & (0,0) \\ (5,1) & (0,0) & (0,0) & (0,0) \\ (7,1) & (2,2) & (0,0) & (0,0) \\ (7.45,3) & (2.45,3) & (0.45,3) & (0,0) \end{pmatrix}$$

Looking at the bottom left entry, we get that the last action is $\{\alpha_1\}$ and then looking at the entry above it, that the first action is $\{\alpha_3, \alpha_2\}$ and $\langle\{\alpha_3, \alpha_2\}, \alpha_1\rangle$ has ECR 7.45. Even though the initial $\frac{P_\alpha}{C_\alpha}$ ordering is different, the algorithm returns the same troubleshooting sequence (because it forms the same initial compound action). However,

$$ECR\left(\langle\{\alpha_1, \alpha_3\}, \alpha_2\rangle\right) = 6 + 0.35 \cdot 4 = 7.4$$

which shows that the algorithm does not lead to an optimal sequence.

7.4 A Heuristic for The General Problem

We have seen that by changing the order of the atomic actions, we could improve the ECR compared to keeping the ordering fixed. We shall therefore consider the orthogonal task of optimizing the ordering given that the partitioning is fixed.

Definition 44 (Partition Equivalent Sequences)**.** *Let $s^c = \langle A_1, \ldots, A_\ell \rangle$ and $s^d = \langle B_1, \ldots, B_\ell \rangle$ be two compound action sequences of the same size. Then s^c and s^d are **partition equivalent** if*

$$|A_i| = |B_i| \ \ \forall i \in \{1, 2, \ldots, \ell\}. \tag{7.16}$$

Lemma 16. *Let $s^c = \langle \ldots, A_x, \ldots, A_y, \ldots \rangle$ be partition equivalent with $s^d = \langle \ldots, B_x, \ldots, B_y, \ldots \rangle$ where both are compound action sequences. Furthermore, let the only difference between s^c and s^d be that two actions $\alpha \in A_x$ and $\beta \in A_y$ have been swapped such that $\beta \in B_x$ and $\alpha \in B_y$. Then*

$$\begin{aligned} ECR\,(s^c) - ECR(s^d) &= (C_\alpha - C_\beta) \cdot \left[P(\beta) - P(\alpha) + \sum_{i=x}^{y-1} P(A_i) \right] \\ &\quad + [P(\beta) - P(\alpha)] \cdot \sum_{i=x+1}^{y} C_{A_i+W} \ . \end{aligned} \tag{7.17}$$

Proof. Let s^c and s^d be given as above. We then have

$$ECR(s^c) = \sum_{i=1}^{\ell} C_{A_i+W} \cdot \left(1 - \sum_{j=1}^{i-1} P(A_j) \right) = \sum_{i=1}^{\ell} C_{A_i+W} \cdot a_i$$

$$ECR(s^d) = \sum_{i=1}^{\ell} C_{B_i+W} \cdot \left(1 - \sum_{j=1}^{i-1} P(B_j) \right) = \sum_{i=1}^{\ell} C_{B_i+W} \cdot b_i \ .$$

The difference between these two numbers can be expressed as

$$ECR(s^c) - ECR(s^d) = \Delta X + \Delta I + \Delta Y \tag{7.18}$$

where

- ΔX is the difference caused by changes to C_{A_x},
- ΔI is the difference caused by a_i changing into b_i for all compound actions between A_x and A_y, and
- ΔY is the difference causes by changes to C_{A_y} and a_y.

For $i \in \{1, \ldots, x-1, y+1, \ldots, \ell\}$ the terms cancel out. We have

$$\Delta X = a_x \cdot (C_{A_x+W} - C_{B_x+W}) = a_x \cdot (C_\alpha - C_\beta) \ .$$

Furthermore, since $b_i = a_i + P(\alpha) - P(\beta) \forall\ i \in \{x+1, \ldots, y\}$:

$$\Delta I = \sum_{i=x+1}^{y-1} (a_i - b_i) \cdot C_{A_i+W} = [P(\beta) - P(\alpha)] \cdot \sum_{i=x+1}^{y-1} C_{A_i+W}$$

Finally, we have

$$\begin{aligned} \Delta Y &= a_y \cdot C_{A_y+W} - b_y \cdot C_{B_y+W} \\ &= a_y \cdot C_{A_y+W} - b_y \cdot (C_{A_y+W} - C_\beta + C_\alpha) \\ &= (a_y - b_y) \cdot C_{A_y+W} - b_y \cdot (C_\alpha - C_\beta) \\ &= [P(\beta) - P(\alpha)] \cdot C_{A_y+W} - b_y \cdot (C_\alpha - C_\beta) \ . \end{aligned}$$

Now observe that

$$b_y = a_y - [P(\beta) - P(\alpha)] = \left(a_x - \sum_{i=x}^{y-1} P(A_i) \right) - [P(\beta) - P(\alpha)]$$

which implies that

$$\begin{aligned} \Delta Y &= [P(\beta) - P(\alpha)] \cdot C_{A_y+W} \\ &\quad - \left[a_x - P(\beta) + P(\alpha) - \sum_{i=x}^{y-1} P(A_i) \right] \cdot (C_\alpha - C_\beta) \, . \end{aligned}$$

Inserting these three results into Equation 7.18 yields

$$ECR(s^c) - ECR(s^d) = (C_\alpha - C_\beta) \cdot \left[P(\beta) - P(\alpha) + \sum_{i=x}^{y-1} P(A_i) \right]$$

$$+ [P(\beta) - P(\alpha)] \cdot \sum_{i=x+1}^{y} C_{A_i+W}$$

as required. □

Theorem 16. *Let s^c, α and β be given as in Lemma 16. Then the ECR of s^c can be improved by swapping α and β if and only if*

$$\begin{aligned} (C_\alpha - C_\beta) \cdot \left[P(\beta) - P(\alpha) + \sum_{i=x}^{y-1} P(A_i) \right] & \\ > \; [P(\alpha) - P(\beta)] \cdot \sum_{i=x+1}^{y} C_{A_i+W} \, . & \end{aligned} \tag{7.19}$$

Proof. We should swap the two actions if

$$ECR(s^c) - ECR(s^d) > 0$$

$$\Updownarrow$$

$$(C_\alpha - C_\beta) \cdot \left[P(\beta) - P(\alpha) + \sum_{i=x}^{y-1} P(A_i) \right] > [P(\alpha) - P(\beta)] \cdot \sum_{i=x+1}^{y} C_{A_i+W}$$

by Lemma 16. □

Proposition 11. *Theorem 16 is a generalization of the*

$$ef(\alpha_i) \geq ef(\alpha_{i+1}) \; \forall i \in \{1, \ldots, n-1\} \tag{7.20}$$

optimality condition for atomic action sequences (with independent actions).

