Tractable Cognition: Complexity Theory in Cognitive Psychology

by

Iris van Rooij
M.A., Katholieke Universiteit Nijmegen, 1998

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Psychology

We accept this dissertation as conforming to the required standard

Supervisors: Dr. Helena Kadlec, Dr. Ulrike Stege

Abstract

This research investigates the import and utility of computational complexity theory in cognitive psychology. A common conception in cognitive psychology is that a cognitive system is to be understood in terms of the function that it computes. The recognition that cognitive systems—being physical systems—are limited in space and time has led to the *Tractable Cognition thesis*: only tractably computable functions describe cognitive systems. This dissertation considers two possible formalizations of the Tractable Cognition thesis. The first, called the P-Cognition thesis, defines tractability as *polynomial-time computability* and is the dominant view in cognitive science today. The second, called the FPT-Cognition thesis, is proposed by the author and defines tractability as *fixed-parameter tractability* for some "small" input parameters. The FPT-Cognition thesis is shown to provide a useful relaxation of the P-Cognition thesis. To illustrate how the FPT-Cognition thesis can be put into practice, a set of simple but powerful tools for complexity analyses is introduced. These tools are then used to analyze the complexity of existing cognitive theories in the domains of coherence reasoning, subset choice, binary-cue prediction and visual matching. Using psychologically motivated examples, a sufficiently diverse set of functions, and simple proof techniques, this manuscript aims to make the theory of classical and parameterized complexity tangible for cognitive psychologists. With the tools of complexity theory in hand a cognitive psychologist can study the *a priori* feasibility of cognitive theories and discover interesting and potentially useful cognitive parameters. Possible criticisms of the Tractable Cognition thesis are discussed and existing misconceptions are clarified.

Examiners:

Dr. H. Kadlec, Supervisor (Department of Psychology)

Dr. U. Stege, Supervisor (Department of Computer Science)

Table of Contents

List of Tables

List of Figures

Acknowledgements

Four years ago I decided to leave the Netherlands and come to Canada to complete my education and to embark upon an interdisciplinary research project at the intersection of cognitive psychology and theoretical computer science. This decision was unexpected, to say the least—not only because I was never much of a traveler, but also because at the time I knew little or nothing about theoretical computer science. It turns out that it was one of the best decisions I ever made. With joy I look back upon my experiences here at the University of Victoria and I am grateful for every day I got to spend in the beautiful city of Victoria. My stay here has resulted in, among other things, the dissertation that lies before you. I could not have realized this work without the intellectual, practical, and emotional support of many others.

First and foremost I would like to thank my two Ph.D. supervisors Helena Kadlec and Ulrike Stege. I am grateful to Helena for her continued support of my unorthodox ideas about and methods for cognitive psychological research. Being my most critical audience, Helena has contributed to this research in invaluable ways. I have enjoyed our discussions through the years and I believe that this dissertation is the better for it. Ulrike has taught me most (if not all) I now know about theoretical computer science. I owe to her training an appreciation for rigor in mathematical analysis and a new understanding of the role of mathematics in science. Ulrike's enthusiasm for interdisciplinary research continues to be a source of inspiration for me.

I am indebted to Mike Fellows for introducing me to the fascinating field of computational complexity and for being the one who made it possible for me to come to Victoria in the first place. Thank you, Mike, for your vision and confidence in my potential as a student and as a researcher.

I thank Mike Masson and Hausi Müller for their service as committee members and their constructive comments on my work.

My intellectual development over the years has been shaped by and profited from interactions with many other researchers. In particular, I thank: Pim Haselager and Raoul Bongers (my M.A. supervisors at the University of Nijmegen), Jelle van Dijk, Piet Smit, Gert-Jan Bleeker and Andre de Groot (members of Pim's EEC group), Michelle Arnold and Cindy Bukach (my fellow students and dear friends), Allan Scott, Parissa Agah, and Fei Zhang (members of Ulrike's research group), Steve Lindsay, Mike Masson, Dan Bub, David Mandel, Mandeep Dhami and the other members of the Cognitive group at the University of Victoria.

I also thank Steve Lindsay for coordinating the weekly Cognitive Seminar at the University of Victoria: It has (further) opened my eyes to the beauty and variety of cognitive research. On several occasions I have had the opportunity to present my own research in this seminar and I have found the consistent enthusiasm and interest with which my work has been received truly rewarding.

I thank my brother and ultimate role model, Tibor, for bringing me to Victoria. I thank him and his wife Andrea for making a foreign country feel like home. This dissertation is dedicated to their children: my nephew Ronan and my niece Somerset.

I am grateful to my parents, Peter and Emöke, for their unwavering confidence in me. I am sure they are proud and, above all, happy that I am returning to the Netherlands.

So too will be my all-time comrade Judith. I owe to her, and our friend Nique, a sense of history that I lacked for too long. The distance between us during the last four years has disrupted the regularity of our philosophical discussions, but importantly, not our friendship.

Last, but certainly not least, I thank my wife Denise for her endless love and support. There are no words to express my gratitude for the last 7 years with her and the many more years to come.

To Ronan and Somerset, whom I miss very much

Preface

Cognitive systems are often thought of as information-processing or computational systems, i.e., cognitive processes serve to transform inputs into outputs. On this view, the psychologist's job is to single out the (mathematical) function that captures the input-output behavior of the system under study. The question driving the present study is: What kinds of functions can (not) be computed by cognitive systems? This question is of great importance for all computational approaches to cognition, because its answer will ultimately determine which psychological theories are realistic from a computational perspective and, more importantly, which are not.

In order to answer the question raised, a precise definition of computation is required. In 1936, Alan Turing provided a now widely accepted formalization of the notion of computation. Turing believed that, with his formalization, he had discovered the theoretical boundaries on the ability of humans to compute functions. Following Turing, the early cognitive scientists considered computability the only real theoretical constraint on cognitive theories.

Later an appreciation arose in computer science for the difference between 'computable in principle' and 'computable in practice,' leading to the development of computational complexity theory in the 1970s. Complexity theorists have argued that functions that are computable, in Turing's sense, may not be tractably computable; meaning that no realistic physical machine can be expected to compute those functions in a reasonable amount of time.

In recent years, cognitive scientists have started to use computational complexity theory in analyzing psychological theories and many of them (explicitly or implicitly) view computational tractability as a real constraint on human computation. It is this latter development that motivates this work. This work sets out to critically analyze the notion of computational tractability and its role in cognitive theory.

The general purpose of this research is threefold. The first purpose is to motivate a theoretical discussion on computational tractability in cognitive theory and to make this discussion accessible for the interested cognitive psychologist. The second purpose is to expose and clarify the dominant view of tractability in present-day cognitive science[1] based on classical complexity theory, and to propose and defend an alternative view of tractability based on parameterized complexity theory. The third purpose is to present cognitive psychologists with formal tools for analyzing the classical and parameterized complexity of their theories and to illustrate the use of these tools in different cognitive domains.

Note to the Reader

This research is inherently interdisciplinary—being situated at the crossroads of theoretical computer science and cognitive psychology. Nevertheless I will pitch my discussion particularly towards cognitive psychologists and other psychologically interested cognitive scientists. This is partly because I, myself, am a cognitive psychologist, and thus my interest in computational complexity is primarily motivated by psychological considerations; but also because I think that psychologists, as opposed to, say, artificial intelligence researchers or philosophers, have been most ignorant of computational complexity theory and its application to cognitive theory.

In an attempt to give the reader a firm grip on computational complexity theory I will cover many more details of the mathematical theory of computation than is common in the psychological literature. I believe that a proper application of complexity theory in psychology, and a full appreciation of its unique contribution, demands more than a superficial understanding of this theory. For the mathematician and computer scientist reader I note that I will assume an introductory level knowledge of cognitive psychology, as well as some familiarity with computational modeling as it is practiced in cognitive science (see e.g. Eysenck & Keane, 1994; Stillings, Feinstein, Garfield, Rissland, Rosenbaum, Weisler, & Baker-Ward, 1987, for accessible introductions). Some awareness of philosophical issues with respect to the computational theory of mind, though not necessary to understand the material, may help to appreciate the wider

1 With the term *cognitive science* I mean the interdisciplinary study of cognition, including fields such as cognitive psychology, artificial intelligence, and philosophy of mind.

implications of the ideas pursued here (see e.g. Bechtel, 1988; Chalmers, 1994; Putnam, 1975, 1994).

Overview

Chapter 1 situates the present discussion by explaining the role of cognitive function in psychological theory. Chapter 2 presents preliminaries of the formal theory of computability and complexity, and traces the development from the Church-Turing thesis to a first, informal formulation of the Tractable Cognition thesis. Then Chapter 3 discusses two possible formal instantiations of the Tractable Cognition Thesis. The first is called the P-Cognition thesis. This thesis maintains that cognitive functions are (and must be) computable in polynomial time. I will show that the P-Cognition thesis is the dominant version of the Tractable Cognition thesis in cognitive science today. Further, I will argue that the P-Cognition thesis poses too strict constraints on the set of cognitive functions, with the risk of excluding potentially veridical cognitive theories from psychological investigation. As an alternative I propose and defend the FPT-Cognition thesis. Towards this end I introduce a new branch of complexity theory to cognitive psychology, called parameterized complexity theory.

Chapter 4 presents a primer on strategies and techniques for parameterized complexity analysis. The following three chapters, Chapters 5, 6, and 7, each discuss existing cognitive theories, and subject them to critical complexity analyses using the techniques explained in Chapter 4. These chapters are not intended to have a final say on "the" complexity of the respective theories, but instead are meant to motivate healthy and critical discussion among cognitive scientists on how best to pursue complexity analysis of this type of theories. Furthermore, these chapters illustrate how rigorous complexity analysis of cognitive theories is possible and informative.

Chapter 8 sets out to synthesize and evaluate the ideas and arguments expressed in the preceding chapters. Toward this end, I identify a set of potential objections and present a response to each. The final chapter, Chapter 9, summarizes the main contributions of this research—both at the metatheoretical level and at the level of specific cognitive theories—and proposes future work on the topic.

Chapter 1. Psychological Theories as Mathematical Functions

Cognitive psychologists are interested in understanding how humans perform cognitive tasks (e.g. reading words, recognizing faces, inferring conclusions from a set of premises, predicting future events, remembering past events). This chapter describes how cognitive psychologists tend to conceive of such tasks. From this we conclude that, generally, a cognitive task can be modeled by a function, and that a cognitive process can be understood as the computation of that function. I will discuss the motivation and generality of this conceptualization, and conclude with the question that ultimately motivates this work: 'Which functions can be computed by cognitive processes?'

1.1. What is a Cognitive Task?

Cognitive tasks come in many kinds and flavors. A cognitive task may be viewed as prescriptive (e.g. when an experimenter instructs a participant to perform a certain task, or when a normative theory defines a certain goal as 'rational') or descriptive (e.g. when we view human memory as performing the task of storing information about the world, or when we view human perception as performing the task of constructing internal representations of the external world). Cognitive tasks may be high-level and/or knowledge rich (e.g. reasoning, language production and understanding, decision-making) or low-level and/or knowledge poor (e.g. sensation, locomotion). Cognitive tasks may be subtasks (e.g. letter recognition) or supertasks (e.g. sentence reading) of other cognitive tasks (e.g. word reading). Finally, a cognitive task may be a task performed by a whole organism (e.g. a human), a part of an organism (e.g. a brain, a neuronal group), a group of organisms (e.g. a social group, a society), an organism-tool complex (e.g. a human-computer combination), or even an altogether artificial device (e.g. a computer).

Typically cognitive psychologists conceive of cognitive tasks as information-processing or computational tasks.[2] Generally, such tasks can be characterized as follows:

[2] The terms information-processing and computational system have the connotation that cognitive systems are assumed to be representational (e.g. Massaro & Cowan, 1993). Even though many (computational and non-computational) approaches to cognition

Given a state of the world i (the initial or input state), the goal is to transform state i into state o (the final or output state). Since, psychologists are interested in cognitive tasks *qua* generic tasks, an input state is usually seen as a particular input state (e.g. a particular tone, a particular word, a particular pattern of neuronal firing) belonging to a general class of possible input states (e.g. all perceivable tones, all English words, all possible patterns of neuronal firing). Further, any non-trivial task has at least two output states. If $I = \{i_1, i_2, \ldots.\}$ denotes the set of possible input states and $O = \{o_1, o_2, \ldots.\}$ denotes the set of possible output states for a given task, then we can describe the task by a function $\Pi: I \rightarrow O$ that maps input states I to output states O. In other words, a cognitive task is modeled by a mathematical function.

A system that performs a cognitive task we call a *cognitive system*, the mapping $\Pi: I \rightarrow O$ we call a *cognitive function*, and the mechanism by which the transformation from an input $i \in I$ to output $o = \Pi(i)$ is realized we call a *cognitive process*. We say that a cognitive system/process 'computes' a cognitive function, and we say a cognitive process is the 'computation' of the respective function. (For now the words 'compute' and 'computation' will be informally used. In Chapter 2 these terms will be formally defined).

1.2. Task-Oriented Psychology

The conceptualization of cognitive systems in terms of the tasks they perform is very useful and pervades psychological practice (see e.g. Eysenck & Keane, 1994, or any other textbook of cognitive psychology for an impression). This task-oriented approach makes sense both historically and methodologically.

First of all, theories in experimental psychology tend to be naturally task-oriented, because participants are typically studied in the context of specific experimental tasks. Furthermore, since the birth of cognitive psychology the information-processing

indeed assume some form of representationalism (whether symbolic, distributed, or otherwise; e.g. Haugeland, 1991; but see also Haselager, de Groot, van Rappard, 2003; Wells, 1996, 1998), no such assumption is necessary for an application of computability and complexity theory to psychological theories, and therefore, no such assumption is made here. Whether input and output states (or any intervening states) bear any representational content is irrelevant for analyzing the functional form of a task.

approach to explaining human task performance has been dominant. This approach views human cognition as a form of information processing—i.e., as the transformation of stimuli into mental representations, of mental representations into other mental representations, and of mental representations into responses (e.g. Massaro & Cowan, 1993). To explain how a participant performs a cognitive task, the cognitive psychologist hypothesizes the existence of several cognitive systems and sub-systems, each of them responsible for performing a particular information-processing (sub-)task. Then the task, as a whole, is seen as the conglomeration of the hypothesized set of sub-tasks.

Consider, for example, the task of detecting a tone among noise. According to Signal Detection Theory (SDT; Green & Swets, 1966) this task is performed by two cognitive sub-systems: a perceptual system and a decisional system. The perceptual system transduces the stimulus (e.g. either a tone among noise or noise alone) into an internally represented perceptual impression. This perceptual impression is usually modeled by a point in a one-dimensional space (but see e.g. Kadlec & Townsend, 1992, for extensions of SDT to multiple dimensions). The decisional system serves to make a decision about whether the perceptual impression provides sufficient evidence that the tone was present in the stimulus. It does so by defining a criterion (modeled by a critical point in perceptual space), and output "yes, tone is present" if the perceptual impression exceeds the criterion, and output "no, tone is absent" otherwise.

The idea that cognitive systems and sub-systems serve particular purposes fits with both evolutionary and developmental perspectives of cognition (Anderson, 1990; Glenberg, 1997; Inhelder & Piaget, 1958; Margolis, 1987). This conceptualization of cognition has also proven useful in the area of cognitive neuropsychology, where double dissociation[3] methodology is being used to "carve-up" the human brain into modules, each module presumably responsible for a particular mental function (Kolb & Whishaw, 1996). Finally, cognitive psychology's focus on the functioning of cognitive sub-systems follows from its philosophical commitment to *functionalism*. Functionalism postulates that cognitive processes are defined by their functionality (i.e., how cognitive states

[3] For recent discussions on the notion of 'mental modules' and 'double dissociation methodology' see e.g. Dunn (2003), Dunn and Kirsner (2003), Kadlec and van Rooij (2003), van Orden and Kloos (2003).

functionally relate to other cognitive states), as opposed to, say, the “stuff” they are made of (e.g. Block, 1980; Putnam, 1975; but see also Putnam, 1988; Searle, 1980). It is also functionalism that fuels the cognitive psychologist’s belief that s/he can understand human cognition in terms of its functional properties, more or less independently from the physical properties of human cognitive systems. The next section further discusses this view of psychological explanation.

1.3. Levels of Psychological Explanation

On the very general view presented here, a psychological theory of a cognitive task should minimally specify the function (or set of functions) that the system is believed to compute when performing the task. Such a functional level description answers the “what”-question; i.e., what is the task as construed by the system under study? Once such a description is successfully formulated, the psychologist may be interested in addressing the “how”-question; i.e., how is the computation of the function (physically) realized? This distinction between “what is being computed” (the cognitive function) and “how it is computed” (the cognitive process) is also reflected in a well-known framework proposed by David Marr (1982).

Marr (1982) proposed that, ideally, a cognitive theory should describe a cognitive system on three different levels. The first level, called the *computational level*, specifies what needs to be computed in order to perform a certain task. The second level, the *algorithmic level*, specifies how the computation described at the first level is performed (i.e., a description of the exact representations and algorithms used to compute the goal). The third and final level, the *implementation level*, specifies how the representations and algorithms defined at the second level are physically implemented by the “hardware” system performing the computation.

Hence, in Marr’s terminology, the description of a cognitive system in terms of the function it computes is a computational level theory.[4] Since one and the same

[4] Note that many theories in cognitive psychology referred to as ‘computational’ are in fact algorithmic level theories (cf. Humphreys, Wiles, & Dennis, 1994). Because a computational level description of a cognitive system does not specify the algorithmic procedures employed to compute the system’s function, some people have found the name ‘computational level’ misleading or confusing. Alternative names proposed for the

function can be computed by many different algorithms (e.g. serial or parallel), we can describe a cognitive system at the computational level more or less independently of the algorithmic level. Similarly, since an algorithm can be implemented in many different physical systems (e.g. carbon or silicon), we can describe the algorithmic level more or less independently from physical considerations.[5] Marr argued for the priority of computational level descriptions in psychological theories.[6] He believed this was the best way to make progress in cognitive theory, because "an algorithm is likely to be understood more readily by understanding the nature of the problem being solved than by examining the mechanism (and the hardware) in which it is embodied" (Marr, 1982, p. 27; see also Marr, 1977). Marr's view is succinctly summarized by Frixione (2001, p. 381) as follows:

> "The aim of a computational theory is to single out a function that models the cognitive phenomenon to be studied. Within the framework of a computational approach, such a function must be effectively computable. However, at the level of the computational theory, no assumption is made about the nature of the algorithms and their implementation" (Frixione, 2001, p. 381).[7]

Marr's ideas have been very influential in cognitive psychology and artificial intelligence alike. Although his approach has not been without criticism (e.g. McClamrock, 1991), his general framework has found wide application in cognitive science and psychology (albeit often in adjusted form), both among symbolists

---

computational or comparable level include: the semantic level (Pylyshyn, 1984), the level of cognitive state transitions (Horgan & Thienson, 1996), the knowledge level (Newell, 1982), and the rational level (Anderson, 1990). Since all these names have connotations that I do not intend, I will avoid usage of these terms.

[5] Few present-day psychological theories include descriptions at the implementation level. Exceptions are probably best found in the literature on low-level cognitive processes (e.g. sensation and perception). Cognitive psychology neglects the implementational level because, in line with functionalism, it regards implementation details to a large extent irrelevant to cognitive theory and psychologically uninteresting.

[6] cf. Anderson's (1990) arguments for the priority of his *rational* level.

[7] Probably Marr had a bit more in mind when he proposed his computational level. For example, on his view a computational level theory also needed to include a rationale for *why* the proposed function is the right function to solve the task at hand. This added information, however, does not contradict the interpretation of the computational level by Frixione and myself.

(Pylyshyn, 1984; Newell, 1982) and connectionists (Rumelhart, McClelland, & the PDP Research Group, 1986), and interestingly, even among dynamicists (Horgan & Tienson, 1996).[8]

1.4. Motivation and Research Question

The present study pertains to cognitive theories that are formulated at the computational level; meaning, they minimally specify a function hypothesized to be computed by the system under study. The question on which the discussion will center is: 'Which functions can be computed by cognitive processes?' The answer to this question is invaluable for any computational approach to cognition, as it directly answers the question 'Which functions can serve as computational level theories?' In this investigation we make two assumptions:

> Assumption 1. (*computationalism*) The cognitive process is believed to be a mechanical process in the sense that it can be described by an algorithm.
>
> Assumption 2. (*materialism*) The cognitive process is believed to be a physical process occurring in space and time; i.e., no cognitive step is instantaneous and the physical space in which the process unfolds is limited.

Like functionalism, these assumptions are part of the philosophical foundations of all computational approaches to cognition.[9]

---

[8] Even theories that are often seen as being formulated at the algorithmic level—such as connectionist or neural network models (e.g. Rumelhart, McClelland, & the PDP Research Group, 1986)—are not free from computational level considerations. Also for neural networks it is of interest to study which functions they can and cannot compute (Parberry, 1994). For example, neural network learning is a computational task: A neural network is assumed to learn a mapping from inputs to outputs by adjusting its connection weights. Here the input of the learning task is given by (*I*) all network inputs in the training set *plus* the required network output for each such network input, and the output is given by (*O*) a setting of connection weights such that the input-output mapping produced by the trained network is satisfactory. This learning task, like any other task in the more symbolic tradition, can be analyzed at the computational level (Judd, 1990; Parberry, 1994).

[9] This work is written completely from the perspective of computationalism (see e.g. Chalmers, 1994). For information on non-computationalist (or anti-computationalist) approaches to cognition see e.g. Haselager, Bongers, and van Rooij (forthcoming), Port and van Gelder (1995), Thelen and Smith (1994), van Gelder (1995, 1998, 1999), van Rooij, Bongers and Haselager (2000, 2002).

Chapter 2. Problems, Algorithms, Computability and Tractability

While the previous chapter situated the present work in cognitive psychology, this chapter situates it in theoretical computer science. Here I present some preliminaries of the theory of computation. I start by discussing three different classes of computational problems. Then I introduce the Turing machine formalism of computation, and discuss its application in cognitive theory in the form of the Church-Turing thesis. Finally, I introduce preliminaries of computational complexity theory and close with an open-ended formulation of the Tractable Cognition thesis.

2.1. Classes of Problems

In the theory of computation, functions are often also referred to as *problems*. The only difference between the use of the words 'problem' and 'function' is a matter of perspective; the word 'problem' has a more prescriptive connotation of an input-output mapping that *is to be* realized (i.e., a problem is to be solved), while the word 'function' has a more descriptive connotation of an input-output that *is being* realized (i.e., a function is computed). Since the words 'function' and 'problem' refer to the same type of mathematical object (an input-output mapping) we can use the terms interchangeably (e.g. a 'cognitive function' can be called a 'cognitive problem,' depending on the perspective taken). Section 2.1.1 formally introduces three classes of problems. To support psychological intuition, Section 2.1.2 briefly illustrates examples of these classes in cognitive theory.

2.1.1. Search, Decision and Optimization problems

We will denote a problem (function) by Π. Each problem Π is specified as follows.

First we specify the name of the problem, Π. Then we specify:

Input: Here we describe an instance i in terms of the general class I to which it belongs.

Output: Here we describe the output $\Pi(i)$ required for any given instance i.

This type of description is called the *problem definition* of Π.

We consider the problem Vertex Cover as an example. Let $G = (V, E)$ be a graph.[10] Then a set of vertices $V' \subseteq V$ is called a *vertex cover* for G if every edge in E is incident to a vertex in V' (i.e., for each edge $(u, v) \in V$, $u \in V'$ or $v \in V'$). The problem definition of Vertex Cover is as follows: [11]

Vertex Cover (*search version*)

Input: A graph $G = (V, E)$ and a positive integer k.

Output: A vertex cover V' for G with $|V'| \leq k$ (i.e., V' contains at most k vertices), if such a vertex set exists, else output "no".

Traditionally, a distinction is made between different classes of problems. We consider three classes: search problems, decision problems and optimization problems. A *search problem* asks for an object that satisfies some criterion, called a *solution* to the problem. The problem Vertex Cover as defined above is a search problem: It asks for a vertex cover of size at most k (if such a vertex cover does not exist the problem outputs "no" to indicate this fact). Alternatively, a decision problem just asks whether or not a solution exists. Hence, the output for a decision problem is either "yes" or "no." Conventionally the output specification for a decision problem is formulated as a question. For example, the decision version of Vertex Cover is defined as follows:

Vertex Cover (*decision version*)

Input: A graph $G = (V, E)$ and a positive integer k.

---

[10] All graph theoretic terminology and notation used throughout the text is defined in Appendix A. The appendix also provides a brief introduction to graph theory for readers unfamiliar with this branch of mathematics.

[11] The problem Vertex Cover will be used as a running example. The problem is widely studied in computer science and operations research (e.g. Garey & Johnson, 1979), including computational biology (Stege, 2000) and cognitive science (Jagota, 1997; Stege, van Rooij, Hertel, & Hertel, 2002; van Rooij, Stege, & Kadlec, 2003). It finds application, among other things, as a model of scheduling tasks. For example, one can view the graph as a representation of a conflict situation, with the vertices representing activities that one wishes to undertake and each edge (u, v) indicating that activity u and v cannot be performed simultaneously. The problem Vertex Cover then asks for a set V' of at most k activities such that, after removal of the activities in V' from the set V, all remaining activities in $V \backslash V'$ can be simultaneously performed (Here $V \backslash V' = \{v \in V \mid v \notin V'\}$ denotes the set difference between V and V'); or, in other words, Vertex Cover asks for a selection of at least $|V| - k$ activities that can be simultaneously performed (cf. Stege et al., 2002).

Question: Does there exist a vertex cover V' for G with $|V'| \leq k$?

Solving a decision problem (also called *deciding* a decision problem) consists of correctly answering the question by either "yes" or "no." An input (or instance) i for a decision problem Π is called a *yes-instance* for Π if the answer to the question posed by Π is "yes" for i. Otherwise, i is called a *no-instance*. If an algorithm that solves a decision problem also solves its corresponding search problem, we say the algorithm is *constructive*. Unless otherwise stated, all algorithms that we consider are constructive.

Finally, some problems are called *optimization problems*, because they ask for a solution that is optimized (either maximized or minimized) on some dimension. Vertex Cover naturally allows for a reformulation as an optimization problem, in this case a minimization problem:

Vertex Cover (*optimization version*)

Input: A graph $G = (V, E)$ and a positive integer k.

Output: A *minimum* vertex cover V' for G (i.e., a vertex cover V' such that $|V'|$ is minimized over all possible vertex covers for G).

Note that an optimization problem is a special type of search problem, and that it always has a solution. The optimization version of Vertex Cover is often also called Minimum Vertex Cover.

As we have seen, one and the same "problem" (e.g. Vertex Cover)[12] can be formulated as a search problem, as a decision problem, and as an optimization problem. Following convention in computer science, we will typically work with decision problems. This will not overly restrict computational analysis, because results obtained for decision problems often generalize directly to results for their respective search versions and, if applicable, optimization versions.

2.1.2. Illustrations in Cognitive Theory

It is not hard to find examples of the different types of computational problems in cognitive theory. For example, on page 6 in Chapter 1, we already encountered a decision

[12] Note that Vertex Cover takes discrete inputs (graphs and integers) and gives discrete outputs (a vertex set, and e.g. '1' for "yes" and '0' for "no"). In the literature, problems like this are also called *combinatorial problems*. This work is concerned with combinatorial problems only.

problem: viz., the task performed by the decision system in Signal Detection Theory (Green & Swets, 1966). For an example of a search problem we consider Prototype Theory (e.g. Rosh, 1973). This cognitive theory assumes that an object (e.g. an animal), called an exemplar, is classified as belonging to a certain category (e.g. the category dogs) if it is sufficiently similar to the *prototype* of that category. Both the exemplar and prototype are defined by sets of features, and the similarity between an exemplar *e* and a prototype *p* is measured by some function $s(e, p)$. The following search problem captures this computational level description:

Prototype Categorization

Input: A set of categories C. For each category $c_i \in C$ an associated prototype p_i. An exemplar e and a threshold λ.

Output: A category $c_i \in C$ such that $\mathrm{s}(e, p_i) \geq \lambda$, if such a category exist, else output "no."

Here the output "no," can be read as meaning that the exemplar is seen as unclassifiable.

Lastly, as an example of an optimization problem we consider Utility Theory (Luce & Raiffa, 1957; 1990). This (normative) cognitive theory assumes that, when presented with a set of alternatives with uncertain outcomes, people (should) choose the alternative with maximum expected utility. The following optimization problem captures this computational level description.

Maximum Expected Utility

Input: A set of alternatives A and a set of outcomes O. For each $a \in A$ and each $o \in O$, $\mathrm{P}(o|a)$ denotes the probability that o occurs if a is chosen. Further, each outcome o has an associated utility, $u(o)$.[13]

Output: An alternative $a \in A$ with maximum expected utility (i.e., an alternative a such that $\sum_{o \in O} u(o)P(o \mid a)$ is maximized over all possible a).

[13] Because we limit inputs to discrete objects it is assumed that the values $P(o|a)$ and $u(a)$ are rational numbers.

2.2. Formalizing Computation

Up to now we have been using the terms ‘computation’ and ‘algorithm’ informally. To have a solid theory of computability we need a formal definition of these terms. This section introduces the Turing machine formalization of computation. For more information on computability and Turing machines the reader is referred to Herken (1988), Hopcroft, Motwani, and Ullman (2001) and Lewis and Papadimitriou (1998). Also, Li and Vitányi (1997) give a brief but accessible introduction to the Turing machine formalization. Further, psychologists and cognitive scientists may find treatments by Frixione (2001), Putnam (1975; 1994), and Wells (1998) particularly illustrative. Readers familiar with computability theory may skip this section without loss of continuity.

2.2.1. The Intuitive Notion of a Computation

Informally, when we say a system computes a function or solves a problem, $\Pi: I \to O$, we mean to say that the system reliably transforms every $i \in I$ into $\Pi(i) \in O$ in a way that can be described by an algorithm.[14] An algorithm is a step-by-step finite procedure that can be performed, by a human or machine, without the need for any insight, just by following the steps as specified by the algorithm. The notion of an algorithm, so described, is an intuitive notion. Mathematicians and computer scientists have pursued several formalizations (e.g. Church, 1936; Kleene, 1936; Post, 1936). Probably the best-known formalization, in particular among cognitive scientists and psychologists, is the one by Alan Turing (1936). One of the strengths of Turing’s formalization is its intuitive appeal and its simplicity.

Turing motivated his formalization by considering a paradigmatic example of computation: The situation in which a human sets out to compute a number using pen and paper (see Turing, 1936, pp. 249-252). Turing argued that a human computer can be in at most a finite number of different “states of mind,” because if “we admitted an infinity of states of mind, some of them will be ‘arbitrarily close’ and will be confused” (p. 250). Similarly, Turing argued a human computer can read and write only a finite number of

[14] In the literature an algorithm is also sometimes referred to as mechanical procedure, effective procedure, or computation procedure.

different symbols, because if "we were to allow an infinity of symbols, then there would be symbols different to an arbitrarily small extent" (p. 249). On the other hand, Turing allowed for a potentially infinite paper resource. He assumed that the paper is divided into squares (like an arithmetic note book) and that symbols are written in these squares. With respect to the reading of symbols Turing wrote: "We may suppose that there is a bound B on the number of symbols or squares that the computer can observe at one moment. If [s/he] wishes to observe more [s/he] must use successive operations" (p. 250). This restriction was motivated by the observation that for long lists of symbols we cannot tell them apart in "one look." Compare, for example, the numbers 96785959943 and 96785959943. Are they the same or different?

According to Turing, the behavior of a human computer at any moment in time is completely determined by his/her state of mind and the symbol(s) s/he is observing. The computer's behavior can be understood as a sequence of operations, with each operation "so elementary that it is not easy to imagine [it] further divided" (p. 250). Turing distinguished the following two elementary operations:

(a) A possible change of a symbol on an observed square.

(b) A possible change in observed square.

Each operation is followed by a possible change in state of mind. With this characterization of computation Turing could define a machine to do the work of a human computer. Figure 2.1 illustrates this machine.

2.2.2. The Turing Machine Formalism

A Turing machine M is a machine that at any moment in time is in one of a finite number of *machine states* (analogue to "states of mind"). The set of possible machine states is denoted by $Q = \{q_0, q_1, q_2, \ldots, q_n\}$. One machine state q_0 is designated the *initial state*; this is the state that M is in at the beginning of the computation. There is also a non-empty set $H \subseteq Q$ of *halting states*; whenever the machine goes into a state $q_i \in H$ then the machine *halts* and the computation is terminated.

The machine has a *read/write head* that gives it access to an external memory, represented by a one-dimensional *tape* (analogue to the paper). The tape is divided in discrete regions called *tape squares*. Each tape square may contain one or more *symbols*.

The machine can move the read/write head from one square to a different square, always moving the read/write head to the right or left at most one square at a time. The read/write head is always positioned on one (and at most one) square, which it is said to *scan*. If a square is scanned then the machine can read a symbol from and/or write a symbol to that square. At most one symbol can be read and/or written at a time.

The set of possible symbols is denoted by *S*, and is called the *alphabet* of *M*. *S* is a finite set. Often it is assumed that $S = \{0, 1, B\}$, where B is called the *blank*. Time is discrete for *M* and time instants are ordered 0, 1, 2, …. At time 0 the machine is in its initial state q_o, the read/write head is in a starting square, and all squares contain Bs except for a finite sequence of adjacent squares, each containing either 1 or 0. The sequence of 1s and 0s on the tape at time 0 is a called the *input*.

The Turing machine can perform two types of basic operations:

(a') it can write an element from *S* in the square it scans; and

(b') it can shift the head one square left (L) or right (R).

After performing an operation of either type (a') or (b') the machine takes on a state in *Q*. At any one time, which operation is performed and which state is entered is completely determined by the present state of the machine and the symbol presently scanned. In other words, the behavior of a Turing Machine can be understood as being governed by a function *T* that maps a subset of *Q* x *S* into *Q* x *A*, where $A = \{0, 1, B, L, R\}$ denotes the set of possible operations.

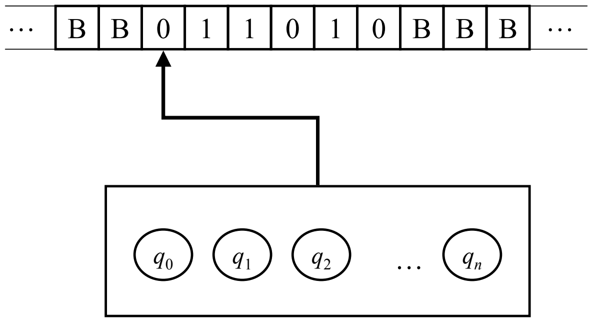


Figure 2.1. Illustration of a Turing machine.
The tape extends left and right into infinity. Each square on the tape contains a symbol in the set {1, 0, B}. The machine can read and write symbols with a read/write head, and can be in a finite number of different machine states $\{q_0, q_1, q_2, \ldots, q_n\}$.

We call T the *transition function* of M. A transition $T(p, s) = (a, q)$ is interpreted as follows: If $p \in Q$ is the current state and $s \in S$ is the current scanned symbol, then the machine performs operation $a \in A$ of type (a') or (b'), and the machine enters the state $q \in Q$. For example, $T(p, 0) = (1, q)$ means that if M is in state p and read symbol 0 then M is to write symbol 1 and go into state q; $T(p, 1) = (\mathrm{L}, q)$ means that if M is in state p and read symbol 1 then M is to move its read/write head one square to the left and go into state q. Note that Q, S, and A are finite sets. Thus we can also represent the transition function T as a finite list of transitions. Such a list is often called the *machine table* and transitions are then called *machine instructions*.

Under the governance of T the machine M performs a uniquely determined sequence of operations, which may or may not terminate in a finite number of steps. If the machine does halt then the sequence of symbols on the tape is called its *output*. A Turing machine is said to compute a function Π if for every possible input i it outputs $\Pi(i)$. A function is called *computable* (or *Turing-computable*) if there exists a Turing machine that computes it. Turing (1936) proved that there exist (infinitely many) problems that are not computable. For example, he showed that the Halting problem is not computable. This decision problem is formulated as follows:

Halting problem
Input: A Turing machine M and an input i for M.
Question: Does M halt on i?

2.2.3. Extensions of the Turing Machine Concept

The reader may wonder to what extent the particular limitations placed by Turing on his machine are crucial for the limitations on computability. Therefore, a few notes should be made on the computational power of the Turing machine with certain extensions. It has been shown that several seemingly powerful adjustments to Turing's machine do *not* increase its computational power (see e.g. Lewis & Papadimitriou, 1998, for an overview). For example, the set of functions computable by the Turing machine described above is the same as the set of functions computable by Turing machines with one or more of the following extensions:

(1) Turing machines with multiple tapes and multiple read/write heads

(2) Turing machines with any finite alphabets (i.e., not necessarily $A = \{0, 1, B\}$)
(3) Turing machines with random access: These are Turing machines that can access any square on the tape in a single step.
(4) Non-deterministic Turing machines: These are Turing machines that, instead of being governed by a transition function, are governed by a transition *relation*, mapping some elements in Q x S to possibly more than one element in A x Q. Such a non-deterministic machine is said to "compute" a function Π if for every input i there exist one *possible* sequence of operations that, when performed, would lead to output $\Pi(i)$.

It should be noted that machines of type (4) are not considered to really compute in the sense that Turing meant to capture with his formalism. Namely, in non-deterministic machines not every step of the computation is uniquely determined and thus, a human wishing to follow the set of instructions defined by the machine table would not be able to unambiguously determine how to proceed at each step. It will become clear in Chapter 3 that even though non-deterministic machines are purely theoretical constructs they do serve a special purpose in theories of computational intractability.

The extensions (1) – (3), on the other hand, are considered reasonable extensions (see also Section 2.4.2). Hereafter, the term Turing machine will be used to refer to Turing machines with and without such extensions.

2.3. The Church-Turing Thesis

Turing (1936) presented his machine formalization as a way of making the intuitive notions of "computation" and "algorithm" precise. He proposed that every function[15] for which there is an algorithm–which is intuitively computable–is computable by a Turing machine. In other words, functions that are not computable by a Turing machine are not computable in principle by any machine. In support of his thesis, Turing showed that Turing-computability is equivalent to a different formalization independently proposed

[15] To be precise, Turing (1936) made the claim specifically for *number theoretic* functions; but the notion of Turing computability is naturally extended to functions involving other discrete mathematical objects.

by Church (1936). The thesis by both Turing and Church that their respective formalizations capture the intuitive notion of algorithm is now known as the Church-Turing thesis. Further, Turing's and Church's formalizations have also been shown equivalent to all other accepted formalizations of computation, by which the thesis gained further support (see e.g. Israel, 2002; Gandy, 1988; Kleene, 1988, for discussions).

The Church-Turing thesis has a direct implication for the type of cognitive theories described in Chapter 1. Namely, consider the situation in Figure 2.2. This figure illustrates that, assuming that the Church-Turing thesis is true, the set of functions computable by cognitive systems (the cognitive functions) is a subset of the set of functions computable by a Turing machine (the computable functions). On this view, a computational level theory that assumes that the cognitive system under study computes an uncomputable function can be rejected on theoretical grounds.

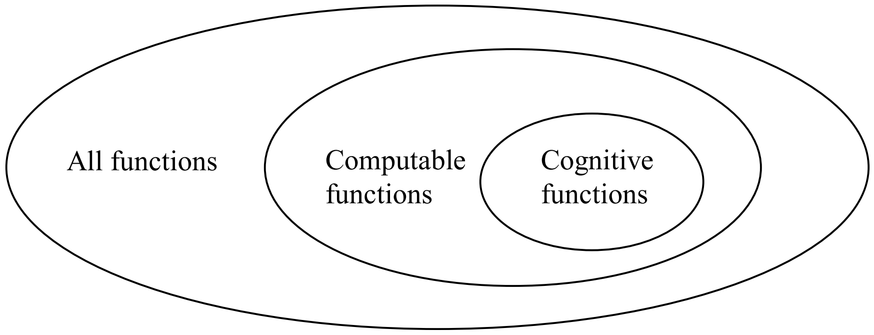


Figure 2.2. Illustration of the Church-Turing Thesis.
On the Church-Turing thesis cognitive functions are a subset of the computable functions.