Proof. When s^c and s^d are atomic action sequences, $\alpha = A_x, \beta = A_y$, and so $|A_x| = |A_y| = 1$. If α and β are neighbours we have $x + 1 = y$. Theorem 16 then yields

$$
\begin{aligned}
& (C_\alpha - C_\beta) \cdot [P(\beta) - P(\alpha) + P(\alpha)] > [P(\alpha) - P(\beta)] \cdot (C_\beta + C_W) \\
\Updownarrow & \\
& P(\beta) \cdot (C_\alpha + C_W) > P(\alpha) \cdot (C_\beta + C_W) \\
\Updownarrow & \\
& ef(\beta) > ef(\alpha)
\end{aligned}
$$

as required. □

It turns out that in certain cases we can avoid the full analysis of Theorem 16.

Proposition 12. *Let s^c, α and β be given as in Lemma 16. If*

$$P(\beta) > P(\alpha) \quad \text{and} \quad C_\alpha \geq C_\beta, \ \text{or} \tag{7.21}$$

$$P(\beta) = P(\alpha) \quad \text{and} \quad C_\alpha > C_\beta \tag{7.22}$$

then $ECR(s^c)$ can be improved by swapping α and β.

Proof. If $P(\beta) > P(\alpha)$ and $C_\alpha \geq C_\beta$, then $P(\beta) - P(\alpha) < 0$, but $C_\alpha - C_\beta \geq 0$. In Theorem 16 the two other factors are strictly positive, so the left hand side becomes non-negative while the right hand side becomes negative making the inequality true. If $P(\beta) = P(\alpha)$ and $C_\alpha > C_\beta$, then the left hand side is strictly positive while the right hand side is zero which also makes the inequality true. □

Based on the above results, we can easily search for a better partition equivalent troubleshooting sequence. The procedure is given in Algorithm 18. Arguably, after the algorithm has terminated, it could be that more actions could be swapped to improve the ECR. However, we have been unsuccessful in proving the number of possible swaps. Therefore we prefer an algorithm that terminates deterministically. Furthermore, we tried to run the procedure until no more swaps could be made on our test models, and we found that it did not improve the ECR compared to Algorithm 18.

It should be clear that the algorithm runs in $O(|\mathcal{A}|^3)$ time. The reason it cannot run in $O(|\mathcal{A}|^2)$ time is that checking Theorem 16 takes linear time on average, albeit when Proposition 12 applies it only takes constant time. Because any swapping invalidates the involved sums, it is not possible to precompute these as with the greedy approaches we saw earlier. We can also formulate the improved heuristic described in Algorithm 19.

Algorithm 18 Optimizing The Partition by Swapping

procedure OPTIMIZEPARTITION(&s^c=$\langle A_1, \ldots, A_n \rangle$)
  for $x = 1$ to n **do**
    for $\alpha \in A_x$ **do**
      for $y = x + 1$ to n **do**
        for $\beta \in A_y$ **do**
          if $(C_\alpha - C_\beta) \cdot \left[P(\beta) - P(\alpha) + \sum_{i=x}^{y-1} P(A_i) \right] >$
            $[P(\alpha) - P(\beta)] \cdot \sum_{i=x+1}^{y} C_{A_i+W}$
          then
            SWAP(α, β)
          end if
        end for
      end for
    end for
  end for
end procedure

Algorithm 19 Action Sequence with Optimized Partition

function COMPOUNDACTIONSEQUENCE2($\mathcal{A}$)
  Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ such that $ef(\alpha_i) \geq ef(\alpha_{i+1})$ or $\frac{P_{\alpha_i}}{C_{\alpha_i}} \geq \frac{P_{\alpha_{i+1}}}{C_{\alpha_{i+1}}}$
  Let s = OPTIMALPARTITION(s)
  OPTIMIZEPARTITION(s)
  return s
end function

7.5 Non-reordable Models

Finding a small example that shows a case where Algorithm 19 is not admissible is not completely trivial. By means of a computer we did construct such examples where the algorithm is not guaranteed to find optimal troubleshooting sequences. However, for certain models we can prove that optimality is ensured.

Definition 45 (Non-reordable Model)**.** *Consider a troublehooting model with independent actions and non-trivial system test cost. If the actions of this model can be ordered into the sequence $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ such that*

$$P_{\alpha_i} \geq P_{\alpha_{i+1}} \text{ and } C_{\alpha_i} \leq C_{\alpha_{i+1}} \quad \text{for } i \in \{1, 2, \ldots, n-1\}\,, \tag{7.23}$$

*then we call this model for a **non-reordable model**.*

Theorem 17. *Let $s = \langle \alpha_1, \ldots, \alpha_n \rangle$ be an atomic troubleshooting sequence in a non-reordable model such that $ef(\alpha_i) \geq ef(\alpha_{i+1})$. Then there exists an optimal troubleshooting sequence s^c consistent with s.*

Proof. Let $s^d = \langle B_1, \ldots, B_\ell \rangle$ and $s^c = \langle A_1, \ldots, A_\ell \rangle$ be partition equivalent compound troubleshooting sequences where s^d is optimal and s^c is consistent with s. Now assume s^d is not consistent with s. Since $s^c \neq s^d$, we can find the index of the first pair of compound actions A_x and B_x where the two sequences differ. Then at least one action $\beta \in A_x$, but $\beta \notin B_x$ and at least one action $\alpha \in B_x$, but $\alpha \notin A_x$. Furthermore, α must be found in A_y with $y > x$. Since s^c is consistent with s and since the model is non-reordable

$$P_\beta \geq P_\alpha \text{ and } C_\beta \leq C_\alpha \tag{7.24}$$

We now consider two cases.

Case 1: at least one of the inequalities in Equation 7.24 is strict. Then if we apply Theorem 16 on s^d we find that we can improve the ECR of s^d by swapping α and β. However, this is a contradiction to s^d being optimal. Hence in this case any optimal sequence must be consistent with s.

Case 2: we have equality in Equation 7.24. In that case we can safely swap α and β in s^d without altering the ECR. If we now have $s^d = s^c$, then we are done. If not, we can repeat the procedure until $s^d = s^c$. Since each swap puts at least one atomic action into its rightful compound action, the procedure terminates in a finite number of steps, thus proving that there exists an optimal sequence consistent with s. □

Corollary 6. *Algorithm 17 and 19 find an optimal troubleshooting sequence in non-reorderable models.*

Proof. Both algorithms calls `OptimalPartition`$(\cdot)$ after having sorted the actions with respect to descending efficiency. □

At this point we shall stop our search for improving the heuristics. A natural extension is to derive a result that states when it is beneficial to move a single atomic action from one compound action to another. Experimental knowledge shows that this can indeed be beneficial, but as we shall see in the next section, the heuristics above already perform quite well.

7.6 Empirical Evaluation

In this section we shall investigate the performance of the heuristic algorithms. In particular, we investigate how the following two model parameters affect the precision:

1. The closeness of the initial efficiency of actions.
2. The closeness of the cost of the actions.

Table 7.1: Model 1. Close efficiencies + close costs.

	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8
P_α	0.21	0.17	0.19	0.155	0.185	0.139	0.17	0.135
C_α	1.8	1.4	1.7	1.3	1.6	1.2	1.5	1.1
$ef(\alpha)$	0.086	0.090	0.083	0.088	0.085	0.085	0.084	0.091

Table 7.2: Model 2. Close efficiencies + non-close costs.