Note that the Church-Turing thesis is not a mathematical conjecture that can be proven right or wrong. Instead the Church-Turing thesis is a hypothesis about the state of the world. Even though we cannot prove the thesis, it would be in principle possible to falsify it; this would happen, for example, if one day a formalization of computation were developed that (a) is not equivalent to Turing computability, and that, at the same time, (b) would be accepted by (most of) the scientific community. For now the situation is as follows: Most mathematicians and computer scientists accept the Church-Turing thesis, either as plainly true or as a reasonable working-hypothesis. The same is true for many cognitive scientists and cognitive psychologists—at least, for those who are familiar with the thesis.

2.3.1. Criticisms

Despite its wide acceptance, the Church-Turing thesis, and its application to human cognition, is not without its critics. The critics can be roughly divided into two camps: Those who believe that cognitive systems can do "more" than Turing machines and those who think they can do "less."

Researchers in the first camp pursue arguments for the logical possibility of machines with so-called super-Turing computing powers (e.g. Copeland, 2002; Steinhart, 2002).[16] Much of this work is rather philosophical in nature, and is concerned more with the notion of what is computable in principle by hypothetical machines and less so with the notion of what is computable by real, physical machines (though some may agree to disagree on this point; see e.g. Cleland, 1993, 1995; but see also Horsten & Roelants, 1995). The interested reader is referred to the relevant literature for more information about the arguments in this camp (see also footnote 16 for references). Here, we will be concerned only with the critics in the second camp.

Researchers in the second camp do not doubt the truth of the Church-Turing thesis (i.e., they believe that the situation depicted in Figure 2.2 is veridical), but they question its practical use for cognitive theory. Specifically, these researchers argue that computability is a not a strict enough constraint on cognitive theories (e.g. Frixione, 2001; Oaksford & Chater, 1993, 1998; Parberry, 1994). Namely, cognitive systems, being physical systems, perform their tasks under time- and space-constraints and thus

[16] Sometimes claims about super-Turing computing powers are made in the cognitive science literature without any reasonable argument or even a reference (e.g. van Gelder, 1999), and very often the distinction between computability (as defined in Section 2.2) and computational complexity (as discussed in Section 2.4 and Chapter 3) is muddled or ignored (e.g. van Gelder, 1998). It is true that results about super-Turing computing can be found in the theoretical computer science literature (e.g. Siegelmann & Sontag, 1994). However, these results seem to depend crucially on the assumption of infinite precision (or infinite speed-up; e.g. Copeland, 2002), and thus the practicality of these results can be questioned. Furthermore, even if infinite precision is possible in some physical systems, it may still not be possible in human cognitive systems (cf. Chalmers, 1994; Eliasmith, 2000, 2001).

functions computed by cognitive systems need to be computable in a realistic amount of time and with the use of a realistic amount of memory (cf. Simon, 1957, 1988, 1990). The study of computational resource demands is called computational complexity theory. It is to this theory that we turn now.

2.4. Computational Complexity

This section introduces the basic concepts and terminology of computational complexity theory. For more details on computational complexity theory refer to Garey and Johnson (1979), Karp (1972), and Papadimitriou and Steiglitz (1988). Cognitive psychologists may find treatments by Frixione (2001), Parberry (1997), and Tsotsos (1990) particularly illustrative. The reader familiar with computational complexity may skip this section without loss of continuity.

2.4.1. Time-complexity and *Big-Oh*

Computational complexity theory, or complexity theory for short, studies the amount of computational resources required during computation. In computational complexity theory, the complexity of a problem is defined in terms of the demands on computational resources *as function of the size of the input*. The expression of complexity in terms of a function of the input size is very useful and natural. Namely, it is not the fact *that* demand on computational resources increases with input size (this will be true for practically all problems), but *how* it increases, that tells us something about the complexity of a problem. The most common resources studied by complexity theorists are *time* (how many steps does it take to solve a problem) and *space* (how much memory does it take to solve a problem). Here we will be concerned with time complexity only.

We first introduce the *Big-Oh notation*, $O(.)$, that we use to express input size and time complexity. The $O(.)$ notation is used to express an asymptotic upperbound. A function $f(x)$ is $O(g(x))$ if there are constants $c \geq 0$, $x_o \geq 1$ such that $f(x) \leq cg(x)$, for all $x \geq x_o$.[17] In other words, the $O(.)$ notation serves to ignore constants and lower order polynomials in the description of a functions. For this reason $O(g(x))$ is also called the

[17] The definition of $O(.)$ can be straightforwardly extended to functions with two or more variables. For example, a function $f(x, y)$ is $O(g(x, y))$ if there are constants $c \geq 0$ and x_o, $y_o \geq 1$ such that $f(x, y) \leq cg(x, y)$, for all $x \geq x_o$ and $y \geq y_o$.

order of magnitude of $f(x)$. For example, $1 + 2 + \ldots + x = x(x + 1)/2$ is on the order of x^2, or $O(x^2)$, and $x^4 + x^3 + x^2 + x + 1$ is $O(x^4)$.

The notions 'input size' and 'time complexity' are formalized as follows. Let M be a Turing machine and let i be an input. Then the input size, denoted by $|i|$, is the number of symbols on the tape of M used to represent i. Consider, for example, an input i that is a graph $G = (V, E)$. We can encode the graph in several ways. For example, we can encode it as an adjacency matrix, with each row r (and column c) of the matrix representing a vertex in V, and each cell (r, c) in the matrix coded '1' if $(r, c) \in E$, and '0' otherwise. Alternatively, we can encode G as a list of vertices and edges, as follows: $v_1, v_2, \ldots, v_n, (v_1, v_2), (v_2, v_3), (v_1, v_3), \ldots, (v_{n-1}, v_n)$. Let $|V| = n$ and $|E| = m$. Since an $n \times n$ matrix has n^2 cells, the size of the matrix encoding is $O(n^2)$. The size of the list encoding is $n + m$. Since $m < n^2$ and $n + n^2$ is $O(n^2)$, also $n + m$ is $O(n^2)$.[18]

We start by considering the time complexity of algorithms, i.e., of Turing machines. Later we define the time complexity of problems in terms of algorithms that solve them. Let M be a Turing machine. If for any input i, M halts in at most $O(t(|i|))$ steps, then we say M *runs* in time $O(t(|i|))$ and that M has *time complexity* $O(t(|i|))$. Note that, since we require M to halt in time $O(t(|i|))$ for *all* possible inputs, $O(t(|i|))$ should be interpreted as an asymptotic bound on M's *worst-case* running time.

For purposes that will become clear later, we distinguish between algorithms with polynomial-time complexity and algorithms with exponential-time complexity. A polynomial-time algorithm runs in time $O(|i|^\alpha)$ for some constant α. On the other hand, the running time of an exponential-time algorithm is unbounded by any polynomial and is $O(\alpha^{|i|})$, for some constant α.

The exact form of $O(t(|i|))$ will depend, of course, on the particular encoding used to represent the input i on M's tape, the size of M's alphabet, the number of tapes and read/write heads that M has, etc. Although, the $O(.)$ notation allows us to abstract away from some of these machine details some arbitrary choices about the machine model need to be made. For convenience we assume that we are dealing with Turing machines, with any reasonable extensions as discussed in Section 2.2.3, that can perform basic arithmetic

[18] Sometimes we express the size of a graph by n, instead of $n + m$, since $m < n^2$.

operations (e.g. add two numbers, compare two numbers) in a single time step. In Section 2.4.4 I will discuss the consequences of this particular choice of machine model.

2.4.2. Illustrating Algorithmic Complexity

To illustrate running-time analysis of algorithms we consider two algorithms; one a polynomial-time algorithm and the other an exponential-time algorithm. The first algorithm is called Greedy Vertex Cover algorithm. Here it is in pseudo-code:

```
Algorithm Greedy Vertex Cover
    Input: a graph $G = (V, E)$
    Output: A vertex cover $V' \subseteq V$
    1. $V' := \varnothing$
    2. while $E \neq \varnothing$ do
        3. find a vertex $v \in V$ such that $\deg_G(v)$ is maximum
        4. $V' := V' \cup \{v\}$
        5. $V := V \backslash \{v\}$
        6. $E := E \backslash R_G(\{v\})$        \\Here $R_G(\{v\}) = \{(x, y) \in E \mid v \in \{x, y\}\}$
    7. end while
    8. return $V'$
```

This algorithm takes as input a graph $G = (V, E)$ and builds a vertex cover $V' \subseteq V$ as follows. First it finds a vertex v of maximum degree (line 3). Then it includes v in the vertex set V' (line 4) and removes v and all its incident edges from G (lines 5 and 6). The algorithm repeats this loop until no edge is left in the graph. When finished, the algorithm outputs V' (line 8).

We show this algorithm runs in time $O(n^2)$, where $n = |V|$. First to perform line 3, the algorithm has to consider at most n vertices to determine which vertex has maximum degree. We may assume that for each vertex its degree is explicitly encoded. Then we can determine a vertex degree by simply reading it off in constant time $O(1)$. We conclude that line 3 can be performed in time $O(n + n) = O(n)$. Both lines 4 and 5 take constant time $O(1)$. To delete each edge adjacent to v, in line 6, we consider at most $n - 1$ edges and this can be done in time $O(n)$. Note that we also need to update the vertex degrees of vertices in G that lost an edge in line 6. Since there are at most $n - 1$ such vertices, updating can be done in time $O(n)$. In sum, a single run through the loop can be done in time $O(n + 1 + 1 + n + n) = O(3n) = O(n)$. How often do we go through the loop? At most n times; since there are no more than n vertices in the graph. This allows us to conclude a running time of $O(n \times n) = O(n^2)$.

We now consider an exponential-time algorithm, called the Exhaustive Vertex Cover algorithm. I give the algorithm below in pseudo-code and explain it using Figure 2.3.

```
    Algorithm Exhaustive Vertex Cover
    Input: a graph $G = (V, E)$ and a positive integer $k$
    Output: A vertex cover $V' \subseteq V$, with $|V'| \leq k$, if one exists, and $\varnothing$ otherwise.
1.  $V' := \varnothing$
2.  pick any vertex $v \in V$ such that $\deg_G(v) \geq 1$
// If there does not exist a vertex $v \in V$ with $\deg_G(v) \geq 1$, then $v$ = null
3.  if $v \neq$ null then
    4. $V'$ := Branch($G$, $V'$, $k$, $v$)
5.  return $V'$
```

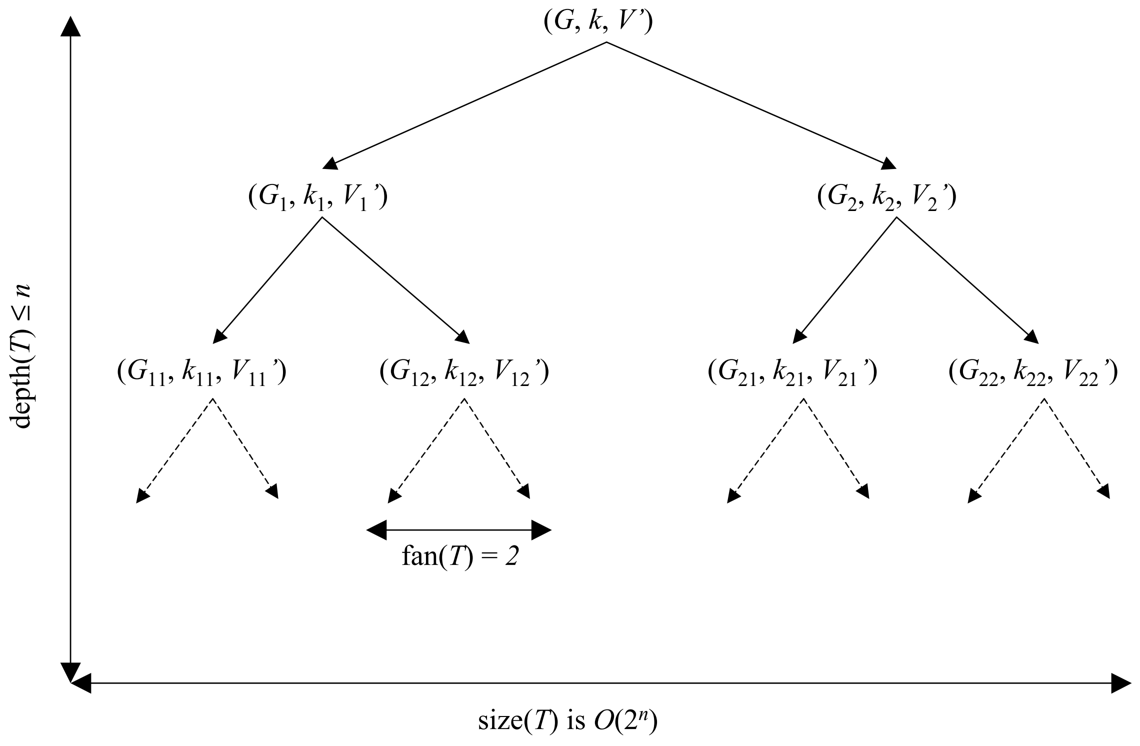
The algorithm Branch is recursively called, and is defined as follows:

```
    Algorithm Branch($G$, $V'$, $k$, $v$):
    Input: A graph $G = (V, E)$, a vertex set $V' \subseteq V$ ($V'$ may be empty), a
           positive integer $k$, a vertex $v \in V$
    Output: A vertex cover $V' \subseteq V$, with $|V'| \geq k$, if one exists, and $\varnothing$ otherwise.
6.  if $k \leq 0$ then
    7. $V' := \varnothing$
    8. return $V'$
// Creating the left branch:
9.  $V_1' := V' \cup \{v\}$
10. $G_1 := (V_1, E_1)$ with $V_1 := V \backslash \{v\}$ and $E_1 := E \backslash R_G(\{v\})$
//Reminder: $R_G(\{v\}) = \{(x, y) \in E \mid v \in \{x, y\}\}$
11. $k_1 := k - 1$
12. pick any vertex $v_1 \in V$ such that $\deg_{G_1}(v_1) \geq 1$
```

```
13. **if** $v_1 \neq$ null **then**
    14. $V_1$':= Branch($G_1$, $V_1$', $k_1$, $v_1$)
15. **else** if $k_1 < 0$ **then**
    16. $V_1' := \varnothing$
17. **if** $V_1' \neq \varnothing$ **then** //answer "yes" is found
    18. **return** $V_1$'
// Creating the right branch
19. $V_2' := V'$
20. $G_2 := (V_2, E_2)$ with $V_2 := V \backslash \{v\}$ and $E_2 := E \backslash \mathrm{R}_G(\{v\})$
21. $k_2 := k$
22. pick any vertex $v_2 \in V$ such that $\deg_{G_2}(v_2) \geq 1$
23. **if** $v_2 \neq$ null **then**
    24. $V_2$':= Branch($G_2$, $V_2$', $k_2$, $v_2$)
25. **else** //answer "yes" is found and// **return** $V_2$'
26. **return** $V'$
```

The Exhaustive Vertex Cover algorithm can be seen as a procedure for deciding whether a graph G has a vertex cover of size at most k. The algorithm performs an exhaustive search that can be conceptualized as the building of a search tree T (see Figure 2.3 for illustration). To avoid confusion, we refer to the vertices in this search tree as (*search tree*) *nodes*. The root s of the search tree is labeled by the original input G and k, plus the vertex set V', which is initialized as empty (line 1). In line 2 an arbitrary vertex v with at least one incident edge is chosen, and then, if such a vertex indeed exists (as checked by line 3), line 4 calls the procedure Branch.

In lines 9–11 of algorithm Branch, the possibility that v *is* in the vertex cover is considered (called the *left branch*). That is, line 9 includes v in V'_1, line 10 removes v and its incident edges from G resulting in G_1, and in line 11, k is updated as k_1 to note that one vertex has been included in the vertex cover so far. All this is represented by the creation of a new node s_1 in the search tree, which is labeled by (G_1, k_1, V'_1). Then, in line 12, a new vertex v_1 is picked and the procedure Branch is called again; now taking G_1, k_1, V'_1, and v_1 as input. Each time a left branch is created, the vertex set V' is extended (line 9) by the vertex picked in line 12 (or in line 22; depending on whether the parent of the present search tree node is on a left branch or on a right branch). Further, *G* and *k* are updated to reflect the changes in *V'*. Such left branches are terminated if (1) the graph has no edges anymore (as checked in line 12) or (2) *k* has become smaller than 0, in which case $|V'|$ has become larger than *k* (as checked in line 15). If case (1) applies, but not case (2), then, in line 18, it is noted that *V'* is a vertex cover of size at most *k* and the set is outputted in line 18, and the algorithm halts. If, on the other hand, a left branch terminates without reaching line 18, then a right branch is created in lines 19–21.



(G, k, V')
(G_1, k_1, V_1')
(G_2, k_2, V_2')
$(G_{11}, k_{11}, V_{11}')$
$(G_{12}, k_{12}, V_{12}')$
$(G_{21}, k_{21}, V_{21}')$
$(G_{22}, k_{22}, V_{22}')$
fan(T) = 2
depth(T) ≤ n
size(T) is $O(2^n)$

Figure 2.3. Illustration of the Exhaustive Vertex Cover algorithm.
The behavior of the algorithm can be thought of as the construction of an exhaustive search tree whose nodes are labeled by partial candidate solutions for Vertex Cover. If a leaf in this tree is labeled by a vertex cover for *G* of size *k* then the algorithm returns the answer "yes."

Each *right branch* considers the possibility that the picked vertex is *not* in the vertex cover, in lines 19-21. This is represented by the creation of a new node $s_{1\ldots2}$ in the search tree, which is labeled by $(G_{1\ldots2}, k_{1\ldots2}, V'_{1\ldots2})$, with $v_{1\ldots1} \notin V'_{1\ldots2}$. Right branches also terminate if no more edges are left in the graph. However, since *k* does not change in a right branch an analogue to line 15 is not needed here (Note: at the start of the Branch procedure, in line 6, it is always checked if *k* is not yet exceeded). If a right branch is terminated, without a solution found (as checked in line 24), then the algorithm goes back to a previous node in the search tree, for which again a right branch is created. After that, again left branches are created until no longer possible; and everything repeats itself until either (a) a vertex cover of size at most k is found, or (b) no more branching is possible, in which case the algorithm outputs the empty set in line 5, and we conclude that no vertex cover of size *k* exists for *G*.

Note that this behavior of the algorithm corresponds to creating search tree nodes in a particular order. To illustrate, assume that the length of each path from root to leaf in *T* is, say, 4. Then the nodes in *T* are being created in the following order: s_1, s_{11}, s_{111}, s_{1111}, s_{1112}, s_{112}, s_{1121}, s_{1122}, s_{12}, s_{121}, s_{1211}, s_{1212}, s_{122}, s_{1221}, s_{1222}, s_2, s_{21}, s_{211}, s_{2111}, s_{2112}, s_{212}, s_{2121}, s_{2122}, s_{22}, s_{221}, s_{2211}, s_{2212}, s_{222}, s_{2221}, s_{2222}.

In general, we denote the maximum number of children created per node in a search tree by fan(*T*) and the maximum length of the path from root to any leaf in *T* by depth(*T*). The total number of nodes in the search tree is denoted by size(*T*). Note that size(*T*) is bounded by $2\text{fan}(T)^{\text{depth}(T)} - 1$ which is $O(\text{fan}(T)^{\text{depth}(T)})$.

In the particular search tree created by the Exhaustive Vertex Cover algorithm, and illustrated in Figure 2.3, fan(*T*) = 2 and depth(*T*) ≤ *n* (viz., there are never more than *n* vertices to pick). Thus, size(*T*) is $O(2^n)$, which is exponential size. Even though we spend polynomial time (i.e., $O(n^\alpha)$ for some constant α) for creating each node in the tree, the exponential function dominates the running time (i.e., $O(2^n\, n^\alpha)$ is $O(2^n)$), and hence Exhaustive Vertex Cover is an exponential-time algorithm that runs in time $O(2^n)$.