	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8
P_α	0.36	0.205	0.32	0.155	0.28	0.11	0.25	0.05
C_α	8	4	7	3	6	2	5	1
$ef(\alpha)$	0.026	0.030	0.026	0.030	0.027	0.031	0.029	0.029

Table 7.3: Model 3. Non-close efficiencies + close costs.

	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8
P_α	0.4	0.105	0.52	0.055	0.78	0.13	0.6	0.02
C_α	1.8	1.4	1.7	1.3	1.6	1.2	1.5	1.1
$ef(\alpha)$	0.085	0.029	0.117	0.016	0.187	0.042	0.153	0.007

Table 7.4: Model 4. non-close efficiencies + non-close costs.

	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8
P_α	0.16	0.15	0.05	0.165	0.165	0.159	0.07	0.165
C_α	8	4	7	3	6	2	5	1
$ef(\alpha)$	0.018	0.035	0.007	0.051	0.025	0.073	0.013	0.152

Table 7.5: Percent-wise deviations for model 1. The model has close efficiencies + close costs (5828 runs.)

	$ef(\alpha_i)$	Alg. 14	Alg. 13	Alg. 17	Alg. 19
min	0	0	0/0	0/0	0/0
max	128.3	45.6	4.3/2.8	1.5/0.7	1.1/0.6
mean	73.1	10.1	1.7/0.7	0.8/0.1	0.2/0.02
median	79.4	6.0	1.5/0.4	0.8/0	0.1/0
% optimal	1.5	0.1	1.5/26.6	1.5/62.1	39.9/84.8

Table 7.6: Percent-wise deviations for model 2. The model has close efficiencies + non-close costs (4201 runs).

	$ef(\alpha_i)$	Alg. 14	Alg. 13	Alg. 17	Alg. 19
min	0	0	0/0	0 /0	0/0
max	97.2	38.9	6.2/2.3	5.1/0.5	4.2/0.5
mean	54.7	9.0	2.8/0.3	1.9/0.1	1.1/0.0
median	58.9	5.7	3.2/0.1	1.8/0.0	1.2/0
% optimal	0.33	0.1	8.4/32.9	9.5/46.0	18.6/63.9

Due to the difficulty of finding an optimal solution, we could not investigate models with more than 8 actions. In the models below the repair probabilities have not been normalized to make the specification accurate (but they are of course normalized at runtime) and the efficiency stated is the efficiency rounded to three decimals when $C_W = 0$.

For each of the four initial models in Table 7.1 to Table 7.4 we ran all the algorithms for increasing values of C_W. The following observation shows when it does not make sense to increase C_W anymore.

Proposition 13. *Once C_W is made so high that the optimal sequence consists of one compound action, then increasing C_W can never change the optimal sequence.*

The increment in C_W was determined as one permille of the largest cost of any action in the model. Starting from $C_W = 0$ we kept sampling until the optimal sequence consists of only one compound action. Table 7.5 to Table 7.8 summarize the result for the four models where each column describes the results of a given algorithm. The first four rows summarize the percent-wise deviation from the optimal ECR, and the last row shows the percent of cases where the algorithm was optimal. The number of increments to C_W is given in the parenthesis after the model name. The first algorithm column is the simple $ef(\alpha_i) \geq ef(\alpha_{i+1})$ sorting. A 0.0 indicates a number close to zero whereas 0 really means zero. For Algorithm 13,

Table 7.7: Percent-wise deviations for model 3. The model has non-close efficiencies + close costs (79145 runs).

	$ef(\alpha_i)$	Alg. 14	Alg. 13	Alg. 17	Alg. 19
min	0	0	0/0	0/0	0/0
max	149.5	3.9	2.8/2.8	0/0	0/0
mean	124.3	0.1	0.2/0.2	0/0	0/0
median	137.1	0	0/0	0/0	0/0
% optimal	0.45	76.3	73.9/73.9	100/100	100/100

Table 7.8: Percent-wise deviations for model 4. The model has non-close efficiencies + non-close costs (18085 runs).

	$ef(\alpha_i)$	Alg. 14	Alg. 13	Alg. 17	Alg. 19
min	0	0	0/0	0/0	0/0
max	219.1	4.9	2.6/2.6	0/0	0/0
mean	162.8	0.4	0.2/0.2	0/0	0/0
median	181.0	0	0/0	0/0	0/0
% optimal	0.3	53.1	76.1/76.1	100/100	100/100

17, and 19 we give two numbers. The first corresponds to an initial $ef(\cdot)$ ordering and the second corresponds to an initial $\frac{P_\alpha}{C_\alpha}$ ordering. We can summarize the results as follows:

1. Algorithm 19 is by far the best heuristic. Algorithm 14 is the worst of the new heuristics. Algorithm 13 performs surprisingly well, almost as good as Algorithm 17 and would thus be considered the preferred choice for a system under real-time constraints.

2. Models with non-close efficiencies are much easier to solve than models with close efficiencies. Models with non-close costs seem to induce larger deviations from the optimal ECR than models with close costs. We consider none of these findings surprising.

We have also tested the algorithms on a real-world model with 32 actions. The results are summarized in Table 7.9 where the statistics are computed as the percent-wise deviation from the overall best algorithm (Algorithm 19 with $\frac{P_\alpha}{C_\alpha}$ ordering). The -0 indicates that a small percentage of the cases where better than Algorithm 19. Also notice that the apparent conflict between "median" and "% best" in column two is due to rounding of the median.

Table 7.9: Percent-wise deviation for model 5 (relative to best heuristic). The model is a real-world model with 32 actions (33145 runs).

	$ef(\alpha_i)$	Alg. 14	Alg. 13	Alg. 17	Alg. 19
min	0	0	0/0	−0/0	−0
max	667.3	3.8	25.3/19.1	10.7/0.1	6.1
mean	555.3	0.1	5.9/3.1	3.3/0	1.5
median	598.7	0	3.7/1.3	3.6/0	1.3
% best	0.0	48.2	0.0/0	0.0/99.4	0.0

The conclusion here is the same with one noticeable difference: Algorithm 14 is now the third best algorithm, only beaten by Algorithm 17 and 19 with $\frac{P_\alpha}{C_\alpha}$ ordering and far better than any other heuristic. The fact that Algorithm 17 and 19 are very close to each other also suggests that we are quite close to the optimal value.

7.7 Miscelaneous Remarks

In this section we shall discuss a few theoretical properties of troubleshooting with postponed system test. We first investigate if the optimal expected cost $ECR^*(\cdot)$ is monotone when viewed as a function of C_W.

If we keep all other parameters constant, we can regard the optimal ECR as a function of the system test cost, which we shall write $ECR^*(C_W)$. For a particular compound troubleshooting sequence s we write $ECR\,(s; C_W)$ instead, and so $ECR^*(C_W) = ECR\,(s^*; C_W)$. The function takes the general form

$$ECR\,(s; C_W) \;=\; \sum_{i=1}^{\ell}(C_{A_i} + C_W) \cdot (1 - \sum_{j=1}^{i-1} P_{A_j}) \tag{7.25}$$

where $s = \langle A_1, \ldots, A_\ell \rangle$ is a compound troubleshooting sequence.

Remark 28. *The optimal sequence s^* is itself a function of C_W.*

Proposition 14. *Let s be a compound troubleshooting sequence. Then the expected cost $ECR\,(s; C_W)$ is a strictly increasing function of C_W.*

Proof. This is clear immediately from Equation 7.25. □

Proposition 15. *$ECR^*(C_W)$ is a strictly increasing function of C_W.*

Proof. Let $C_W = x$, and let s^* be the optimal sequence for this value. Then set $C_W = y > x$ such that a new sequence $s^{**} \neq s^*$ is now optimal. The question is now: can

$$ECR\,(s^*; x) \geq ECR\,(s^{**}; y)$$

? If so, then $ECR^*(C_W)$ could not be strictly monotone. But since

$$ECR\,(s^{**}; y) > ECR\,(s^{**}; x) \geq ECR\,(s^*; x)$$

this is impossible. □

This leads immediately to the following result.

Theorem 18. *The P-over-C algorithm on a model with $C_W = 0$ computes a lower bound on $ECR^*(\cdot)$ for the same model with $C_W > 0$. Furhermore, for any evidence $\boldsymbol{\epsilon}$, we can add $P(\boldsymbol{\epsilon}) \cdot C_W$ to this this lower bound and still have a lower bound.*

Proof. The first part follows from Proposition 15. The second part follows from the fact that at any point after we have performed the system test, we have to perform it at least once more if there are any more actions to perform. (More formally, one may rewrite the ECR as the sum of (a) the ECR of the same sequence, but with a lower C_W and (b) $P(\boldsymbol{\epsilon}) \cdot C_W$). □

As a consequence we may now use this lower bound in an A* like algorithm or in depth-first search. (Lín, 2011) describes a bottom-up dynamic programming algorithm that uses in $\Theta(2^{|\mathcal{A}|})$ time and memory. It appears to us that it should possible to make a classical top-down search that runs in $O(2^{|\mathcal{A}|})$ by applying coalescing. However, since the evidence set can now also contain actions that are performed, but not tested, it is not obvious that the decision tree is of $O(2^{|\mathcal{A}|})$ size. Since we perform the system test at $O(2^{|\mathcal{A}|})$ nodes in the search graph (corresponding to the different ways that we can form a compound action in) and since these nodes corresponds to the same nodes that we normally apply coalesing at, then the memory requirement *might* stay at $O(2^{|\mathcal{A}|})$, but this is not obvious. (It might be enough to generate one extra successor node for each decision node; this successor node would be the system test.) Furthermore, it must be possible to perform a classical search that exploits the following:

(a) Coalescing is applied whenever possible.

(b) A tight upper bound is computed initially in $\Theta(|\mathcal{A}|^3)$ time and updated whenever we explore a better solution.

(c) Nodes are pruned whenever path cost plus the lower bound exceeds the upper bound.

(d) Nodes are subject to efficiency-based pruning.

We believe such an approach will be quite efficient as a large number of sequences are very far from the optimal one. Furthermore, we belive efficiency-based pruning works better in this context than with depedent actions.

Next, let us discuss the connection between troubleshooting with postponed system test and troubleshooting cost clusters with inside information (where the cluster must be closed before the system can be tested). It should be very clear that the latter is at least as difficult as the former; for example, imagine a cost cluster model with one empty parent cluster and a non-empty child cluster. Now we face the exact same problem as with a postponed system test.

Now imagine again a cost cluster model with two clusters, but let the parent cluster be non-empty too. We might have hoped that if one could solve the bottom level cluster in isolation, then the order among actions would be unaltered when we merge with the actions from the root cluster. Unfortunately, we have constructed a counter example that shows this is not the case.

7.8 Summary

In this chapter we have extended a the classical polynomial troubleshooting scenario of independent actions with postponed system test. Albeit this is a somewhat simple extension requiring only minor modifications to the definition of the expected cost of repair, it leads to an NP-hard problem.

First, we presented to two greedy $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time heuristics which were both motivated by theoretical considerations. Then we described a dynamic programming algorithm, which given a seed sequence finds the optimal partitioning of the actions. As an extra benefit, this optimization can also be adopted to models with dependent actions. This algorithm then lead to a powerful $\Theta(|\mathcal{A}|^3)$ time heuristic. To further improve this heuristic we proposed an $O(|\mathcal{A}|^3)$ time post-processing step that swaps actions in partitions to improve the sequence. We furthermore identified a class of non-reordable models where both $\Theta(|\mathcal{A}|^3)$ time algorithms are admissible.

Using a naive (but simple) exhaustive search procedure we benchmarked all heuristics against optimal solutions. For reordable models, we found that the suggested heuristics provide quite close approximations to an optimal solution, especially the most powerful of them. Finally, we ran tests on a medium sized real world model and the algorithms appear to perform very well—even heuristics with an $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ complexity, albeit we are unable to verify optimality for such a large model. However, we described (but did not implement) a search scheme which might make it possible to verify optimality for such medium sized models. Furthermore, for this type of search we developed an admissible heuristic function.

Similar to troubleshooting with dependent actions, it seems that good approximations are relatively easy to find. Therefore we believe future research should be directed towards finding guarantees on the quality of the solutions returned by the heuristics as well as guarantees about local optimality when optimizing a given partition.

Chapter 8

Conclusion

> At any rate it seems that I am wiser than he is to this small extent, that I do not think that I know what I do not know.
> –*Socrates*

Ever since high school I have been fascinated by Socrates' sentence above. It explains the essence of true scientific thinking, albeit it may not be taken very seriously in the non-exact sciences anymore (climate-, medical- and social-sciences spring to mind). Years later my admiration only increased when I learned that probability theory is based on the axiom that we must always take exactly what we know into account, nothing less and nothing more. And now time has finally come to summarize what we know (and do not know) as a result of the work presented in this thesis.

8.1 Contributions

We believe that the contributions of this thesis are the following:

1. In Chapter 3 we have given a detailed account of decision theoretic troubleshooting, and in particular the underlying assumptions and the constraints induced by semantic requirements that we must uphold in a realistic model. This enabled us to give a comparison of the single-fault troubleshooting model with a multi-fault troubleshooting model (with dependent faults). This comparison highlighted several open problems for the multi-fault model and enforced our view of the appropriateness of the single-fault troubleshooting model.

2. Secondly, we have given detailed account of various solution methods in Chapter 4. This can hardly be considered research, but we have put an effort into describing many practical (and several theoretical) details that are often ignored by existing literature. A bit surprising perhaps, we found that AO* does not stand out as a clear winner from a theoretical perspective.

3. Our discussion of troubleshooting with dependent actions (Chapter 5) contains theoretical as well as empirical results. On the theoretical side we determined that the well-known heuristic function by (Vomlelová and Vomlel, 2003) was monotone for models containing only actions. As explained in Chapter 4, we can always construct an equally informed monotone heuristic function from an non-monotone one, and therefore the practical implications are minor.