2.4.3. Problem Complexity

In the previous section we defined time complexity for algorithms. Here, we consider the time complexity of problems. A problem Π is said to be solvable in time $O(t(|i|))$, if there exists an algorithm that solves Π and that runs in time $O(t(|i|))$. If the fastest algorithm solving Π runs in time in time $O(t(|i|))$ then we say that Π has time complexity $O(t(|i|))$. Often we do not know what the fastest algorithm for a problem is. In that case, we use the running time of the fastest *known* algorithm for Π as a measure of time complexity of Π.

To illustrate, we again consider the problem Vertex Cover. In the previous section we have seen that the Exhaustive Vertex Cover algorithm solves Vertex Cover, and that this algorithm runs in time $O(2^n)$. Hence, we can safely conclude that Vertex Cover is solvable in time $O(2^n)$. However, before we use $O(2^n)$ as a measure of the inherent complexity of Vertex Cover, we will wish to ensure—or at least have reason to expect—that there does not exist a faster algorithm that also solves Vertex Cover. Now consider, for example, the much faster Greedy Vertex Cover algorithm, which runs in $O(n^2)$. Does the Greedy Vertex Cover algorithm solve the Vertex Cover problem? The answer is a definite "no."[19] Thus, for now we can conclude that Vertex Cover can be solved in time $O(2^n)$, but we remain unsure about whether this is a tight upperbound. In Chapter 3 we will see, however, that whatever the fastest algorithm for Vertex Cover is, it will unlikely be a polynomial-time algorithm.

2.4.4. The Invariance Thesis

The previous sections introduced a notion of (worst-case) time complexity that attempts to abstract away from particular machine details using the $O(.)$ notation. But what does it mean to abstract away from machine details? Not only will the running time depend on the particulars of the Turing machine used for a computation, but also Turing machines are just one class of possible computing machines.

In this context it is useful to mention the Invariance thesis.[20] This thesis states that, given a "reasonable encoding" of the input and two "reasonable machines" M_1 and M_2, the complexity of a given problem Π for M_1 and M_2 will differ by at most a polynomial amount (e.g. Garey & Johnson, 1979). Here, with "reasonable encoding" is meant an encoding that does not contain unnecessary information and in which numbers are represented in b-ary with $b > 1$ (e.g. binary, or decimal, or any fixed base other than 1). Further, with "reasonable machine" is meant any type of Turing machine (with

[19] The reader may want to verify this for him/herself.
[20] Frixione (2001) uses the term 'Invariance thesis' to express both the Invariance thesis itself and, what I will call, the Tractable Cognition thesis. Since I intend to investigate two alternative formalizations of tractability in Chapter 3 (viz., classical and fixed-parameter tractability), whereas Frixione (2001) only considers one option (classical tractability), I purposely divorce the Invariance thesis from the Tractable Cognition thesis.

possible extension as described in Section 2.2.3) or any other realistic computing machine under a different formalization (including e.g. neural networks, cellular automata). Note, however, that a machine capable of performing arbitrarily many computations in parallel (cf. non-deterministic Turing machine or Quantum computer[21]) is not considered a “reasonable” machine (Garey & Johnson, 1979).

Like the Church-Turing thesis, the Invariance thesis is widely accepted among computer scientists and cognitive scientists. The Invariance thesis, if true, implies that we can analyze the worst-case complexity of problems, independent of the machine model, up to a polynomial amount of precision. In other words, if a problem is of polynomial-time complexity under one model it will be of polynomial-time complexity under any other reasonable model. Similarly, if a problem is of exponential-time complexity under one model it will be of exponential-time complexity under any other reasonable model.[22]

Although some computer scientists believe that one day it will be possible to build Quantum computers (allowing for arbitrarily many parallel computations), this still remains to be seen. Furthermore, even if a Quantum computer—or some other computational device of comparable power—were physically realizable, it still need not be the right model of human computation. Whichever will turn out to be the case, for present purposes we will adopt the Invariance thesis.

2.5. The Tractable Cognition Thesis

So far we have talked of “reasonable encoding” and of “reasonable machines,” but what about “reasonable time”? As discussed in Section 2.3.1, critics of the Church-Turing thesis argue that computability does not pose strict enough constraints on human computation, because human cognitive systems have to compute functions in a “reasonable time.” In other words, cognitive functions are *tractably* computable. The notion of tractable computability, like computability was before its formalization as Turing-computability, is an intuitive notion. If we could somehow formally define the class of functions that are tractably computable, and a class of functions that are not, then

[21] See e.g. Deutsch (1985) for a treatment of quantum computation.
[22] In Chapter 3 we will introduce fpt-time algorithms, and we will show that on the Invariance thesis also problems of fpt-time complexity will be of fpt-time complexity under any reasonable model.

we could have a Tractable Cognition thesis, which states that cognitive functions are among the tractably computable functions. Such a thesis could then serve to provide further theoretical constraints on the set of cognitive functions, as illustrated in Figure 2.4.

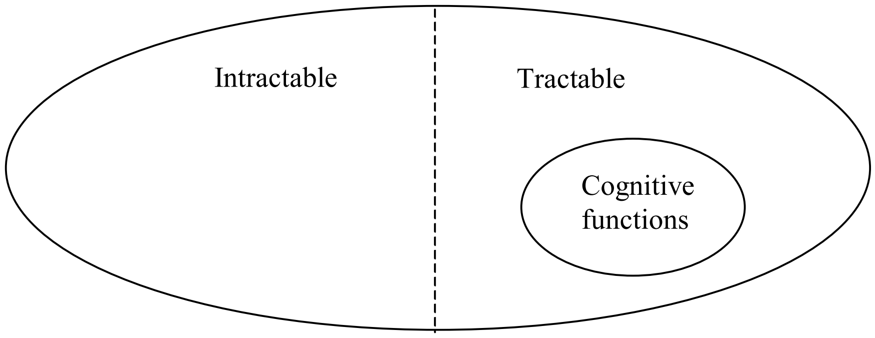


Figure 2.4. Illustration of the Tractable Cognition thesis.
The set of all computable functions partitioned into tractable and intractable functions. According to the Tractable Cognition thesis cognitive functions are a subset of the tractable functions. The dotted line indicates that the set of tractable functions remains to be formalized.

We do not know exactly what type of computational devices human cognitive systems are. Therefore it would be particularly useful if we could define a Tractable Cognition thesis that abstracts away from machine details. That is, like the Church-Turing thesis provides a definition of computability independent of the Turing-machine formalization, we would like to have a Tractable Cognition thesis that provides a definition of tractability independent of the Turing-machine formalization. From the Invariance thesis (Section 2.4.4) we know that if we could define the set of tractable and intractable problems such that the classification is insensitive to a polynomial amount of complexity in computation of the problem, we would have a machine independent formalization of the notion of (in)tractability.

For now we have only formulated the Tractable Cognition thesis as a "mold" for a formal version of the thesis (indicated by the dotted line in Figure 2.4). Chapter 3 investigates two possible formalizations of the Tractable Cognition thesis. The first is called the P-Cognition thesis and the second is called the FPT-Cognition thesis. We will see that both theses abstract away from machine details to achieve the desired generality.

Chapter 3. P-Cognition versus FPT-Cognition

This chapter contrasts two possible formalizations of the Tractable Cognition thesis: the P-Cognition thesis and the FPT-Cognition thesis. I start by discussing classical complexity theory and its motivation to define tractability as polynomial-time computability. I discuss how many cognitive scientists have adopted this classical notion of tractability, leading them to advance the P-Cognition thesis. Then I introduce the theory of parameterized complexity and its accompanying notion of fixed-parameter tractability. Using parameterized complexity, I will illustrate that polynomial-time computability provides a too strict constraint on cognitive functions. I propose the FPT-Cognition thesis and discuss how this thesis applies to cognitive theory formation.

3.1. Classical Complexity and Classical Tractability

Classical complexity theory[23] proposes that the distinction between polynomial-time algorithms and exponential-time algorithms, as introduced in Chapter 2, is a useful (approximate) formalization of the intuitive distinction between tractable and intractable algorithms (e.g. Garey & Johnson, 1979; Papadimitriou & Steiglitz 1988). As Garey and Johnson (1979, p. 8) put it:

> "Most exponential time algorithms are merely variations on exhaustive search, whereas polynomial time algorithms generally are made possible only through the gain of some deeper insight into the nature of the problem. There is wide agreement that a problem has not been "well-solved" until a polynomial time algorithm is known for it. Hence, we shall refer to a problem as *intractable*, if it is so hard that no polynomial time algorithm can possibly solve it."

Hence, we have the following classical definition of (in)tractability. A problem Π is said to be (*classically*) *tractable* if Π can be solved by a polynomial-time algorithm. [24]

---

[23] The reader is advised that what I call classical complexity theory is typically referred to as (computational) complexity theory in both the computer science and cognitive science literature. Because I wish to contrast this theory with a newer form of complexity theory, called parameterized complexity theory, I refer to the earlier theory as "classical."

[24] One may argue that not all polynomial-time algorithms are preferable to all exponential-time algorithms (e.g. a running time of $O(2^n)$ would in theory be preferred to

Conversely, a problem Π is said to be (*classically*) *intractable* if Π cannot be solved by a polynomial-time algorithm (i.e., it requires an algorithm that runs in exponential (or worse) time).

The classical definition of tractability is widely adopted in computer science (e.g. Garey & Johnson, 1979; Papadimitriou & Steiglitz, 1988; or any introductory text in theoretical computer science). To see that the definition has merit, consider Table 2.1. Table 2.1 shows that an exponential function, like $O(2^{|i|})$ grows every so much faster than a polynomial function, like $O(|i|^2)$. If we assume that a computing machine can compute, say, 100 basic computations per second then, as $|i|$ grows, the exponential running time gets unrealistic very fast for any reasonable computing machine (compare Columns 2 and 3 of Table 2.1).

Table 2.1. Classical tractability and intractability
Illustration of how a polynomial running time, $O(|i|^2)$, and an exponential running time, $O(2^{|i|})$, compare for different $|i|$; assuming either 100 or 10,000 computational steps per second.

	Assume 100 steps/sec.		Assume 10,000 steps/sec.	
$\|i\|$	$O(\|i\|^2)$	$O(2^{\|i\|})$	$O(\|i\|^2)$	$O(2^{\|i\|})$
2	0.04 sec	0.04 sec	0.02 msec	0.02 msec
5	0.25 sec	0.32 sec	0.15 msec	0.19 msec
10	1.00 sec	10.2 sec	0.01 sec	0.10 sec
15	2.25 sec	5.46 min	0.02 sec	3.28 sec
20	4.00 sec	2.91 hrs	0.04 sec	1.75 min
30	9.00 sec	4.1 mths	0.09 sec	1.2 days
50	25.0 sec	8.4 x 10^4 yrs	0.25 sec	8.4 centuries
100	1.67 min	9.4 x 10^{19} yrs	1.00 sec	9.4 x 10^{17} yrs
1000	2.78 hrs	7.9 x 10^{290} yrs	1.67 min	7.9 x 10^{288} yrs

a running time of, say, $O(n^{100})$). It turns out, however, that problems that are computable in polynomial time typically allow for algorithms with small constants in the exponent. This suggests that polynomial-time computability to some extent captures our intuitive notion of computational tractability. Even more important for present purposes, however, is the fact that it does not matter that some polynomial-time algorithms are impractical, as long as it is safe to say that all exponential-time algorithms are impractical.

Note that here the assumption of 100 basic operations per second is for illustrative purposes only. In the context of cognitive theory, what constitutes a reasonable assumption depends, of course, on both the cognitive function under consideration and the hardware of the system that is presumed to compute the function. Importantly, however, increasing the speed with which a basic computational operation can be performed has very limited impact on the time required to compute a function if the function is of exponential-time complexity. For example, if we could realize a 100-fold speed-up of our computing machine (e.g. by improving its hardware so that it can perform each sequential operation 100 times faster; or by having it run 100 computational channels in parallel), then an exponential time algorithm running in time $O(2^{|i|})$ would still be impractical for all but trivially small input sizes (compare Columns 4 and 5 of Table 2.1).

3.2. The Theory of NP-completeness

Section 3.3 discusses how the classical notion of tractability has been applied in cognitive psychology. Many of these applications use the theory of NP-completeness to infer classical (in)tractability of cognitive functions. Therefore I first explain NP-completeness (Section 3.2.1) and its accompanying notion of polynomial-time reducibility (Section 3.2.2). For more information see also Cook (1971), Karp (1972), Garey and Johnson (1979), and Papadimitiou and Steiglitz (1988). Readers familiar with the theory of NP-completeness can skip this section without loss of continuity.

3.2.1. The classes P and NP

Let Π be a decision problem. Then Π is said to belong to class P (we write $\Pi \in$ P) if Π can be decided by a (deterministic) Turing machine (DTM) in polynomial-time; and Π is said to belong to class NP ($\Pi \in$ NP) if all yes-instances for Π can be decided by a non-deterministic Turing machine (NTM) in (non-deterministic) polynomial time.[25] An NTM

[25] For completeness I note that a decision problem Π is said to belong to class co-NP ($\Pi \in$ co-NP) if all no-instances for Π can be decided by a non-deterministic Turing machine in (non-deterministic) polynomial-time problem. The class co-NP will not be considered further here.

decides a yes-instance *i* for Π if there exists at least one possible sequence of transitions, as defined by the machine's transition relation, that returns "yes" for *i* (cf. Section 2.2.3).

Many problems that are encountered in practice, and many of the cognitive functions that are considered in later chapters, belong to the class NP. Note that also Vertex Cover is in the class NP. Namely, in Section 2.4.2, we saw that the Exhaustive Vertex Cover algorithm can be used to decide the problem Vertex Cover. Now imagine a NTM with a transition relation that, analogous to the Exhaustive Vertex Cover algorithm, defines two possible transitions for each vertex *v* in the input graph *G*: either (1) *v* is in the vertex cover, or (2) *v* is not in the vertex cover. Clearly, if a vertex cover of size *k* exists for *G* then at least one sequence of transitions in this non-deterministic machine will return such a vertex cover.

The class NP is also sometimes referred to as the class of problems for which solutions are "easy to check." This is because, when given a candidate solution for a yes-instance of a problem $\Pi \in$ NP, one can check in polynomial time whether the candidate solution is indeed a solution for Π. For example, when given a vertex set $V' \subseteq V$ one can easily check in polynomial time whether (a) *V'* is a vertex cover for *G* (we delete every vertex in *V'* from *G* and see if any edges are left in *G*; this can be done in $O(n^2)$) and (b) its size is at most *k* (we count the number of vertices in *V'*; this can be done in time $O(n)$).

Since a DTM is a special kind of NTM (viz., one in which the transition relation is a function), we have P $\subseteq$ NP. It is widely believed that there exist problems in NP that are not in P, and thus that P $\neq$ NP (see Figure 3.1). This conjecture is motivated, among other things, by the existence of so-called NP-hard problems.

To explain NP-hardness we define the notion of polynomial-time reducibility: For problems Π_1 and Π_2 we say that Π_1 *reduces* to Π_2, if there exists an algorithm that transforms any input i_1 for Π_1 into an input i_2 for Π_2 such that input i_1 is a yes-instance for Π_1 if and only if input i_2 is a yes-instance for Π_2. We say the reduction is a *polynomial-time reduction* if the algorithm performing this transformation runs in polynomial time. A decision problem Π is NP-*hard* if every problem in NP can be polynomial-time reduced to Π. Thus, NP-hard problems are not in P unless P = NP. Problems that are NP-hard *and* members of NP are called NP-*complete*.

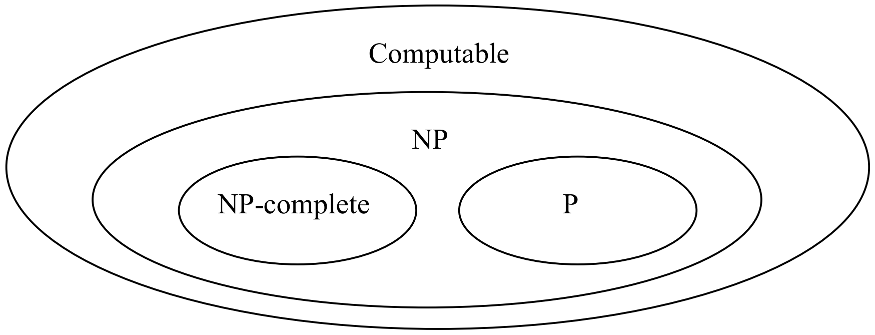


Figure 3.1. The view of NP on the assumption that P ≠ NP.
NP-complete problems are not in P if and only if P ≠ NP.

The technique of polynomial-time reduction is very useful. Once a problem is known to be NP-hard we can prove other problems to be NP-hard by polynomial-time reducing it to these other problems. In 1971, Stephen Cook proved the first problem NP-hard. This problem is called Satisfiability, abbreviated SAT (see Appendix B for its problem definition). Since then many problems have been shown to be NP-hard by (direct or indirect) reduction from SAT, and presently hundreds of problems are known to be NP-hard. Among them is the problem Vertex Cover (Karp, 1972).

Despite great efforts from many computer scientists, nobody to date has succeeded in finding a polynomial-time algorithm that solves an NP-hard problem (hence, the belief that P ≠ NP). Therefore the finding that a decision problem Π is NP-hard is seen as very strong evidence that all algorithms solving Π run at best in exponential-time; e.g., in time $O(\alpha^n)$, where α is a constant and n is the input size.

3.2.2. Illustrating polynomial-time reduction

This section illustrates the technique of polynomial-time reduction: We reduce the problem Vertex Cover to a different graph problem, called Dominating Set. A vertex set $V' \subseteq V$ is called a *dominating set* for $G = (V, E)$ if for every vertex in V either $v \in V'$ or v has a neighbor $u \in V'$.

Dominating Set

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Does there exist a dominating set V' for G with $|V'| \leq k$?

As mentioned above, Vertex Cover is NP-hard. Hence, a polynomial-time reduction from Vertex Cover to Dominating Set proves that Dominating Set is NP-hard too.[26] Recall that, to establish a reduction from Vertex Cover to Dominating Set, we need to present a transformation from any instance $i = \{G, k\}$ for Vertex Cover to an instance $i^* = \{G^*, k^*\}$ for Dominating Set such that $i = \{G, k\}$ is a yes-instance for Vertex Cover if and only if $i^* = \{G^*, k^*\}$ is a yes-instance for Dominating Set.

For simplicity, in the following we assume that the input graph G is of minimum degree 1 (i.e., G does not contain any singletons). The assumption is validated by the fact that Vertex Cover is also NP-hard for graphs without singletons.[27] We start by making two observations.

Observation 3.1. Let $G = (V, E)$ be a graph without singletons and let k be a positive integer. If G and k form a yes-instance for Vertex Cover then G and k form a yes-instance for Dominating Set.

Proof: Let G and k form a yes-instance for Vertex Cover. Then there exists a vertex set $V' \subseteq V$ such that V' is a vertex cover for G and $|V'| \leq k$. This means that for every edge $(u, v) \in E$, $u \in V'$ or $v \in V'$. But then also for every vertex $v \in V$ either v in V' or v has at least one neighbor $u \in V'$. Thus V' is a dominating set for G with $|V'| \leq k$. We conclude G and k form a yes-instance for Dominating Set. ■

Observation 3.2. Let $G = (V, E)$ be a graph without singletons and let k be a positive integer. If G and k form a yes-instance for Dominating Set then G and k *need not* form a yes-instance for Vertex Cover.

Proof: We prove the claim by example. Let $G = (V, E)$ with $V = \{u, v, w\}$ and $E = \{(u, v), (v, w), (u, w)\}$ and let $k = 1$. Then G and k form a yes-instance for Dominating Set (e.g., vertex set $\{u\}$ is a dominating for G of size 1), but G and k do not form a yes-instance for Vertex Cover (i.e., the smallest vertex cover for G is of size 2, e.g. $\{u, v\}$). ■

[26] Of course, it is long known that Dominating Set is NP-hard (e.g. Garey & Johnson, 1979). The polynomial-time reduction discussed in this section is for illustrative purposes only.
[27] Singletons are easily dealt with, both in Vertex Cover (where they can always be excluded from the vertex cover) and Dominating Set (where they always need to be included in the dominating set).

From these observations we conclude that a reduction from Vertex Cover to Dominating Set of the form $G^* = G$ and $k^* = k$ will not work; viz., although all vertex covers are dominating sets (Observation 1), the reverse is not true (Observation 2). If we could somehow define G^* such that every dominating set for G^* is also a vertex cover for G, without losing the relationship between the two problems as stated in Observation 1, then we would be able to establish a proper reduction. Lemma 3.1 shows that indeed the following transformation works: First we set $G^* = G$ and then for every edge (u, v) in G we add a new vertex x to G^* and connect it to the vertices u and v in G^* with the edges (x, u) and (x, v). See Figure 3.2 for an illustration.

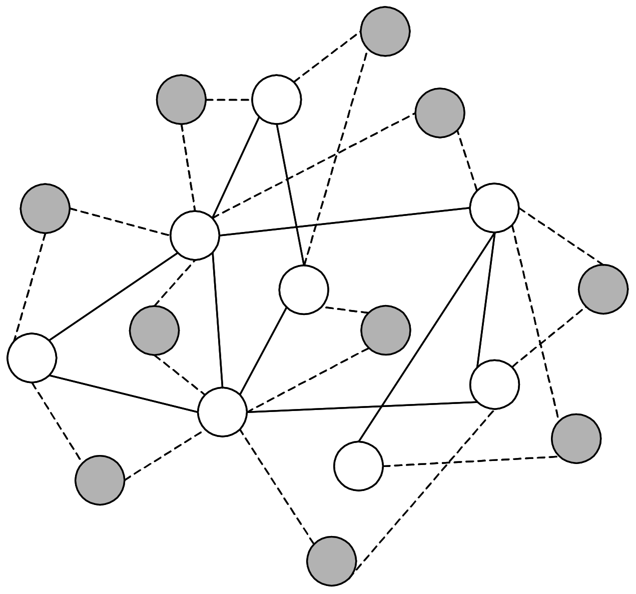


Figure 3.2. Illustration of the reduction in Lemma 3.1.
The graph G consists of the white vertices and the solid edges, and graph G^* consists of G plus the gray vertices and the dotted edges. Note that G has a vertex cover of size k if and only if G^* has a dominating set of size k.

Lemma 3.1. Let $G = (V, E)$ be a graph without singletons and let k be a positive integer. Further, let $G^* = (V^*, E^*)$ with $V^* = V \cup A$, where $A = \{x_{uv} \notin V : (u, v) \in E\}$, and $E' = E \cup E^*$, where $E^* = \{(y, z) : y = x_{uv}$ and $z = u$ or $y = x_{uv}$ and $z = v)$ and let $k^* = k$. Then G and k form a yes-instance for Vertex Cover if and only if G^* and k^* form a yes-instance for Dominating Set.

Proof: ($\Rightarrow$) Let G and k form a yes-instance for Vertex Cover. Then there exists a vertex set $V' \subseteq V$ such that V' is a vertex cover for G and $|V'| \leq k$. This means that for every edge $(u, v) \in E$, $u \in V'$ or $v \in V'$. From Observation 1 we know that for every vertex $v \in V$, $v \in V'$ or v has at least one neighbor $u \in V'$. Further, because every vertex $x_{uv} \in A$ is adjacent to both $u \in V$ and $v \in V$, with $(u, v) \in E$, we also know every vertex $x_{uv} \in A$ has at least one neighbor in V'. Thus V' is a dominating set for G^* with $|V'| \leq k = k^*$. We conclude G^* and k^* form a yes-instance for Dominating Set.

($\Leftarrow$) Let G^* and k^* form a yes-instance for Dominating Set. Then there exists a vertex set $V' \subseteq V^*$ such that V' is a dominating set for G^* and $|V'| \leq k^*$. This means that for every vertex $v \in V^*$, $v \in V'$ or v has a neighbor $u \in V'$. We prove that there exists a vertex cover $V'' \subseteq V$ for G with $|V''| \leq |V'| \leq k$. We distinguish two cases (1) Let $V' \subseteq V$. Then we consider $V'' = V'$. Note that for V'' to dominate all vertices in A in G^*, V'' has to cover every edge in E (viz. every edge in E has an associated vertex $x_{uv} \in A$). Hence, V'' is a vertex cover for G of size at most $k^* = k$. We conclude that (G, k) is a yes-instance for Vertex Cover. (2) Let $V' \not\subset V$. Then we transform V' into a set V'' as follows. For each vertex $x_{uv} \in V'$, with $x_{uv} \notin V$, we remove x_{uv} from V' and replace it by u or v (or neither, if both already happen to be in V'). Note that the resulting set V'' is also a dominating set for G^* and $|V''| \leq |V'|$. Further, $V'' \subseteq V$ and thus case (1) applies to $V' = V''$. ■

Note that the transformation from G and k to G^* and k^* in Lemma 3.1 can be done in polynomial time: To create G^* we copy G in time $O(n^2)$ and we then add at most m vertices and $2m$ edges to G^* in time $O(n^2)$. Hence, with Lemma 3.1 we have shown a polynomial-time reduction from Vertex Cover to Dominating Set. Since, Vertex Cover is known to be NP-hard, we can conclude from Lemma 3.1 that Dominating Set is NP-hard. In other words, we have shown that if Dominating Set is in P, then so is Vertex Cover (and so are all other problems in NP). Namely, if there exists a polynomial-time algorithm M_1 that solves Dominating Set, then we can solve Vertex Cover with a polynomial-algorithm $M_2 = RM_1$ that consists of M_1 preceded by a sub-procedure R that transform the instance for Vertex Cover into an equivalent instance for Dominating Set.

3.3. The P-Cognition thesis

Classical complexity theory, including the theory of NP-completeness, has found its way into cognitive psychology. Its application spans many cognitive domains, including: learning (Judd, 1990; Parberry, 1994; 1997), judgment and prediction (Martignon & Schmitt, 1999), categorization (Anderson, 1990), decision-making (Payne, Bettman, & Johnson, 1993; Simon, 1988, 1990), knowledge representation (Parberry, 1997), reasoning (Cherniak, 1986; Levesque, 1988; Millgram, 2000; Oaksford & Chater, 1993, 1998; Thagard, 2000; Thagard & Verbeurgt, 1998), linguistic processing (Barton, Berwick, & Ristad, 1987; Ristad, 1993, 1995; Wareham, 1996, 1998), visual perception (Kube, 1991; Tsotsos, 1988, 1989, 1990, 1991), and visual problem-solving (Graham, Joshi, & Pizlo, 2000; MacGregor & Ormerod, 1996; MacGregor, Ormerod, & Chronicle, 1999, 2000; van Rooij, Schactman, Kadlec, & Stege, 2003; van Rooij, Stege, & Schactman, 2003; Vickers, Butavicius, Lee, & Medvedev, 2001).

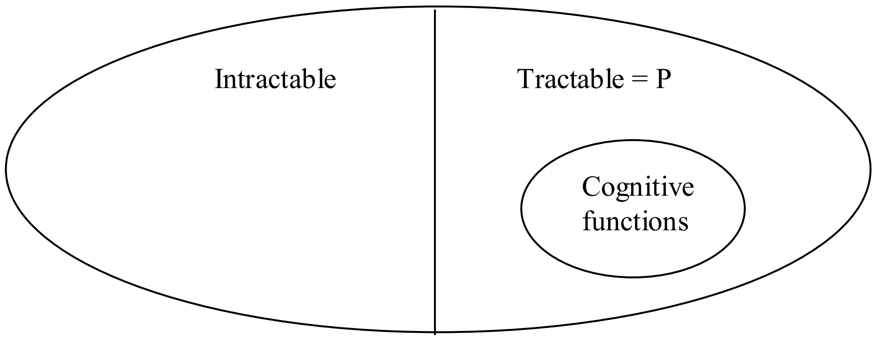


Figure 3.3. Illustration of the P-Cognition Thesis.
The set of all computable functions partitioned into tractable and intractable functions based on the definition of classical (in)tractability. On the P-Cognition thesis any function outside P can be rejected as computational level theory of a cognitive system.

In present-day cognitive psychology classical complexity theory is used, among other things, to evaluate the feasibility of computational theories of cognition. One of the first researchers to explicate this usage of complexity theory in cognitive psychology is Frixione (2001). Frixione made the case for the thesis that cognitive functions are among the polynomial-time computable functions, and he called upon cognitive psychologists to use this thesis to constrain computational level theories. I will refer to Frixione's proposed formalization of the Tractable Cognition thesis as the P-Cognition thesis (see Figure 3.3 for an illustration). In some form or another, the P-Cognition thesis has been advanced by many researchers in cognitive psychology (e.g. Anderson, 1990; Cherniak, 1986, Judd, 1990; Levesque, 1988; Martignon & Schmitt, 1999; Millgram, 2000; Oaksford & Chater, 1993, 1998; Parberry, 1994; Simon, 1988, 1990; Thagard, 2000; Thagard & Verbeurgt, 1998; Tsotsos, 1988, 1989, 1990, 1991), as well as in artificial intelligence (see e.g. Cooper, 1990; Nebel, 1996). The application of the P-Cognition thesis in psychological practice will be discussed below as well as in later chapters (see Chapters 5 and 7).

3.3.1. P-Cognition in Psychological Practice

The P-Cognition thesis is presently employed in at least three qualitatively different ways. First, some researchers view the finding that a given cognitive function is not in P (assuming $P \neq NP$) as a reason to abandon the associated computational level theory altogether. For example, in the domain of reasoning, Oaksford and Chater (1993, 1998) argued that logicists' approaches to modeling commonsense reasoning are untenable and should be abandoned, because checking whether a set of beliefs is logically consistent is NP-hard. Comparable reactions to NP-hardness results can be found in Martignon and Schmitt (1999) and Millgram (2000).

Other researchers view complexity theory as a tool for refining (not all-round rejecting) computational level theories such that they satisfy tractability constraints. Researcher in this group reason as follows: If a computational level theory Π is classically intractable, then the cognitive system cannot be solving Π in its generality; hence it must (or might) be solving some variant or special case of Π. For example, Levesque (1988), like Oaksford and Chater, recognized the inherent exponential-time complexity of general logic problems, but unlike Oaksford and Chater, he concludes that we need to adjust logic, not abandon it, in order to obtain psychologically realistic models of human reasoning. Similarly, upon finding that visual search, in its general (bottom-up) form, is NP-complete, Tsotsos (1988, 1989, 1990, 1991) did not abandon his model of vision, but instead adjusted it by assuming that top-down information helps constrain the visual search space.

Lastly, there are researchers that, upon finding that a cognitive function Π is of exponential-time complexity, do not reject Π as computational level theory, nor adjust it to accommodate tractability constraints. Instead they assume that, at the algorithmic level, the function Π is being computed by heuristics or approximation algorithms. This approach is taken, for example, by Thagard and Verbeurgt in the domain of abductive (coherence) reasoning (1998; Thagard, 2000; but see also Martignon and Schmitt, 1999). This last group, in a sense, does not take the computational level theory very seriously; i.e., the constraint that tractability places upon algorithmic level theories is recognized, but not the constraints that tractability places upon computational level theories.

3.3.2. A Comment on Psychological Practice

I briefly comment on the validity of the three approaches sketched above. In my view, the only sensible approach is the second one: An intractability result for Π indicates that the function Π cannot be practically computed in all its generality, and this knowledge should be used to reshape the cognitive theory Π into a cognitive theory Π' that is of more realistic form.

The first approach is unreasonable simply because no single intractability result for a function Π can overthrow the entire framework in which Π was formulated.[28] In my response to the third approach I distinguish between algorithmic level descriptions that are heuristics for the computational level function Π and those that are approximation algorithms for Π.[29] If we were to accept (mere) heuristics as algorithmic level descriptions for the computational level function Π then we can do away with the computational level description Π altogether. Namely, then there is no principled relationship between the algorithmic level and the computational level description. On

[28] Interestingly, Oaksford and Chater (1998) side with the framework of Bayesian inference, which is—as they themselves admit (p. 289)—equally plagued by computational intractability results.

[29] I use the words 'heuristic' and 'approximation algorithm' as mutually exclusive. An *approximation algorithm* is a procedure that guarantees a bound on the extent to which the computed output deviates from the required output. A *heuristic* is an approach that (intuitively) seems to give approximate solutions to a problem, but does not provably do so. This distinction between heuristics and approximation algorithms is important and is often overlooked in the cognitive science literature.

the other hand, if one wishes to take an approximation approach then I would propose the following: First define the adequate level of approximation that would be needed for the cognitive system to perform the task under study, and then incorporate this as part of the computational level theory. This way, the algorithmic level does not approximate the computational level; instead the computational level is reformulated as an approximation problem.

3.4. Parameterized Complexity and Fixed-Parameter Tractability

This section presents a brief introduction to the theory of parameterized complexity as developed by Downey and Fellows (1999; see also Downey, Fellows, & Stege, 1999a, 1999b; Fellows, 2002). A detailed discussion of techniques for parameterized complexity analysis is postponed until Chapter 4. This section primarily serves to introduce the notion of fixed-parameter tractability, to be used in Section 3.5.

3.4.1. The classes FPT and W[1]

We have seen that classical complexity theory analyzes the complexity of problems in terms of the size of the input, measured by $|i|$. In contrast, parameterized complexity theory allows for a more fine-grained identification of sources of complexity in the input. This is done by analyzing complexity in terms of the size of the input, $|i|$, *plus* some input parameter(s), κ.

Inputs to decision problems often have several (explicit and implicit) aspects and any one of them may be considered an input parameter. For example, Vertex Cover and Dominating Set both have two aspects that are explicitly stated in the input: a graph G and a positive integer k. The problems also have several implicit parameters; e.g., the maximum (or minimum) degree of vertices in G, the length of the longest path in G, the number of cycles in G, the ratio between the number of edges and vertices in G. All of these aspects may be considered a problem parameter, and a problem may be parameterized on one or more parameters at the same time.

We denote a parameter set by $\kappa = \{k_1, k_2, ..., k_m\}$, where k_i, $i = 1, 2, \ldots, m$, denotes an aspect of the input. We denote a problem Π parameterized on κ by κ-Π, and call κ-Π a *parameterized problem*. An instance for κ-Π, with input *i* and parameter κ, is denoted by a tuple (*i*, κ).

When a problem is shown to be NP-hard it is important to understand which aspects of the input are actually responsible for the non-polynomial time behavior of the problem. The theory of parameterized complexity is motivated by the following observation. Some NP-hard problems can be solved by algorithms whose running time is non-polynomial in some parameter κ but polynomial in the input size |*i*|. In other words, the main part of the input contributes to the overall complexity in a "good" way, while only κ contributes to the overall complexity in a "bad" way. In these cases, we say κ *confines* the non-polynomial time complexity in the problem, and the problem is said to be fixed-parameter tractable for parameter κ.

More formally, a parameterized problem κ-Π is said to be *fixed-parameter tractable* if any instance (*I*, κ) for κ-Π can be decided in time $O(f(\kappa)n^{\alpha})$, where $f(\kappa)$ is a function depending only on κ. An algorithm that solves a parameterized problem κ-Π in time $O(f(\kappa)n^{\alpha})$ is called a fixed-parameter tractable (fpt-) algorithm. Parameterized decision problems that are not fixed-parameter tractable are called *fixed-parameter intractable*.

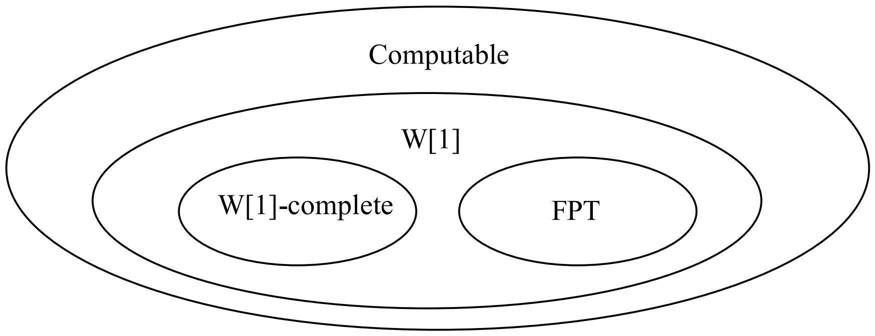


Figure 3.4. The view of W[1] on the assumption that FPT ≠ W[1]. W[1]-complete problems are not in FPT if and only if FPT ≠ W[1].

Analogous to the classical complexity classes P and NP, parameterized complexity theory introduces the parameterized complexity classes FPT and W[1], with FPT ⊆ W[1].[30]

[30] See e.g. Downey and Fellows (1999) for a definition of the class W[1].

Fixed-parameter tractable parameterized problems are said to be in the class FPT. It is widely believed that there exist parameterized problems in W[1] that are fixed-parameter intractable, and thus that FPT ≠ W[1] (see also Figure 3.4). This conjecture is, among other things, motivated by the observation that there exist W[1]-hard problems.

To explain W[1]-hardness we define the notion of parametric reduction:[31] For parameterized problems κ_1-Π_1 and κ_2-Π_2 and functions f and g, we say a *parametric reduction* from κ_1-Π_1 to κ_2-Π_2 is a reduction that transforms any instance i_1 for κ_1-Π_1 into an instance i_2 for κ_2-Π_2 with $\kappa_2 = g(\kappa_1)$. Further, the reduction runs in time $f(\kappa_1)|i_1|^\alpha$, where α is a constant. A parameterized problem κ-Π is said to be W[1]-*hard* if any parameterized problem κ'-$\Pi' \in$ W[1] can be transformed to κ-Π via a parametric reduction (problems that are W[1]-hard *and* in W[1] are called W[1]-*complete*). Since, membership of a W[1]-hard problem in FPT would imply that FPT = W[1], the finding that a problem is W[1]-hard is seen as very strong evidence that the problem is not in FPT.

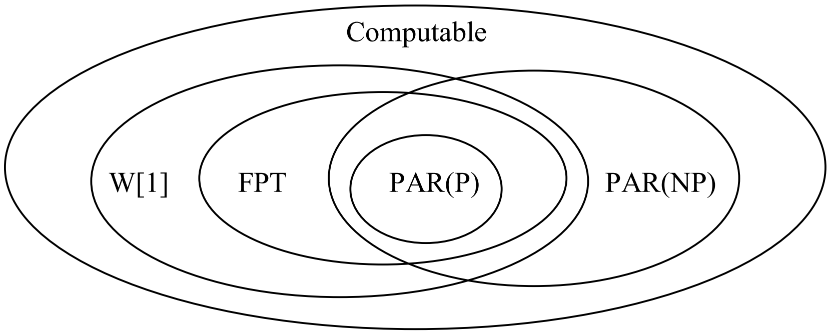


Figure 3.5. Illustration of the relationship between classes W[1], FPT, P and NP. Because W[1] and FPT are classes of parameterized problems we cannot compare them directly to the classes P and NP. Therefore we define the class of all possible parameterizations of problems in P, denoted PAR(P), and the class of all possible parameterizations of problems in NP, denoted PAR(NP). Since P $\subseteq$ NP, we have PAR(P) $\subseteq$ PAR(NP). Further, since every problem in P is in FPT for any parameter we have PAR(P) $\subseteq$ FPT $\subseteq$ W[1].

When relating Figure 3.4 to Figure 3.1, it is important to keep in mind that W[1] and FPT are classes of parameterized problems, while NP and P are classes of (non-

[31] The technique of parametric reduction is illustrated in Chapter 4.

parameterized) problems. The relationship between W[1] and NP can be understood as follows. Let PAR(NP) denote the set of all possible parameterizations of problems in NP, and let PAR(P) denote the set of all possible parameterizations of problems in P, then PAR(P) $\subseteq$ PAR(NP) and PAR(P) $\subseteq$ FPT $\subseteq$ W[1]. Figure 3.5 illustrates these relationships. Note that all problems that are solvable in polynomial time are solvable in fpt-time for all possible parameterizations. Since parameterization does not change the nature of a problem (it only specifies how its complexity is to be analyzed) we could say that "P $\subseteq$ FPT."