4. Chapter 5 also contains an empirical evaluation of the A* algorithm and the effect of efficiency-based pruning. Seen in isolation this leads to an exponential drop in complexity, but taken together with coalescing, the effect of efficiency-based pruning only increased the performance by a factor between 2 and 7. Most importantly, perhaps, was the positive effect it had on the memory requirement which for some models was reduced by a factor of 4. Attempts to extend efficiency-based pruning to subsequences of size 3 proved futile, and we were able to explain that this is due to coalescing.

5. But the implementation of A* carried other unforseen benefits. An important lesson was that simple greedy heuristics (e.g., the updating P-over-C algorithm) seem to perform surprisingly well and return a sequence within one percent from an optimal one. Of course, the models in question may be seen as very special because of the high degree of dependency among actions, and so we should gather and generate more models to be more certain about this hypothesis. In particular, we belive models with close efficiencies, but non-close costs are challenging for greedy heuristics. Nevertheless, we dare to conjecture that there exists a polynomial algorithm which is guaranteed to return a close-to-optimal solution.

6. We then continued with a discussion of the intuitive notion of dependency sets. Unfortunately, we came to the conclusion that the conjecture set forth by (Koca and Bilgic, 2004a) was false. This is somewhat tragic because the conjecture, if true, would have lead to a major reduction in the complexity associated with dependent actions. We believe, however, that there might be a more restrictive definition of dependency sets that preserves admissibility, but this is of course an open problem.

7. On the bright side, we had already implemented an A* hybrid approach that took advantage of the conjecture. These experiments showed that only one or two optimal solutions were missed out of 21 models. And we may add that the distance to the optimal solution was very small. Therefore the idea of dependency sets deserves a prominent place in the history of troubleshooting with dependent

actions. The experiments also revealed that the use of dependency sets allow us to solve (albeit, without admissibility) models with an average dependency below 1.6 almost instantly. Our students later pushed this limit to models with an average dependency of 1.9.

8. Another interesting trend revealed by the experiments was that an increased average dependency among actions lead to an easier problem for A*. We speculated that it could be (a) the heuristic function that became more accurate, or (b) perhaps by accident, the entropy of the normalized efficiencies decreased. Explaining this behavior remains an open problem.

9. In Chapter 6 we started the investigation of troubleshooting with conditional costs by looking into the cost cluster framework under the assumption of inside information and independent actions. The first theoretical result was that we could prove that the extended P-Over-C algorithm (Langseth and Jensen, 2001) for solving flat cost cluster models is in fact admissible. The practical importance of this type of model is emphasized by the entire existing literature on troubleshooting with conditional costs.

10. We then generalized the flat cost cluster model with the tree cost cluster model. To our great delight, we were able to derive an optimally efficient ($O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time) algorithm (the bottom-up P-over-C algorithm) and prove that it too returns an optimal troubleshooting sequence.

11. We furthermore sketched how to extend the tree structure to a DAG which could be solved by deriving a set of tree structures. The induced tree cost cluster models can then be solved by applying the bottom-up P-over-C algorithm. This method will work well if the number of clusters with more than one parent is low, and this is probably true for some real-world models. We furthermore described the open problem of troubleshooting with multiple independent faults (Srinivas, 1995) in a cost cluster framework.

12. Finally, in Chapter 7 we investigated heuristics for troubleshooting models with a non-trivial system test cost. Even under the assumption of independent actions and no questions, this is an NP-hard problem. We describe two very efficient heuristics running in $O(|\mathcal{A}| \cdot \lg |\mathcal{A}|)$ time and motivate their greedy choice by theoretical considerations.

13. Then we gave a dynamic programming algorithm that exploited the structure of the troubleshooting sequences to compute the optimal partitioning of a seed-sequence in $\Theta(|\mathcal{A}|^3)$ time. As a bonus, this algorithm can be modified to work also when the actions are dependent. We then suggested two powerful $\Theta(|\mathcal{A}|^3)$ time heuristics based on the dynamic programming algorithm.

14. Theoretical considerations allowed us to identify a new class of troubleshooting models, so-called non-reordable models, where both $\Theta(|\mathcal{A}|^3)$ time algorithms are shown to be admissible.

15. The chapter ends with an empirical evaluation of the proposed heuristics. A naive, but very simple exhaustive search method is used to find the optimal expected cost of repair. The experiments reveal that models with close efficiencies are most challenging, and this is especially true if the costs are non-close at the same time.

16. Overall the heuristics perform quite well, with the best greedy heuristic never being more than 2.8 percent away from the optimal expected cost of repair on the artificial test models. The best $\Theta(|\mathcal{A}|^3)$ time heuristic is never more than 0.6 percent from the optimal value. We also ran experiments with a real-world model, and here the best greedy heuristic was never more than 3.8 percent from the best $\Theta(|\mathcal{A}|^3)$ time heuristic. In such medium sized models we cannot find the optimal value with naive search methods, but we outlined a top-down search procedure based on an admissible heuristic function which might be able to do so.

17. Again, the empirical results indicate that it might be possible to prove that a polynomial time algorithm can find approximations that are guaranteed to be close-to-optimal. With a little luck, it might even turn out to be one of the heuristics we presented in Chapter 7.

18. Finally, because troubleshooting with postponed system test is closely connected to troubleshooting a cost cluster model without inside information, then all the algorithms of Chapter 7 are immediately applicable in the cost cluster domain. Furthermore, since we know that we cannot lift an optimal solution from a child cluster into a parent cluster and preserve admissibility, then we are better off by optimizing over the whole sequence. The heuristics from Chapter 7 may be adopted (extended) for this purpose.

We believe that this is an appropriate mixture of theoretical results and empirical evaluations of troubleshooting with dependent actions and conditional costs.

8.2 Discussion

In more general terms, this thesis has been devoted to two issues: solution methods and heuristics. The sheer complexity of finding an optimal solution often means that we have to give up and settle for an approximate solution. One may argue that we can just solve a troubleshooting model on a big super-computer. However, as soon as the model includes questions, the space requirements of a strategy often means that we cannot even store a solution (optimal or not). Even though there is probably some interesting work to be done on how to compress a strategy most efficiently, we cannot escape the exponential complexity. Furthermore, a decision support system needs to be flexible, and so it should support re-calculations of the next step to perform given alternative information (skipping a step may be handled without problems by picking the best sub-strategy). This means that we are forced to fall back on heuristic methods in almost all real-world situations. The primary purpose of off-line solution methods is then to serve as evaluation tools for our heuristic methods. And as such, solution methods are certainly indispensable.

When it comes to heuristics, then a primary goal is to be really fast. A quadratic or cubic complexity may very well be the limit for what is acceptable. This is because real-world models with a fifty to eighty steps are not uncommon for troubleshooting. Therefore a central theme of this thesis has been to devise heuristics that operate quite efficiently while making as theoretically informed choices as possible.

We have exclusively studied very narrow troubleshooting scenarios which viewed in isolation carry many assumptions that are not present in real-world situations. When we have done so, it is of course to make a theoretical analysis easier. We may ask, what is the advantage of knowing that you can solve a tree cost cluster model in near-linear time when almost any real-world model contains dependent actions and questions. Or what benefit do we have from knowing that heuristics for troubleshooting with postponed system test work well in isolation when a normal model may have cost cluster and questions as well. Well, there are at least two reasons why such isolated results have great practical value.