3.4.2. Illustrating Fixed-parameter Tractability

To illustrate the qualitative difference between fixed-parameter tractability and fixed-parameter intractability, we consider the problems Vertex Cover and Dominating Set when parameterized by *k*. Both of these problems can be solved by an exhaustive search that simply checks for every vertex set of size *k* whether or not it is a vertex cover (dominating set) for *G*. For a graph on *n* vertices there exist $\binom{n}{k}$ such subsets. Since $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ is $O(n^k)$ we can solve Vertex Cover and Dominating Set in time $O(n^k)$. Notably, it is possible to decide the problem Vertex Cover must faster than this. Namely, using a technique from parameterized complexity theory—to be discussed in Chapter 4—we can show that Vertex Cover is solvable in time $O(2^k n)$.[32] Thus the parameterized problem *k*-Vertex Cover is in FPT. A similar running time is most likely not possible for Dominating Set: viz., the parameterized problem *k*-Dominating Set has been shown to be W[1]-hard (Downey & Fellows, 1999).

Table 2.2 illustrates how the running times $O(2^{|i|})$, $O(|i|^\kappa)$ and $O(2^\kappa|i|)$ compare, for $\kappa = 10$ and $\kappa = 20$ and different $|i|$. Clearly, an fpt-algorithm that runs in time $O(2^\kappa|i|)$ is feasible even for large $|i|$ provided only that κ is not too large.

[32] In fact, with the use of further techniques and a better analysis, we can show that *k*-Vertex Cover is solvable in time $O(1.28^k + kn)$ (Chen, Kanj, & Jia, 2001).

The take home message of this illustration is that the classical characterization of exponential-time algorithms as "all bad" is too crude: Not all exponential-time algorithms are created equal (cf. Downey & Fellows, 1999; Niedermeier, 2003). Namely, we have seen that the known NP-hard problem Vertex Cover can be solved by an exponential-time algorithm $O(2^k n)$ where the running time is exponential only in *k*. Since the size of *k* is but one (relatively small) part of the input to this problem, it seems inappropriate to consider Vertex Cover as generally intractable.

Table 2.2. Fixed-parameter tractability and intractability.
Illustration of how the running time $O(2^{|i|})$, $O(|i|^{\kappa})$ and $O(2^{\kappa}|i|)$ compare for different levels of $|i|$, when $\kappa = 10$ or $\kappa = 20$ (assuming 10,000 computational steps per second)

		$\kappa = 10$		$\kappa = 20$	
$\lvert i\rvert$	$O(2^{\lvert i\rvert})$	$O(\lvert i\rvert^{\kappa})$	$O(2^{\kappa}\lvert i\rvert)$	$O(\lvert i\rvert^{\kappa})$	$O(2^{\kappa}\lvert i\rvert)$
2	0.02 msec	0.10 sec	0.20 sec	1.75 min	3.5 min
5	0.19 msec	16.3 min	0.51 sec	3 centuries	8.7 min
10	0.10 sec	11.6 days	1.02 sec	3.2×10^8 yrs	17.5 min
15	3.28 sec	22 mths	1.54 sec	1.1×10^{12} yrs	26.2 min
20	1.75 min	32.5 yrs	2.05 sec	3.3×10^{14} yrs	35.0 min
30	1.2 days	19 centuries	3.07 sec	1.1×10^{18} yrs	52.4 min
50	35 centuries	3.1×10^5 yrs	5.12 sec	3.0×10^{22} yrs	1.45 hrs
100	4.0×10^{18} yrs	3.2×10^8 yrs	10.2 sec	3.2×10^{28} yrs	2.9 hrs
1000	3.4×10^{290} yrs	3.2×10^{18} yrs	1.71 min	3.2×10^{48} yrs	29 hrs

3.5. The FPT-Cognition thesis

The theory of parameterized complexity remains largely unknown to and unexplored by cognitive psychologists (to my knowledge, the first and only connection drawn is by Wareham, 1996, 1998).[33] Studying the complexity of different parameterizations of a problem is of great use for cognitive psychology, however, because it gives us insight

[33] Tsotsos (1990) also emphasized that different input parameters may differentially contribute to a problem's complexity, but he did not use parameterized complexity theory in his analyses. Further, note that parameterized complexity theory is already being applied in the field of artificial intelligence (e.g. Gottlob, Scarcello, & Sideri, 2002).

into how the complexity of a problem depends on its different input parameters. If a problem of exponential-time complexity allows for this exponential-time complexity to be confined to parameters that are in practice small, then the problem is in practice not as hard as a classical complexity analysis would suggest. This observation leads me to formulate the FPT-Cognition thesis: Cognitive functions are among the functions that have problem parameters that are "small enough" in practice and that are fixed-parameter tractable for (a subset of) those parameters.

A comment on the qualification "small enough" is in order. First, note that every problem in NP is in FPT for at least one parameter; e.g., when $\kappa = |i|$. The FPT-Cognition thesis is purposely formulated to exclude such trivial cases of fixed-parameter tractability. Furthermore, the range of feasibility for κ is different for an fpt-algorithm that runs in $O(2^{k}n)$ than for an fpt-algorithm that runs in $O(2^{k!}n)$. In sum, whether an fpt-algorithm runs fast enough in practice will depend on (1) the exact function $f(\kappa)$ in the running time, and (2) the range of values that κ may take in practice. The formal theory of parameterized complexity can help determine bounds on the function $f(\kappa)$, but empirical observation and psychological theory will be needed to provide reasonable estimates of bounds on κ.

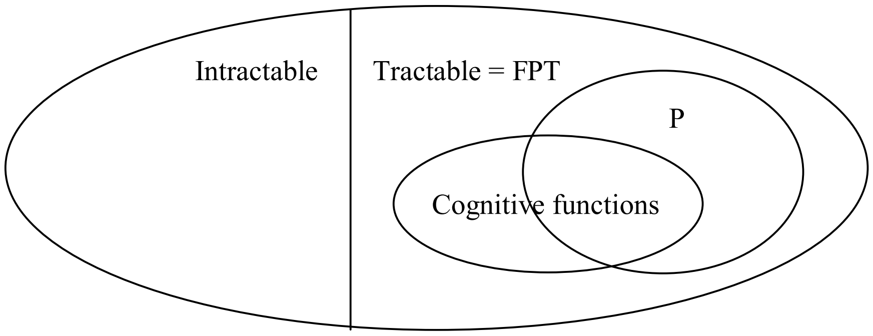


Figure 3.6. Illustration of the FPT-Cognition thesis.
The set of all computable functions partitioned into tractable and intractable functions based on the definition of parameterized (in)tractability. On the FPT-Cognition thesis any computational theory instantiating a function that is not in FPT for some small input parameters can be rejected.

See Figure 3.6 for an illustration of the FPT-Cognition thesis. Since "P $\subseteq$ FPT" the FPT-Cognition thesis instantiates a relaxation of the P-Cognition thesis. That is, under the FPT-Cognition thesis the space of feasible computational theories is larger than the set of feasible theories under the P-Cognition thesis.

Why relax the P-Cognition thesis? If we use computational complexity theory to constrain psychological theorizing we do not want to constrain it too much. That is, we do not want to risk rejecting veridical theories simply because our formalization of the Tractable Cognition thesis is wrong. If the FPT-Cognition thesis is realistic (and I believe it is) then, if we adopt the P-Cognition thesis, we risk exactly this.

3.5.1. FPT-Cognition in Psychological Practice

I close this chapter with a discussion of how the FPT-Cognition thesis might be usefully employed in psychological practice. In this context, it is important to understand that the analytic tools provided by parameterized complexity theory are not meant to replace the tools provided by classical complexity theory; instead the parameterized tools should be seen as extending the classical tools.

With the FPT-Cognition thesis I do not mean to argue that NP-hardness results are of no significance to psychological science. The FPT-Cognition thesis, like the P-Cognition thesis, recognizes that an NP-hard function Π cannot be practically computed in all its generality. If the system is computing Π at all, then it must be computing some "restricted" version of it, denoted Π'. The crux is, however, what is meant by "restricted." The P-Cognition thesis states Π' must be polynomial-time computable, whereas the FPT-cognition thesis states that Π' must have problem parameters that are in practice "small" and that Π' must be fixed-parameter tractable for (a subset of) those parameters.

On the one hand, the FPT-Cognition thesis loses in formality by allowing an undefined notion of "small" parameter in its definition. On the other hand, this allowance is exactly what brings the FPT-Cognition thesis much closer to psychological reality. Many natural cognitive functions have input parameters that are of qualitatively different sizes. Ignoring these qualitative differences, and treating the input always as one big "chunk," would make complexity analysis in psychological practice completely vacuous.

The FPT-Cognition thesis, then, should not be seen as a simple litmus test for distinguishing feasible from unfeasible computational level theories. On the contrary, the FPT-Cognition thesis, as I put it forward, is meant to stimulate active exploration of natural problem parameters in cognitive tasks. It is only when we know how the complexity of a function depends on its problem parameters that we can have a solid understanding of a function's complexity in practice.

Chapter 4. Techniques in Parameterized Complexity

This chapter presents a primer on parameterized complexity analysis. The purpose of this chapter—as well as the three following chapters—is to make the techniques for classical and parameterized complexity analysis accessible to cognitive psychologists interested in analyzing the complexity of their cognitive theories. This chapter gives an overview of basic techniques developed in the field of parameterized complexity. To facilitate understanding I illustrate each technique with one or more examples. For my examples I use Vertex Cover and related graph problems.[34] The techniques presented in this chapter, together with the technique of polynomial-time reduction (Section 3.2), are then used in Chapters 5, 6 and 7 to illustrate complexity analysis for existing cognitive theories.

4.1. Reduction Rules

If Π is an NP-hard problem, then we know that we cannot solve Π in polynomial-time (unless P = NP). Nevertheless, sometimes we can solve parts of an instance *i* for Π in polynomial time by applying so-called reduction rules. A *reduction rule* takes as input an instance *i* for Π and transforms it into an instance *i*' for Π such that *i* is a yes-instance for Π if and only if *i*' is a yes-instance for Π. In other words, a reduction rule instantiates a reduction as discussed in Section 3.1, but in this particular case it is a reduction where *i* and *i'* are instances for one and the same problem. Generally, the goal of a reduction rule is to reduce instance *i* to an instance *i'* such that *i'* smaller and/or easier to work with than *i*. Since the reduction maintains the equivalence between *i* and *i'* for Π (i.e., the answer is either "yes" for both *i* and *i'* or "no" for both *i* and *i'*), we can safely work with *i'* instead of *i*, even if our goal is to solve *i*. Namely, the reduction ensures that as soon as we solve *i'* for Π we have also solved *i* for Π.

[34] I do not claim originality of all examples in this chapter. Many of the examples using Vertex Cover are adopted from Balasubramanian, Fellows, and Raman (1998) and the monograph by Downey and Fellows (1999). Many examples using profit problems derive from my own work together with Ulrike Stege. My contribution here is to bring all these examples together in a coherent way, with the aim of making parameterized complexity techniques easily understandable for a non-expert.

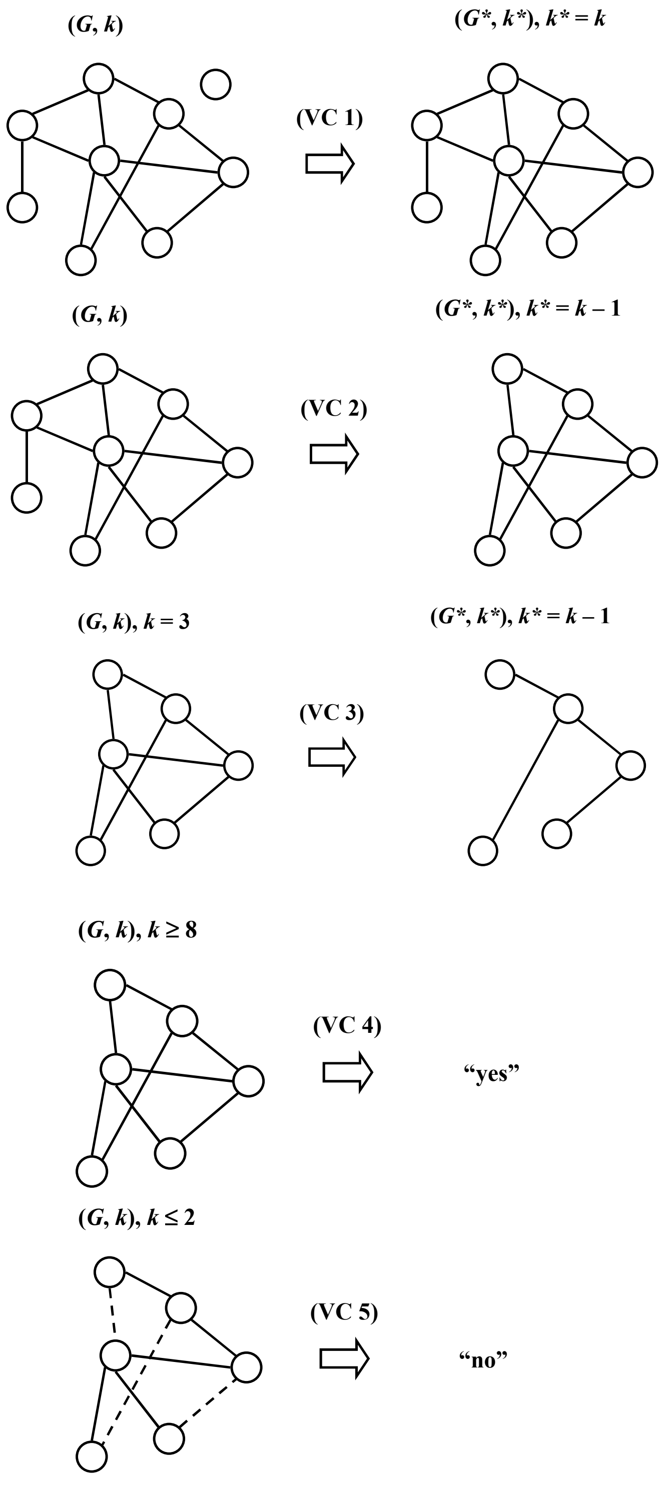


Figure 4.1. Illustration of reduction rules (VC 1) – (VC 5) for Vertex Cover.
Arrows indicate the application of a reduction rule. The instance before reduction rule application is denoted (G, k) and the instance after reduction rule application is denoted (G^*, k^*). Note that, in each case, (G^*, k^*) is defined such that it is a yes-instance for Vertex Cover if and only if (G, k) is a yes-instance for Vertex Cover. Application of (VC 1) causes singletons to be deleted from G. Application of (VC 2) causes pendant vertices to be deleted from G. Note that we set $k^* = k - 1$, because rule (VC 2) assumes that the neighbor of the pendant vertex is in the vertex cover. Application of (VC 3) causes vertices with degree larger than k to be deleted from G. Again we set $k^* = k - 1$ because rule (VC 3) assumes that the deleted vertex is in the vertex cover. Note that, in this example, (G^*, k^*) is a graph with pendant vertices; thus we can apply (VC 1) to (G^*, k^*). If the number of edges in G is smaller than k, then application of (VC 4) returns the answer "yes," because then we can cover each edge by including one of its endpoints in the vertex cover. If the number of independent edges in G exceeds k, then application of (VC 5) returns the answer "no," because for each independent edge we will need to include at least one vertex in the vertex cover. In the bottom figure, the dotted lines form a largest set of independent edges for G.

To illustrate the use of reduction rules, we again consider the problem Vertex Cover. Figure 4.1 illustrates the first five reduction rules.

To derive the first reduction rule, we observe that if an input graph G contains a singleton v (i.e., $\deg_G(v) = 0$), then we can delete v from G. Namely, v has no incident edges and thus v need not be considered for inclusion in the vertex cover for G. Hence we can apply the following reduction rule to G when solving Vertex Cover.

(VC 1) Singleton Rule: Let graph $G = (V, E)$ and positive integer k form an instance for Vertex Cover. If there exists a vertex $v \in V$ with $\deg_G(v) = 0$, then let $G^* = (V^*, E)$, with $V^* = V \backslash \{v\}$, and positive integer k form the new instance for Vertex Cover.

To prove that (VC 1) is a valid reduction rule as defined above, we need to show that (G, k) is a yes-instance for Vertex Cover if and only if (G^*, k^*) is a yes-instance for Vertex Cover. That is, the transformation described by the rule must be such that, if the answer for the new instance (G^*, k^*) is "yes" then we can conclude that the old instance (G, k) is "yes," and if the answer for the new instance (G^*, k^*) is "no" then we can conclude that the old instance (G, k) is "no."

Proof of (VC 1): We show that (G, k) is a yes-instance for Vertex Cover if and only if (G^*, k^*) is a yes-instance for Vertex Cover. Let $v \in V$ be a vertex in G without incident edges. ($\Rightarrow$) Let (G, k) be a yes-instance for Vertex Cover. Then G has a vertex cover $V' \subseteq V$ of size at most k. We distinguish two cases: (1) Let $v \notin V'$. Then V' is a vertex cover for G^* of size $k^* = k$. (2) Let $v \in V'$. Then, since v is a singleton, $V' \backslash \{v\}$ is also a vertex cover for G of size at most $k - 1$, and thus $V' \backslash \{v\}$ is also a vertex cover for G^* of size at most $k^* - 1 = k - 1$. Hence, in both case (1) and (2) we conclude that (G^*, k^*) is a yes-instance for Vertex Cover. ($\Leftarrow$) Let (G^*, k^*) be a yes-instance for Vertex Cover. Then G^* has a vertex cover $V' \subseteq V$ of size k^*. Since G contains the same edges as G^*, V' is also a vertex cover for G. We conclude that (G, k) is a yes-instance for Vertex Cover. ■

In the proof of (VC 1) above, we proved each direction ($\Rightarrow$) and ($\Leftarrow$) separately. Sometimes we may choose to prove both directions at once. For example, an alternative proof of (VC 1) could go as follows.

Alternative Proof of (VC 1): Let $V' \subseteq V$ be a *minimum* vertex cover for G and let $v \in V$ be a singleton. We make three observations. (1) We know $v \notin V'$ (namely, if we assume $v \in V'$, then we can have a smaller vertex cover $V' \backslash \{v\}$ for G, contradicting that $|V'|$ is minimum). (2) Since $G^* = (V \backslash \{v\}, E)$, we know V' is a minimum vertex cover for G^*. (3) In general, we know for any instance (G', k') that (G', k') is a yes-instance if and only if a minimum vertex cover for G' has size at most k'. From (1), (2) and (3) we conclude that (G, k) is a yes-instance if and only if (G^*, k^*) is a yes-instance. ■

We derive a second reduction rule. We observe that any instance (G, k) for Vertex Cover can be reduced such that G has no pendant vertices (i.e., vertices of exactly degree 1). Namely, a pendant vertex u, with neighbor v, can cover only edge (u, v), whereas its neighbor v covers the same edge *plus possibly more*. Hence we can always include v instead of u in the vertex cover for G.

(VC 2) Pendant Rule: Let graph $G = (V, E)$ and positive integer k form an instance for Vertex Cover. If there exists a vertex $u \in V$ with $\deg_G(u) = 1$, then let $G^* = (V^*, E^*)$, with $V^* = V \backslash \{u, v\}$, $E^* = E \backslash R_G(\{u, v\})$, and positive integer $k^* = k - 1$ form the new instance for Vertex Cover.

Proof: We need to show that (G, k) is a yes-instance for Vertex Cover if and only if (G^*, k^*) is a yes-instance for Vertex Cover. ($\Rightarrow$) Let (G, k) be a yes-instance for Vertex Cover. Then G has a vertex cover $V' \subseteq V$ of size k. First, we show that w.l.o.g. $v \in V'$. The proof is by contradiction. Assume that $v \notin V'$. Then its neighbor u must be in V', otherwise edge (u, v) is not covered and V' would not be a vertex cover for G. But then we can transform V' into $V'' = (V' \cup \{v\}) - \{u\}$ and conclude that V'' is a vertex cover for G of size k. Second, since w.l.o.g. $v \in V'$ and $E^* = E \backslash R_G(\{u, v\})$, we know that $V' \backslash \{v\}$ is a vertex cover of size $k - 1$ for G^*. We conclude that (G^*, k^*) is a yes-instance for Vertex Cover. ($\Leftarrow$) Let (G^*, k^*) be a yes-instance for Vertex Cover. Then G^* has a vertex cover $V' \subseteq V$ of size k^*. Since v covers every edge in $R_G(\{u, v\})$, and $E = E^* \cup R_G(\{u, v\})$, we know that $V' \cup \{v\}$ is a vertex cover for G of size $k^* + 1$. We conclude that (G, k) is a yes-instance for Vertex Cover. ■

If we apply reduction rules (VC 1) and (VC 2) to an instance (G, k) until no longer possible, then we will have removed all degree-0 and degree-1 vertices from the input graph.

We next show that we can also remove vertices of high degree. More specifically, if G contains a vertex v of degree at least k +1, then any vertex cover for G of size at most k will have to include v. Namely, to cover the edges incident to v we need to include either v in the vertex cover or all of its neighbors. Since v has more than k neighbors, V' is a vertex cover for G of size at most k only if $v \in V'$.

(VC 3) Degree-(k +1) Rule: Let graph $G = (V, E)$ and positive integer k form an instance for Vertex Cover. If there exists a vertex $v \in V$ with $\deg_G(v) > k$, then let $G^* = (V^*, E^*)$, with $V^* = V \setminus \{v\}$, $E^* = E \setminus \mathrm{R}_G(\{v\})$, and positive integer $k^* = k - 1$ form the new instance for Vertex Cover.

Proof: We need to show that (G, k) is a yes-instance for Vertex Cover if and only if (G^*, k^*) is a yes-instance for Vertex Cover. ($\Rightarrow$) Let (G, k) be a yes-instance for Vertex Cover. Then G has a vertex cover $V' \subseteq V$ of size k. We show that $v \in V'$. The proof is by contradiction. Assume that $v \notin V'$. Then all neighbors of v are in V'. Since $|\mathrm{N}_G(v)| = \deg_G(v) > k$ we can conclude $|V'| > k$, contradicting the fact that V' is of size at most k. Since, $v \in V'$ and $E^* = E \setminus \mathrm{R}_G(\{v\})$, we know that $V' \setminus \{v\}$ is a vertex cover of size $k - 1$ for G^*. We conclude that (G^*, k^*) is a yes-instance for Vertex Cover. ($\Leftarrow$) Let (G^*, k^*) be a yes-instance for Vertex Cover. Then G^* has a vertex cover $V' \subseteq V$ of size k^*. Since v covers every edge in $\mathrm{R}_G(\{v\})$, and $E = E^* \cup \mathrm{R}_G(\{v\})$, we know that $V' \cup \{v\}$ is a vertex cover for G of size $k^* + 1$. We conclude that (G, k) is a yes-instance for Vertex Cover. ■

For certain problems it is possible to formulate a reduction rule that tests whether the instance i has a property that allows us to directly decide i for Π. For example, if a graph G has less than k edges then we know right away that G has a vertex cover of size at most k. Namely, for each edge we can just include any one of its incident vertices in the vertex cover.

(VC 4) Less Than k Edges Rule: Let graph $G = (V, E)$ and positive integer k form an instance for Vertex Cover. If $|E| \leq k$ then output "yes."

Proof: Let $G = (V, E)$ be a graph with $|E| \leq k$. Then we can cover every edge $(u, v) \in E$ with the following procedure: For each $(u, v) \in E$ include v into the vertex cover $V' \subseteq V$. Since we have at most k edges, this procedure will lead us to include at most k vertices in the vertex cover. Hence, (G, k) is a yes-instance for Vertex Cover. ■

The rule (VC 4) allows us to recognize some yes-instances for Vertex Cover. There also exist rules that allow us to recognize some no-instances for Vertex Cover. For example, let G and k form an instance for Vertex Cover, and let $E' \subseteq E$ be a set of *independent* edges in G (i.e., no two edges in E' has a vertex in common). If $|E'| > k$ then we know that G and k form a no-instance for Vertex Cover. Namely, to cover an edge (u, v) at least one of u and v must be in the vertex cover. Since the edges in E' are independent, a vertex that covers an edge in E' does not cover any other edge in E'. Hence, we need at least $|E'| > k$ vertices to cover all edges in G.

(VC 5) More Than *k* Independent Edges Rule: Let graph $G = (V, E)$ and positive integer k form an instance for Vertex Cover. If G contains a set of independent edges $E' \subseteq E$ such that $|E'| > k$, then output "no."

Proof: Let $G = (V, E)$ be a graph, and let $E' \subseteq E$ be a set of independent edges. Further, let $|E'| > k$ and let (u, v) be any edge in E'. For a vertex set $V' \subseteq V$ to cover $(u, v) \in E'$, we must have $u \in V'$ or $v \in V'$. Since all edges in E' are independent we know that for any edge $(x, y) \in E'$, with $(x, y) \neq (u, v)$, that $x \neq u$, $x \neq v$, $y \neq u$, and $y \neq v$ (otherwise the two edges are not independent). Hence we need at least as $|E'| > k$ vertices in V' to cover all edges in E', and thus we need at least as $|E'| > k$ vertices in V' to cover all edges in E. We conclude that (G, k) is a no-instance for Vertex Cover. ■

All these reduction rules run in polynomial-time. The time to execute rules (VC 1) and (VC 2) is independent of the size of the input, and thus these rules run in time $O(1)$. The rule (VC 3) requires the removal of a vertex and all its incident edges from G; since a vertex has at most $|V| - 1$ incident edges, this rule can be executed in time $O(|V|)$. Rule (VC 4) involves counting the number of edges in the input graph, which runs in time $O(|V|^2)$. Finally, rule (VC 5) involves counting the number of independent edges in a graph. The problem of finding the maximum number of independent edges in a graph is equivalent to the problem Maximum Matching (see Appendix B for problem definition).

It is known that Maximum Matching can be solved in time $O(|V|^2)$ (e.g. Gross & Yellen, 1999), and thus rule (VC 5) also runs in polynomial-time.[35]

Whenever we have a set of reduction rules for a given problem we often say that a problem is *reduced* if none of the reduction rules apply. For example, we may say that instance (G, k) for Vertex Cover is reduced if and only if (VC 1) – (VC 5) do not apply to (G, k).

4.2. Reduction to Problem Kernel

Sometimes a set of reduction rules can lead to *kernelization* of the input. In such cases we know that, if application of the reduction rules is no longer possible, then the size of the input i is bounded by some function depending only on the parameter κ. The kernelized input i is then called a *problem kernel*. More formally, if there exists a function $f(\kappa)$, depending only on κ, such that $|i| \leq f(\kappa)$ we say that i is *kernelized*. We call i the *problem kernel*, and we call $|i| \leq f(\kappa)$ the *problem kernel size*. A set of reduction rules that leads to a problem kernel is called a *reduction to problem kernel*.

Once a kernelization result is known for a problem κ-Π, we conclude that κ-$\Pi \in$ FPT.[36] Namely, we know that there exists some function $g(|i|)$, such that $O(g(|i|))$ captures the running time for Π. But then, given the problem kernel $|i| \leq f(\kappa)$, we know that we can solve κ-Π in time $O(g(f(\kappa)) + h(|i|, \kappa))$, where function $h(|i|, \kappa)$ denotes the time required for the reduction. If $h(|i|, \kappa)$ is polynomial- or fpt-time we can rewrite $O(g(f(\kappa)) + h(|i|, \kappa))$ as $O(g'(\kappa) + h'(|i|))$, which is fpt-time.

To illustrate this technique, we add to the set of reduction rules (VC 1)–(VC 5) the kernelization rule (VC 6). It is important that before applying rule (VC 6) we first reduce the input (specifically, we have to make sure that the rules (VC 1) and (VC 3) do not apply). Then, using the fact that reduced graphs have no degree-0 and no degree-(k

[35] All reductions considered here happen to be polynomial-time reduction rules. I remark that it is also possible to use reduction rules that run in fpt-time when building fpt-algorithms. In the discussion of bounded search tree, in Section 4.3, we will allow for either polynomial-time or fpt-time reduction rules. A reduction that runs in fpt-time will be illustrated in Section 4.6.

[36] In fact, an even stronger relationship holds: A problem is in FPT if and only if it is polynomial-time kernelizable (Downey et al., 1999a).

+1) vertices, rule (VC 6) concludes that a reduced instance (G, k) such that G has more than $k(k + 1)$ vertices is a no-instance for Vertex Cover. See Figure 4.2 for an illustration of (VC 6).

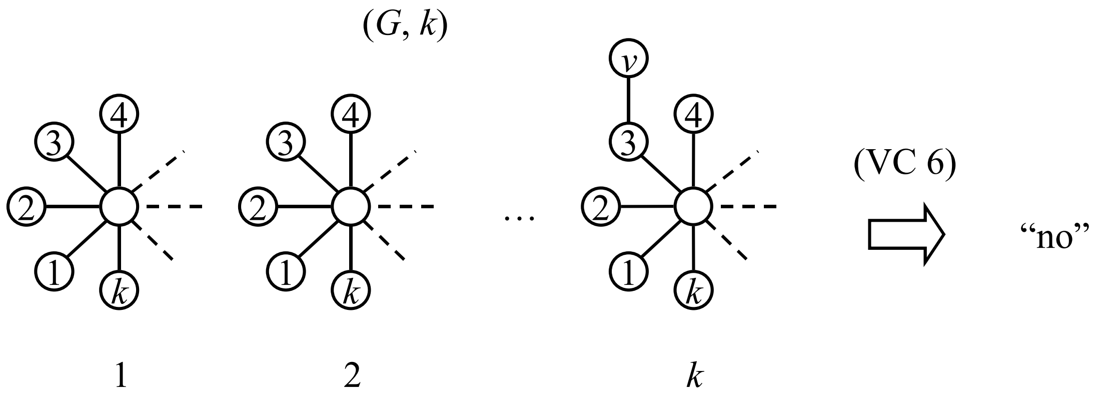


Figure 4.2. Illustration of rule (VC 6) and the intuition behind its proof.
The figure depicts an instance (G, k) for Vertex Cover. The graph G consists of k components: the first $k-1$ components are stars and the k^{th} component is a star on which a pendant vertex v is appended. Further, the root of each component is of degree k. Note that the most economical way of covering every edge in G is by including the root of each star, plus v or its neighbor, in the vertex cover. This means that the smallest vertex cover for G is of size $k + 1$ and thus the answer is "no" for (G, k). To illustrate the intuition behind the proof of (VC 6) we make some further observations: Note that G has exactly $k(k + 1) + 1$ vertices. Consider the graph G' that is the same as G except that vertex v has been deleted. Note that G' has $k(k + 1)$ vertices, and the smallest vertex cover for G' is of size k. If you try to change G', by changing only the edge set, and without creating singletons (because then we can apply (VC 1)) and without creating vertices of degree larger than k (because then we can apply (VC 3)), you will see that it is not possible to create a graph on the same set of vertices that has a vertex cover of size $k-1$. From this we conclude that any graph larger than G' for which (VC 1) and (VC 3) do not apply, like G, does not have a vertex cover of size k.

(VC 6) More Than *k*(*k* + 1) Vertices Rule:[37] Let graph $G = (V, E)$ and positive integer k form a reduced instance for Vertex Cover (i.e., rules (VC 1) – (VC 5) do not apply to (G, k)). If G has more than $k(k + 1)$ vertices, then output "no."

Proof: Let G be a reduced graph with more than $k(k + 1)$ vertices. Further, because G is reduced we know it does not contain vertices with degree larger than k, nor

[37] This kernelization rule also appears in Downey and Fellows (1999), and is due to Sam Buss.

any degree-0 vertices. We show that we need more than k vertices in the vertex cover to cover all the edges.

The proof is by contradiction. Let $V' \subseteq V$ be vertex cover for G of size k. Because G is reduced its vertex degree is bounded by k, and thus the maximum number of edges covered by any one vertex $v \in V'$ is k. Now consider the combined neighborhood of vertices in V', denoted $N_G(V')$. Since each vertex $v \in V'$ can cover at most k edges, and V' contains at most k vertices, we know that $|N_G(V')| \leq k^2$ and thus $|V' \cup N_G(V')| \leq k^2 + k = k(k + 1)$. But G has more than $k(k + 1)$ vertices. This means there must exist at least one vertex u in G such that $u \notin V' \cup N_G(V')$. Since G is reduced we know that this vertex has at least one incident edge (viz., (VC 1) does not apply), and since $u \notin V' \cup N_G(V')$ we know that this edge is not covered by V'. We conclude that V' is not a vertex cover for G. ■

We say that an instance (G, k) for k-Vertex Cover is *kernelized* if it is reduced and (VC 6) does not apply. We now observe that in a kernelized instance (G, k) for k-Vertex Cover, with $G = (V, E)$, the size of V is bounded by a function of k. To be exact, we have a problem kernel $|V| = n < k(k + 1)$. Recall that we can solve Vertex Cover in time $O(2^n)$, simply by trying out all 2^n possible subsets of vertices (e.g., using the Exhaustive Vertex Cover algorithm described on page 24). This means that we can solve a reduced instance of Vertex Cover in time $O(2^{k(k+1)})$. Adding the time required to reduce and kernelize an instance we conclude that we can solve any instance for Vertex Cover in time $O(2^{k(k+1)} + n^2)$. This establishes that k-Vertex Cover is in FPT.

4.3. Bounded Search Tree

Reduction to problem kernel is one technique for showing that a problem is in FPT. Another technique is the bounded search tree technique. The goal of this technique is to construct a search tree T, such that for every instance i for κ-Π the following three conditions are met: (1) the search tree returns a solution for κ-Π (i.e., one of its leaves is labeled by a yes-instance) if and only if i is a yes-instance for κ-Π, (2) the search tree is bounded in size by some function $f(\kappa)$, and (3) each node in the search tree can be created in fpt-time.

Here, requirement (1) is to ensure the *correctness* of the algorithm; i.e., we want the algorithm to solve the problem Π. Requirements (2) and (3) are to ensure that the search terminates in fpt-time $O(f(\kappa)|i|^{\alpha})$. To see that requirements (2) and (3) together ensure an fpt running time consider the following: Let the size of the search tree T be $O(f_1(\kappa))$ and let the time to create a node in T be time $O(f_2(\kappa)\ |i|^{\alpha})$, then the total running time of the search algorithm is given by $O(f_1(\kappa) f_2(\kappa)\ |i|^{\alpha})) = O(f(\kappa)\ |i|^{\alpha})$, which is fpt-time with respect to parameter κ. I discuss and illustrate points (1), (2) and (3) in turn.

To meet requirement (1), we create the search tree by recursively calling one or more branching rules. Like a reduction rule, a branching rule takes as input an instance i for a problem Π, but unlike a reduction rule, a branching rule outputs multiple instances $i_1, i_2, \ldots, i_b$ for Π. To ensure that the search tree returns a solution for κ-Π if and only if (I, κ) is a yes-instance for κ-Π, the instances $i_1, i_2, \ldots, i_b$ are constructed such that i is a yes-instance for Π if and only if i_1 or i_2 or… or i_b is a yes-instance for Π. To illustrate we reconsider the Exhaustive Vertex Cover algorithm discussed in Section 2.4.2 (page 24).

The Exhaustive Vertex Cover algorithm can be seen as building the search tree T in Figure 2.3 (page 26). The root of T is labeled by the original input instance (G, k).[38] For each node s in T the procedure Branch in the Exhaustive Vertex Cover algorithm causes the creation of two children, s_1 and s_2, of s. Here s_1 and s_2 are labeled by (G_1, k_1) and (G_2, k_2) such that (G, k) is a yes-instance for Vertex Cover if and only if (G_1, k_1) or (G_2, k_2) is a yes-instance for Vertex Cover. The procedure Branch accomplishes this by applying the rule "If V' is a vertex cover for a graph G, and v is a vertex in G, then either v is in V' or v is not in V'." This branching rule is formally defined by (VC 7) below (see Figure 4.3 for an illustration).

[38] In Figure 2.3 on page 26, at each node of the search tree we additionally kept track of the set V' that was to be build into a vertex cover. From now on I will no longer explicate that V' is being build during the construction of the search tree. For all algorithms discussed in the text—unless otherwise noted—the reader may just assume that the algorithm builds a constructive solution for every yes-instance of the problem that it solves.

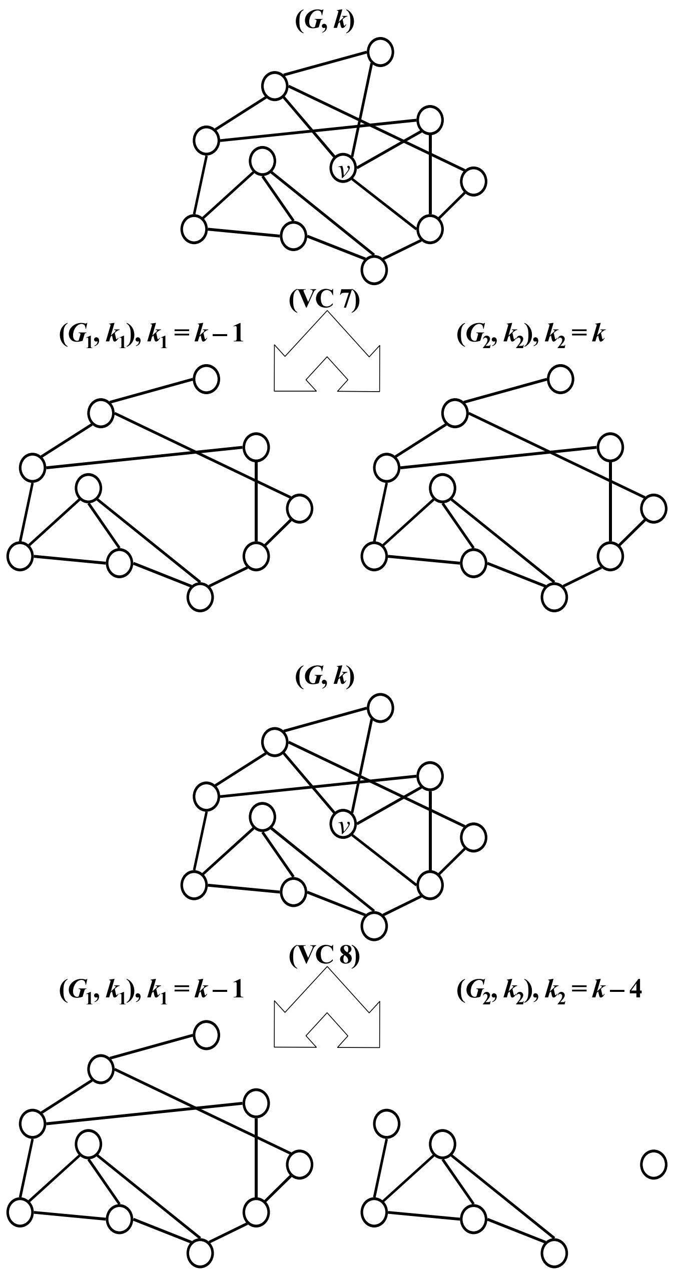

(G, k)
v
(VC 7)
$(G_1, k_1), k_1 = k - 1$
$(G_2, k_2), k_2 = k$
(G, k)
v
(VC 8)
$(G_1, k_1), k_1 = k - 1$
$(G_2, k_2), k_2 = k - 4$

Figure 4.3. Illustration of branching rules (VC 7) and (VC 8). The instance before branching rule application is denoted (G, k) and the two instances obtained after branching rule application are denoted (G_1, k_1) and (G_2, k_2). Note that both branching rules define the new instances (G_1, k_1) and (G_2, k_2) such that (G_1, k_1) or (G_2, k_2) is a yes-instance for Vertex Cover if and only if (G, k) is a yes-instance for Vertex Cover. This way each rule ensures that the branching algorithm returns the answer “yes” (or “no”) if and only if the answer for the original input to the algorithm is “yes” (“no”). The rule (VC 7) considers for a vertex v in G, the possibility that v is in the vertex cover and the possibility that v is not in the vertex cover. If we assume that v is in the vertex cover, then we can delete v and its incident edges from G, leading to G_1, and we set $k_1 = k - 1$. If we assume that v is not in the vertex cover, then we also delete v and its incident edges from G, leading to G_2, but we set $k_2 = k$. Note that, in this example, (VC 2) can be applied to both (G_1, k_1) and (G_2, k_2). The rule (VC 8), like (VC 7), considers for a vertex v in G, the possibility that v is in the vertex cover and the possibility that v is not in the vertex cover. The rule (VC 8), unlike (VC 7), uses the fact that, if v is not in the vertex cover, then all of its neighbors are in the vertex cover (otherwise the edges incident to v would not be covered). If we assume v is in the vertex cover, then we delete v and its incident edges from G, leading to G_1, and we set $k_1 = k - 1$. If we assume v is not in the vertex cover, then the neighbors of v are in the vertex cover. Then we delete v and its incident edges from G, and additionally we delete its neighbors and their incident edges, leading to G_2. Further, we set $k_2 = k - |\mathrm{N}_G(v)|$. In this example, v has 4 neighbors and thus $|\mathrm{N}_G(v)| = 4$. Also note that, in this example, (VC 2) can be applied to (G_1, k_1), and both (VC 1) and (VC 2) can be applied to (G_2, k_2).