First, if you do have a real-world model where the strict assumptions apply, you gain an immediate benefit in terms of the computational efficiency. For example, a model with dependent actions may after a few steps have been performed end up being a model with independent actions. Similarly, a solution method may stumble into a sub-problem where several assumptions no longer apply, and it can then reduce the search space tremendously when solving that sub-problem. It was an illustrative lesson when we were able to re-use the P-over-C algorithm to compute a lower-bound for a totally different problem. Results tend to find uses that we did not imagine when we first derived them.

Secondly, it is not hard to imagine real-world problems where we must battle everything from dependent actions, questions, and postponed system test to cost clusters without inside information. This implies that we may face four NP-hard problems mixed together, and this can be seen as a tremendous challenge. The key observation, however, is that solutions and heuristics for very isolated problems provide a solid theoretical foundation on which to build more advanced heuristics that are likely to perform well.

For example, if we were to extend the tree cost cluster model with dependent actions, we can immediate synthesize a powerful algorithm in the following manner: run the updating P-over-C algorithm in each cluster and compute the absorbtion into the root cluster. If we are not satisfied with that result, then again we have other heuristics in our tool box that allows us to improve the result: run a swapping procedure similar to the one used for troubleshooting with postponed system test and let it further optimize the sequence (in this case there would be restrictions on which actions that may be swapped).

To further illustrate the versatility of this approach, we consider another example. Imagine a tree cost cluster model without inside information and with dependent actions. We can immediate suggest that it would be good to do the following in a bottom up fashion: (a) first use the updating P-over-C algorithm in a cluster, (b) then compute a partitioning of the cluster by applying a heuristic for troubleshooting with postponed system test, and (c) then merge the cluster into its parent cluster; then continue until we are left with the root cluster.

The important lesson here is that isolated solution methods and heuristics are often *orthogonal* to each other. We firmly believe that orthogonality is one of the most powerful features when it comes to battling complexity. Of course, at the time of writing, this is really just a conjecture, and whether it holds true for troubleshooting models must be seen as an open problem. We are better off by following Socrates' maxim and admit that there are many things we do not yet know about troubleshooting.

Appendix

Dansk Resumé (Summary in Danish)

Denne Ph.D.-afhandling har følgende danske titel:

> *Løsninger og Heuristikker for Fejlsøgning med Afhængige Handlinger og Betingede Omkostninger.*

Afhandlingen omhandler således en beslutningsteoretisk tilgang til fejlsøgning af komplekse systemer; for eksempel er man interesseret i at reparere en vindmølle hurtigst (og/eller billigst) muligt, når den er gået i stå af ukendte årsager. Den beslutningsteoretiske tilgang består i, at vi beskriver reparationsprocessen ved hjælp af en matematisk model. Forskellige delområder kræver specielt tilpassede matematiske modeller, men fælles for dem alle er, at de kan håndtere usikkerhed omkring reparationshandlinger og baggrundsobservationer. For eksempel, så kan en handling i gennemsnit måske udføres forkert i 10% af tilfældene, og observationer kan give et ikke-entydigt bevis for bestemte fejl. Derfor bliver den datalogiske opgave at finde algoritmer, som nedbringer den gennemsnitlige omkostning ved at reparere systemet. Således omhandler afhandlingen en vifte af løsninger og heuristikker for forskellige delområder af fejlsøgning.

Det første tema i afhandlingen er modeller, hvor handlingerne er afhængige, det vil sige, at flere handlinger kan fjerne de samme fejl. Vores bidrag her består dels i teoretiske resultater vedrørende løsningsmetoderne til dette område og dels i empiriske resultater, hvor nye optimeringsteknikker kombineres med løsningsmetoderne for at forbedre disses svartider.

Det andet tema er modeller med betingede omkostninger. Det første område, vi analyserer, er klynge-modeller, hvor handlinger grupperes i klynger, alt efter om de deler den samme mængde forarbejde. Udfordringen består her i at finde det optimale tidspunkt for at åbne en klynge. Vi giver effektive algoritmer for flade klyngemodeller og modeller, hvor klyngernes indbyrdes forhold beskrives ved hjælp af en træstruktur. For disse modeller beviser vi tilmed, at de foreslåede algoritmer giver et optimalt resultat. Endelig så beskriver vi en række teoretisk velbegrundede heuristikker til at optimere systemer, hvor omkostningen ved at teste systemet ikke er så lille, at den kan ignoreres. Præcisionen af disse heuristikker afdækkes ved en grundig empirisk undersøgelse, og vi finder, at heuristikkerne giver næsten-optimale resultater.

Index

Bibliography

Abdelbar, A. M. and Hedetniemi, S. M. (1998). Approximating MAPs for belief networks is NP-hard and other theorems. *Artificial Intelligence*, 102(1):21 – 38.

Ahlmann-Ohlsen, K., Jensen, F., Nielsen, T., Pedersen, O., and Vomlelová, M. (2009). A comparison of two approaches for solving unconstrained influence diagrams. *International Journal of Approximate Reasoning*, 50(1):153–173.

Bilgic, M. and Getoor, L. (2007). Voila: efficient feature-value acquisition for classification. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 2*, pages 1225–1230. AAAI Press.

Breese, J. and Heckerman, D. (1996). Decision-theoretic troubleshooting: A framework for repair and experiment. In *Proceedings of The Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 124–132. Morgan Kaufmann Publishers.

Changhe Yuan, Xiaojian Wu, E. H. (2010). Solving multistage influence diagrams using branch-and-bound search. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI-10)*, Catalina Island, CA.

Cheeseman, P. (1985). In defense of probability. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1002–1009.

Cohen, R., Finderup, P., Jacobsen, L., Jacobsen, M., Joergensen, M. V., and Moeller, M. H. (2008). Decision theoretic troubleshooting. Technical report, Computer Science Department, Aalborg University.

Cooper, G. F. (1990). The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42(2-3):393 – 405.

Cowell, R. G., Dawid, A. P., Lauritzen, S. L., and Spiegelhalter, D. J. (1999). *Probabilistic Networks and Expert Systems*. Springer.

Cox, R. (1979). Of inference and inquery—an essay in inductive logic. In Levine and Tribus, editors, *In The Maximum Entropy Formalism*. M.I.T. Press.

Crowley, M., Boerlage, B., and Poole, D. (2007). Adding local constraints to Bayesian networks. In *Proceedings of the 20th conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, CAI '07, pages 344–355, Berlin, Heidelberg. Springer-Verlag.

Dagum, P. and Luby, M. (1993). Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence*, 60(1):141 – 153.

Darwiche, A. (2009). *Modelling and Reasoning with Bayesian Networks*. Cambridge University Press.

Dawid, A. P. (1979). Conditional independence in statistical theory. *Journal of the Royal Statistical Society. Series B (Methodological)*, 41(1):pp. 1–31.

Dechter, R. and Mateescu, R. (2004). Mixtures of deterministic-probabilistic networks and their AND/OR search space. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, UAI '04, pages 120–129, Arlington, Virginia, United States. AUAI Press.

Dechter, R. and Pearl, J. (1985). Generalized best-first search strategies and the optimality af A*. *J. ACM*, 32(3):505–536.

Dezide (2010). The picture is provided by Dezide Aps.

Druzdzel, M. J. (1999). SMILE: structural modeling, inference, and learning engine and genie: a development environment for graphical decision-theoretic models. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference*, pages 902–903, Menlo Park, CA, USA.

ExpertMaker (2010). See `http://www.expertmaker.com/`.

GeNIe (1998-2010). Genie 2.0, decision systems laboratory, university of pittsburgh.

Gökcay, K. and Bilgic, T. (2002). Troubleshooting using probabilistic networks and value of information. *Int. J. Approx. Reasoning*, 29(2):107–133.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, SSC-4(2):100–7.

Heckerman, D., Breese, J. S., and Rommelse, K. (1995). Decision-theoretic troubleshooting. *Communications of the ACM*, 38(3):49–57.

Howard, R. A. and Matheson, J. E. (1981). Influence diagrams. In Howard, R. A. and Matheson, J. E., editors, *Readings on the Principles and Applications of Decision Analysis*, pages 721–762.

Jaynes, E. T. (2003). *Probability Theory—The Logic Of Science*. Cambridge University Press.

Jensen, F. V. (1995). Cautious propagation in Bayesian networks. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 323–328. Morgan Kaufmann.

Jensen, F. V., Kjærulff, U., Kristiansen, B., Langseth, H., Skaanning, C., Vomlel, J., and Vomlelová, M. (2001). The SACSO methodology for troubleshooting complex systems. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 15:321–333.

Jensen, F. V. and Nielsen, T. D. (2007). *Bayesian Networks and Decision Graphs, 2nd ed.* Springer.

Jensen, F. V., Nielsen, T. D., and Shenoy, P. P. (2006). Sequential influence diagrams: A unified asymmetry framework. *International Journal of Approximate Reasoning*, 42(1-2):101 – 118.

Jensen, F. V. and Vomlelová, M. (2002). Unconstrained influence diagrams. In *UAI '02, Proceedings of the 18th Conference in Uncertainty in Artificial Intelligence, University of Alberta, Edmonton, Canada*, pages 234–241. Morgan Kaufmann.

Kadane, J. and Simon, H. (1977). Optimal strategies for a class of constrained sequential problems. *The Annals of Statistics*, 5:237–255.

Kaindl, H. and Kainz, G. (1997). Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317.

Kalagnanam, J. and Henrion, M. (1990). A comparison of decision analysis and expert rules for sequential diagnosis. In *UAI '88: Proceedings of the Fourth Conference on Uncertainty in Artificial Intelligence*, pages 271–282, Amsterdam, The Netherlands. North-Holland Publishing Co.

Kim, J. H. and Pearl, J. (1983). A computational model for combined causal and diagnostic reasoning in inference systems. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence IJCAI-83*, pages 190–193, Karlsruhe, Germany.

Kjærulff, U. B. and Madsen, A. L. (2008). *Bayesian Networks and Influence Diagrams: A Guide to Construction and Analysis*. Information Science and Statistics. Springer.

Koca, E. (2003). A generic troubleshooting approach based on myopically adjusted efficiencies. Master's thesis, Bogazici University, Istanbul.

Koca, E. and Bilgic, T. (2004a). A troubleshooting approach with dependent actions. In de Mántaras, R. L. and Saitta, L., editors, *ECAI 2004: 16th European Conference on Artificial Intelligence*, pages 1043–1044. IOS Press.

Koca, E. and Bilgic, T. (2004b). Troubleshooting with questions and conditional costs. *Proceedings of the 13th Turkish Symposium on Artificial Intelligence and Artificial Neural Networks*, pages 271–280.

Kolmogorov, A. N. (1950). *Foundations of the Theory of Probability*. Chealsea Publishing Company, New York.

Korf, R. E. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27:97–109.

Korf, R. E. (1993). Linear-space best-first search. *Artificial Intelligence*, 62(1):41 – 78.

Korf, R. E., Zhang, W., Thayer, I., and Hohwald, H. (2005). Frontier search. *J. ACM*, 52:715–748.

Langseth, H. and Jensen, F. V. (2001). Heuristics for two extensions of basic troubleshooting. In *Proceedings of the Seventh Scandinavian Conference on Artificial Intelligence*, pages 80–89. IOS Press.

Langseth, H. and Jensen, F. V. (2003). Decision theoretic troubleshooting of coherent systems. *Reliability Engineering and System Safety 80 (1), 49-61*.

Lauritzen, S. L. and Nilsson, D. (2001). Representing and solving decision problems with limited information. *Management Science*, 47:1235–1251.

Likhachev, M., Gordon, G., and Thrun, S. (2004). ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems 16: Proceeding of The 2003 Conference (NIPS-03)*. MIT Press.

Lín, V. (2011). Extensions of decision-theoretic troubleshooting: Cost clusters and precedence constraints. In *Proceedings of Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 11th European Conference, ECSQARU 2011, Belfast, UK*, pages 206–216.

Luque, M., Nielsen, T., and Jensen, F. (2008). *An anytime algorithm for evaluating unconstrained influence diagrams*, pages 177–184.

Mahanti, A. and Ray, K. (1987). Heuristic search in networks with modifiable estimate. In *CSC '87: Proceedings of the 15th annual conference on Computer Science*, pages 166–174, New York, NY, USA. ACM.

Mero, L. (1984). A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, 23(1):13–27.

Monniaux, D. (2008). The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30:12:1–12:41.

Morgenstern, O. and Von-Neumann, J. (1947). *Theory of Games and Economic Behavior*. Princeton University Press, 2. edition.

Nielsen, S., Nielsen, T., and Jensen, F. (2007). *Multi-currency Influence Diagrams*, volume 213 of *Studies in Fuzziness and Soft Computing*, pages 275–294.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Morgan Kaufmann.

Nilsson, N. J. (1998). *Artificial intelligence: a new synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Ottosen, T. J. and Jensen, F. V. (2008a). A* wars: The fight for improving A* search for troubleshooting with dependent actions. In *Proceedings of the Fourth European Workshop on Probabilistic Graphical Models*, pages 233–240.

Ottosen, T. J. and Jensen, F. V. (2008b). Better safe than sorry—optimal troubleshooting through A* search with efficiency-based pruning. In *Proceedings of the Tenth Scandinavian Conference on Artificial Intelligence*, pages 92–97. IOS Press.

Ottosen, T. J. and Jensen, F. V. (2010). The cost of troubleshooting cost clusters with inside information. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 409–416. AUAI Press, Corvallis, Oregon 97339.

Ottosen, T. J. and Jensen, F. V. (2011). When to test? troubleshooting with postponed system test. *Expert Systems with Applications*, 38(10):12142 – 12150.

Ottosen, T. J. and Vomlel, J. (2010a). All roads lead to Rome—new search methods for optimal triangulations. In *Proceedings of the Fifth European Workshop on Probabilistic Graphical Models*, pages 201–208.

Ottosen, T. J. and Vomlel, J. (2010b). Honour thy neighbour—clique maintenance in dynamic graphs. In *Proceedings of the Fifth European Workshop on Probabilistic Graphical Models*, pages 209–216.

Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Pearl, J. (1985). Bayesian networks: a model of self-activated memory for evidential reasoning. In *Proceedings, Cognitive Science Society, UC Irvine*, pages 329–334.

Pearl, J. (1986). Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, 29(3):241 – 288.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Pearl, J. (2009). *Causality—Models, Reasoning and Inference*. Cambridge University Press, 2nd edition.

Pernestål, A. (2009). *Probabilistic Fault Diagnosis With Automotive Applications*. PhD thesis, Department of Electrical Engineering, Linköbing University, Sweden.

Pohl, I. (1973). The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the Third International Joint Conference on Artificial Intelligence IJCAI-73*, pages 12–17.

Raghavan, V., Shakeri, M., and Pattipati, K. R. (1999a). Optimal and near-optimal test sequencing algorithms with realistic test models. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 29(1):11–26.

Raghavan, V., Shakeri, M., and Pattipati, K. R. (1999b). Test sequencing algorithms with unreliable tests. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 29(4):347–357.

Raghavan, V., Shakeri, M., and Pattipati, K. R. (1999c). Test sequencing problems arising in test planning and design for testability. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 29(2):153–163.

Raghavan, V., Shakeri, M., and Pattipati, K. R. (2004). On a multimode test sequencing problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 34(3):1490–1499.

Renooij, S. (2010). Bayesian network sensitivity to arc-removal. In *The Fifth European Workshop on Probabilistic Graphical Models (PGM-10)*, Helsinki, Finland.

Rios, L. H. O. and Chaimowicz, L. (2010). A survey and classification of A* based best-first heuristic search algorithms. In *Proceedings of the 20th Brazilian conference on Advances in artificial intelligence*, SBIA'10, pages 253–262, Berlin, Heidelberg. Springer-Verlag.

Rish, I., Brodie, M., Ma, S., Odintsova, N., Beygelzimer, A., Grabarnik, G., and Hernandez, K. (2005). Adaptive diagnosis in distributed systems. *Neural Networks, IEEE Transactions on*, 16(5):1088 –1109.

Shakeri, M., Raghavan, V., Pattipati, K. R., and Patterson-Hine, A. (2000). Sequential testing algorithms for multiple fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 30(1):1–14.

Shimony, S. E. (1994). Finding maps for belief networks is NP-hard. *Artificial Intelligence*, 68(2):399 – 410.

Shpitser, I. and Pearl, J. (2007). What counterfactuals can be tested. In *UAI '07: Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, pages 352–359.

Skaanning, C., Jensen, F. V., and Kjærulff, U. (2000). Printer troubleshooting using Bayesian network. *IEA/AIE-2000*, pages 367–379.

Skaanning, C., Jensen, F. V., Kjærulff, U., and Madsen, A. L. (1999). Acquisition and transformation of likelihoods to conditional probabilities for Bayesian networks. *AAAI Spring Symposium, Stanford, USA*.

Skaanning, C. and Vomlel, J. (2001). Troubleshooting with simultaneous models. *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 6th*.

Smallwood, R. D. and Sondik, E. J. (1973). The optimal control of partially observable markov processes over a finite horizon. *Operations Research*, 21(5):1071–1088.

Srinivas, S. (1993). A generalization of the noisy-or model. In *Proceedings of the Proceedings of the Ninth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-93)*, San Francisco, CA. Morgan Kaufmann.

Srinivas, S. (1995). A polynomial algorithm for computing the optimal repair strategy in a system with independent component failures. In *UAI '95: Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 515–522. Morgan Kaufmann.

Steele, J. M. (2004). *The Cauchy-Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities*. Cambridge University Press, New York, NY, USA.

Stern, R., Kulberis, T., Felner, A., and Holte, R. (2010). Using lookaheads with optimal best-first search. In *AAAI Proceedings*.

Stewart, B. S., Liaw, C.-F., and White, C. C. (1994). A bibliography of heuristic search research through 1992. *IEEE Transactions on Systems, Man, and Cybernetics*, 24:268–293.

Sun, X., Yeoh, W., Chen, P.-A., and Koenig, S. (2009). Simple optimization techniques for A*-based search. In *Proceedings of The 8th International*

Conference on Autonomous Agents and Multiagent Systems - Volume 2, AAMAS '09, pages 931–936, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Tu, F. and Pattipati, K. R. (2003). Rollout strategies for sequential fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 33(1):86–99.

Tu, F., Pattipati, K. R., Deb, S., and Malepati, V. N. (2003). Computationally efficient algorithms for multiple fault diagnosis in large graph-based systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 33(1):73–85.

Vomlel, J. (2004). Building adaptive tests using Bayesian networks. *Kybernetika*, 40(3):333–348.

Vomlelová, M. (2001). *Decision Theoretic Troubleshooting*. PhD thesis, Faculty of Informatics and Statistics, University of Economics, Prague.

Vomlelová, M. (2003). Complexity of decision-theoretic troubleshooting. *Int. J. Intell. Syst.*, 18(2):267–277.

Vomlelová, M. and Vomlel, J. (2003). Troubleshooting: NP-hardness and solution methods. *Soft Computing Journal, Volume 7, Number 5*, pages 357–368.

Warnquist, H. and Nyberg, M. (2008). A heuristic for near-optimal troubleshooting using ao*. In *Proceedings of the International Workshop on the Principles of Diagnosis*.

Warnquist, H., Nyberg, M., and Säby, P. (2008). Troubleshooting when action costs are dependent with application to a truck engine. In *Proceeding of the Tenth Scandinavian Conference on Artificial Intelligence*, pages 68–75, Amsterdam, The Netherlands. IOS Press.

Wellman, M. P. and Henrion, M. (1994). Explaining "explaining away". Technical report.

Wen, W. (1990). Optimal decomposition of belief networks. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence (UAI-90)*, pages 209–224, New York, NY. Elsevier Science.

Xiang, Y. (2010). Acquisition and computation issues with NIN-AND tree models. In *The Fifth European Workshop on Probabilistic Graphical Models (PGM-10)*, Helsinki, Finland.

Yannakakis, M. (1981). Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79.

Yoshizumi, T., Miura, T., and Ishida, T. (2000). A* with partial expansion for large branching factor problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 923–929. AAAI Press.

Yuan, C., Liu, X., Lu, T.-C., and Lim, H. (2009). Most relevant explanation: Properties, algorithms, and evaluations. In *Proceedings of The 25th Conference on Uncertainty in Artificial Intelligence (UAI-09)*, Montreal, Canada.

Zahavi, U., Felner, A., Schaeffer, J., and Sturtevant, N. R. (2007). Inconsistent heuristics. In *AAAI'07*, pages 1211–1216.

Endnu skal siges: Min Søn, va'r dig! Der er ingen ende på, som der skrives bøger, og megen gransken trætter legemet.
–Præd. 12,12.