(VC 7) The Vertex-In-or-Out Branching Rule: Let node s in the search tree be labeled by instance (G, k), $G = (V, E)$, for Vertex Cover and let $v \in V$. Then we create two children of s in the search tree, called s_1 and s_2, and label s_1 by (G_1, k_1) and label s_2 by (G_2, k_2). Here $G_1 = (V_1, E_1)$ with $V_1 = V \setminus \{v\}$, $E_1 = E \setminus \mathrm{R}_G(\{v\})$, $k_1 = k - 1$ and $G_2 = (V_2, E_2)$ with $V_2 = V \setminus \{v\}$, $E_2 = E \setminus \mathrm{R}_G(\{v\})$, $k_2 = k$.

Proof: We need to show that (G, k) is a yes-instance for Vertex Cover if and only if (G_1, k_1) or (G_2, k_2) is a yes-instance for Vertex Cover. ($\Rightarrow$) Let (G, k) be a yes-instance for Vertex Cover. Then there exists a vertex cover $V' \subseteq V$ for G with $|V'| \leq k$. We distinguish two cases: (1) Let $v \in V'$. Since $E_1 = E \setminus \mathrm{R}_G(\{v\})$ we know that $V' \setminus \{v\}$ is a vertex cover for G_1 of size $k - 1$. We conclude (G_1, k_1) is a yes-instance for Vertex Cover. (2) Let $v \notin V'$. Then V' is a vertex cover for G_2 of size k. We conclude (G_2, k_2) is a yes-instance for Vertex Cover. ($\Leftarrow$) Let (G_1, k_1) or (G_2, k_2) be a yes-instance for Vertex Cover. We distinguish two cases: (1) Let (G_1, k_1) be a yes-instance for Vertex Cover. Then there exists a vertex cover V_1' for G_1, such that $|V_1'| \leq k_1$. Since v covers every edge in $\mathrm{R}_G(\{v\})$, and $E = E_1 \cup \mathrm{R}_G(\{v\})$, we know that $V' = V_1' \cup \{v\}$ is a vertex cover for G of size at most $k_1 + 1$. We conclude (G, k) is a yes-instance for Vertex Cover. (2) Let (G_2, k_2) be a yes-instance, and (G_1, k_1) be a no-instance, for Vertex Cover. Then there exists a vertex cover V_2' for G_2, such that $|V_2'| \leq k_2$. Then V_2' is a vertex cover for G of size k_2 (otherwise, (G_1, k_1) would be a yes-instance for G). We conclude (G, k) is a yes-instance for Vertex Cover. ■

The rule (VC 7) ensures that a leaf of T in Figure 2.3 (page 26) is labeled by a yes-instance for Vertex Cover if and only if the instance that labels the root of T is a yes-instance for Vertex Cover. This is how we ensure that the Exhaustive Vertex Cover algorithm always returns "yes" if the original input is a yes-instance, and "no" if the original input is a no-instance.

Is the search tree size created by (VC 7) bounded by a function of the parameter k? No, it is not. Even though fan(T) is bounded by the function $g(k) = 2$, the depth of the search tree is not bounded any function $h(k)$. Namely, the depth of the path from the root of T to the rightmost leaf can be as long as n. For this reason the search tree obtained by the application of rule (VC 7) does not meet requirement (2).

In general, for a parameter κ, requirement (2) can be met if there exist functions $g(\kappa)$ and $h(\kappa)$, such that $\text{fan}(T) \leq g(\kappa)$ and $\text{depth}(T) \leq h(\kappa)$. Namely, as we have seen in Section 2.4.2, for any search tree T, $\text{size}(T) \leq 2\text{fan}(T)^{\text{depth}(T)} - 1$, which is $O(\text{fan}(T)^{\text{depth}(T)})$. And thus, if we have $\text{fan}(T) \leq g(\kappa)$ and $\text{depth}(T) \leq h(\kappa)$, then we conclude that $\text{size}(T)$ is $O(g(\kappa)^{h(\kappa)})$. To illustrate, I now present a new branching rule, (VC 8), to replace (VC 7). I will show that (VC 8) can be used to construct a search tree that meets requirement (2) for parameter k. Rule (VC 8) uses the observation that, to cover all edges incident to a vertex v in a graph G, if v itself is not in the vertex cover then all of its neighbors must be in the vertex cover. Further, the rule (VC 8) is applied only to vertices of minimum degree 1 to ensure that k gets reduced by at least 1 for each branch (see Figure 4.3).

(VC 8) The Vertex-or-All-Its-Neighbors Branching Rule: Let node s in the search tree be labeled by instance (G, k), $G = (V, E)$, for Vertex Cover and let $v \in V$ with $\text{deg}_G(v) \geq 1$. Then we create two children of s in the search tree, called s_1 and s_2. We label s_1 by (G_1, k_1), with $G_1 = (V_1, E_1)$, $V_1 = V \backslash \{v\}$, $E_1 = E \backslash \text{R}_G(\{v\})$, $k_1 = k - 1$, and we label s_2 by (G_2, k_2), with $G_2 = (V_2, E_2)$ with $V_2 = V \backslash \text{N}_G(v)$, $E_2 = E \backslash \text{R}_G(\text{N}_G(v))$, $k_2 = k - |\text{N}_G(v)|$.

Proof: We need to show (G, k) is a yes-instance for Vertex Cover if and only if and only if (G_1, k_1) or (G_2, k_2) is a yes-instance for Vertex Cover. ($\Rightarrow$) Let (G, k) be a yes-instance for Vertex Cover. Then there exists a vertex cover $V' \subseteq V$ for G with $|V'| \leq k$. We distinguish two cases: (1) Let $v \in V'$. Since $E_1 = E \backslash \text{R}_G(\{v\})$ we know that $V' \backslash \{v\}$ is a vertex cover for G_1 of size $k - 1$. We conclude (G_1, k_1) is a yes-instance for Vertex Cover. (2) Let $v \notin V'$. Then $\text{N}_G(v) \subseteq V'$ and thus $V' \backslash \text{N}_G(v)$ is a vertex cover for G_2 of size $k - |\text{N}_G(v)|$ (Note: Since $\text{deg}_G(v)$, $|\text{N}_G(v)| \leq k - 1$). We conclude (G_2, k_2) is a yes-instance for Vertex Cover. ($\Leftarrow$) Let (G_1, k_1) or (G_2, k_2) be a yes-instance for Vertex Cover. We distinguish two cases: (1) Let (G_1, k_1) be a yes-instance for Vertex Cover. Then there exists a vertex cover V_1' for G_1, such that $|V_1'| \leq k_1$. Since v covers every edge in $\text{R}_G(\{v\})$, and $E = E_1 \cup \text{R}_G(\{v\})$, we know that $V' = V_1' \cup \{v\}$ is a vertex cover for G of size at most $k_1 + 1$. We conclude (G, k) is a yes-instance for Vertex Cover. (2) Let (G_2, k_2) be a yes-instance for Vertex Cover. Then there exists a vertex cover V_2' for G_2, such that $|V_2'| \leq k_2$. Since $\text{N}_G(v)$ covers every edge in $\text{R}_G(\text{N}_G(v))$, and $E = E_2 \cup \text{R}_G(\text{N}_G(v))$, we know that

$V' = V_1' \cup \mathrm{N}_G(v)$ is a vertex cover for G of size at most $k_1 + |\mathrm{N}_G(v)|$. We conclude (G, k) is a yes-instance for Vertex Cover. ■

We can use the branching rule (VC 8) to create a bounded search tree algorithm for k-Vertex Cover as follows: The algorithm takes as input an instance (G, k) and recursively applies (VC 8) to (G, k) until either an instance (G', k') is encountered with $k' \leq 0$ (in which case the algorithm returns the answer "yes") or (VC 8) cannot be applied anymore (this happens if the graph is empty or contains singletons only). If the algorithm halts without returning the answer "yes" then we conclude that (G, k) is a no-instance for Vertex Cover (viz., we have shown in the proof above that the search tree will return the answer "yes" if and only if (G, k) is a yes-instance for Vertex Cover). This meets requirement (1) for a bounded search tree.

Now, to see that the resulting search tree also meets requirement (2), first observe that each application of the rule (VC 8) at node s leads to the creation of at most two children s_1 and s_2. Hence, $\mathrm{fan}(T) \leq g(k) = 2$. Further, because we only apply (VC 8) to a vertex v if $\deg_G(v) \geq 1$, we have $|\mathrm{N}_G(v)| \geq 1$, and thus we know that for each child s_i, $i = 1, 2$, $k_i < k$. Since we terminate our search as soon as we encounter a node s' labeled by (G', k') with $k' = 0$, we conclude that $\mathrm{depth}(T) \leq h(k) = k$. Hence the size of the search tree is bounded as follows: $\mathrm{size}(T) \leq f(k) = 2\mathrm{fan}(T)^{\mathrm{depth}(T)} - 1 = 2g(k)^{h(k)} - 1 = 2^{k+1} - 1$, which is $O(2^k)$.[39]

To ensure that a search tree algorithm runs in fpt-time for parameter κ, we not only need to bound the size of the search tree T by a function of κ, but we also need to make sure that each node in T can be created in fpt-time. This is requirement (3). To illustrate we again consider the search tree created by branching rule (VC 8): To apply the rule we need to find a vertex v in G of minimum degree 1. We can do so in time $O(|V|)$ (Note: Instead of searching for such a vertex, we can first apply (VC 1) until no longer possible, in time $O(|V|)$, and then pick any arbitrary vertex in G to branch on, in time $O(1)$). Further, at each node we spend at most time $O(|V|^2)$ to delete one or more vertices and their incident edges, and to update k. We conclude that the branching

[39] Note that we can shrink the size of the search tree by requiring that the instance is reduced using (VC 2) before applying (VC 8). Then we ensure that $|\mathrm{N}_G(v)| \geq 2$, leading to a tree size that is $O(1.618^k)$.

algorithm for Vertex Cover that uses (VC 8)—possibly in combination with (VC 1)—is an fpt-algorithm for *k*-Vertex Cover and runs in time $O(2^k |V|^2)$.

4.4. Alternative Parameterizations

In the illustrations so far, we have only considered the parameter *k*. In Vertex Cover the integer *k* denotes the bound on the size of the vertex cover in *k*-Vertex Cover; in Dominating Set the integer *k* denotes the bound on the size of the dominating set. Because this parameter is explicitly stated as a part of the input we call it an *explicit parameter*. The parameter *k* is sometimes also referred to as the *natural parameter* for Vertex Cover and Dominating Set, and *k*-Vertex Cover and *k*-Dominating Set are then called the *natural parameterizations* of Vertex Cover and Dominating Set (see e.g. Downey & Fellows, 1999; Niedermeier, 2002; Stege & van Rooij, 2003). Note, however, that many different parameterizations are possible for these problems. In this section I discuss such alternative parameterizations.

4.4.1. Implicit Parameters

A problem Π may have many implicit parameters. An *implicit parameter* for a problem Π is an aspect of an instance *i* for Π that is not explicitly stated as part of the problem's input. Consider, for example, Vertex Cover and Dominating Set. Both take as input a graph $G = (V, E)$. Graphs have many implicit parameters: e.g., the maximum vertex degree in *G* (Δ), the minimum vertex degree in *G* (δ), the maximum number of independent edges in *G*, the number of cycles in *G*, the size of the smallest vertex cover for *G*, and so on. Note that a graph $G = (V, E)$ also has the number of vertices $|V| = n$ and the number of edges $|E| = m$ as implicit parameters.[40] For each of these parameters we may ask whether a graph problem is in FPT for that parameter. For example, we may ask: Is Δ-Vertex Cover in FPT? The following lemma shows that the answer is most likely "no."

Lemma 4.1. Δ-Vertex Cover $\notin$ FPT (unless P = NP)

[40] I will use Arabic letters to denote explicit parameters and Greek letters to denote implicit parameters, unless convention prescribes otherwise (e.g., *n* and *m* are the conventional letters that refer to $|V|$ and $|E|$).

Proof: It is known that Vertex Cover is NP-hard even for graphs with no vertex degree exceeding 4 (Garey & Johnson, 1979). Using this fact, we prove the claim by contradiction. Assume that Δ-Vertex Cover $\in$ FPT and P $\neq$ NP. Then there exists an fpt-algorithm solving Δ-Vertex Cover in time $O(f(\Delta)n^{\alpha})$. Therefore for $\Delta = 4$ we can solve Vertex Cover in time $O(f(4)n^{\alpha}) = O(n^{\alpha})$, meaning that Vertex Cover on graphs of maximum degree 4 is in P. But then P = NP. ■

Since also Dominating Set is know to be NP-hard for graphs of maximum vertex degree 4, with the same argument, we can conclude that Δ-Dominating Set $\notin$ FPT (unless P = NP). Analogously, since for all graphs $\delta \geq 0$, we can also argue that δ-Vertex Cover $\notin$ FPT and δ-Vertex Cover $\notin$ FPT (unless P = NP).

Note that the parameterized problems n-Vertex Cover, n-Dominating Set, m-Vertex Cover and m-Dominating Set, are all trivially in FPT. Namely, we can solve Vertex Cover and Dominating Set by an exhaustive search on all possible 2^n subsets on n vertices in time $O(2^n)$, which is fpt-time for parameter n. Hence, n-Vertex Cover $\in$ FPT and n-Dominating Set $\in$ FPT. Further, since $n < m$, we have $O(2^n) < O(2^m)$, which is also fpt-time for m-Vertex Cover and m-Dominating Set.

4.4.2. Relational Parameterization[41]

The implicit parameters for Vertex Cover and Dominating Set discussed in Section 4.4.1 are all properties of the graph G, and are independent of the integer k. There also exist implicit parameters for Vertex Cover and Dominating Set that involve a relationship between an implicit parameter and the explicit parameter k. We call such parameters *relational parameters*. Here, I discuss two types of relational parameterizations that appear in the literature. The first is called *dual parameterization* and the second *profit parameterization*.

[41] I introduce the notion of 'relational parameterization.' To my knowledge, the notion has not yet been so defined in the parameterized complexity literature. My definition is motivated by the observation that dual parameterizations, (some) profit parameterizations, and parameterizations discussed in Chapter 5 (page 115) and Chapter 6 (page 129), all have in common that the defined parameter is a function of both the input graph and the positive integer. It seems natural, and useful, to group these types of parameterization under one heading.

Dual Parameterization: The parameter in a dual parameterization is a function of k and n. I first define the *dual* of a problem Π with input graph $G = (V, E)$ and positive integer k. If Π asks "Does G have property X that has property Y that depends on k?" then the dual Π' asks "Does G have property X that has property Y that depends on $k' = |V| - k$?" (cf. Khot & Raman, 2000; Downey & Fellows, 1999; Niedermeier, 2002; Stege & van Rooij, 2003). The natural parameterization of Π', denoted k'-Π', is called the *dual parameterization* of Π, and k'-Π' is then called the *parametric dual* of k-Π.

For example, Vertex Cover asks "Does G have a vertex cover that contains at most k vertices?." The dual of Vertex Cover asks "Does G have a vertex cover that contains at most $k' = |V| - k$ vertices?" The dual of Vertex Cover is better known under the name Independent Set. A set of vertices $V' \subseteq V$ is called an *independent set* for a graph $G = (V, E)$ if there does not exists any edge $(u, v) \in E$ with both $u, v \in V'$.

Independent Set

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Does there exist an independent set V' for G with $|V'| \geq k$?

To prevent confusion let k_V denote the positive integer in the input of Vertex Cover, and let k_I denote the positive integer in the input of Independent Set. Now observe that a graph G has a vertex cover of size at most k_V if and only if G has an independent set of size at least $k_I = |V| - k_V$. This means that the natural parameterization of Independent Set, denoted k_I-Independent Set, is equivalent to the dual parameterization of Vertex Cover, denoted $(|V| - k_V)$-Vertex Cover or k_I-Vertex Cover.

We also consider the dual of Dominating Set, called Non-Blocker. A vertex set V' is called a *non-blocking set* if for every vertex $v \in V'$ there exists at least one neighbor u of v, such that $u \notin V'$.

Non-Blocker

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Does there exist a non-blocking set V' for G with $|V'| \geq k$?

Let k_D denote the positive integer in the input of Dominating Set, and let k_N denote the positive integer in the input of Non-Blocker. Observe that a graph G has a dominating set of size at most k_D if and only if G has a non-blocking set of size at least $k_N = |V| - k_D$.

Hence, the natural parameterization of Non-Blocker, denoted k_N-Non-Blocker, is equivalent to the dual parameterization of Dominating Set, denoted $(|V| - k_N)$-Dominating Set or k_D-Dominating Set.

Note that a parameterized problem κ-Π and its parametric dual κ'-Π' need not be in the same complexity class. For example, k-Vertex Cover is in FPT, but its parametric dual $(|V| - k)$-Vertex Cover (i.e., k-Independent Set) is W[1]-hard (Downey & Fellows, 1999). Also, k-Dominating Set is W[1]-hard, but its parametric dual $(|V| - k)$-Dominating Set (i.e., k-Non-Blocker) is in FPT.[42]

Profit Parameterization: Above we have seen that the natural parameterization of the dual of a graph problem Π instantiates a dual parameterization of Π. I will now explain how the natural parameterization of a *profit relaxation* of a graph problem Π instantiates a profit parameterization of Π. Below I will sketch the intuition behind profit relaxation using examples. For a more formal treatment of profit relaxation the reader is referred to Stege and van Rooij (2003).

Assume we wish to relax the problem Vertex Cover as follows: We no longer require the set V' to cover every edge in G; instead we consider the extent to which V' is "close" to being a vertex cover as *gain* (say, the more edges V' covers the better). Further, we consider the size of the set V' as *loss* (in other words, we want to use few vertices to cover many edges). The "profit" associated with a vertex set V' is then given by the function *profit* = *gain* − *loss*. With this relaxation of Vertex Cover we obtain a problem called Profit Cover (Stege, van Rooij, Hertel & Hertel, 2002).

Profit Cover

Input: A graph $G = (V, E)$ and a positive integer p.

Question: Does there exist a vertex set $V' \subseteq V$ such that $\text{profit}_{PC,G}(V') \geq p$? Here $\text{profit}_{PC,G}(V') = |R_G(V')| - |V'|$.

Note that Profit Cover is not just a relaxation of Vertex Cover—it has a special property. Its natural parameterization, p-Profit Cover, is a parameterization for Vertex Cover. Namely, a graph G has a vertex cover of size at most k if and only if there exists a vertex

[42] This is an unpublished result due to Fellows and MacCartin in 1999 (see also Prieto & Sloper, forthcoming; Stege & van Rooij, 2003).

set *V'* with $\text{profit}_{\text{PC},G}(V') \geq p = |E| - k$ (see Stege et al., 2002, for a proof). In other words, *p*-Profit Cover is a relational parameterization for Vertex Cover, with the parameter being the function $p = |E| - k$.

Similarly, if we define the number of vertices dominated by vertex set *V'* as *gain*, and the size of *V'* as *loss* we obtain a problem called Inclusive Profit Domination (Stege & van Rooij, 2001).

Inclusive Profit Domination

Input: A graph $G = (V, E)$ and a positive integer *p*.

Question: Does there exist a vertex set $V' \subseteq V$ such that $\text{profit}_{\text{IPD},G}(V') \geq p$? Here $\text{profit}_{\text{IPD},G}(V') = |\text{N}_G[V']| - |V'| = |\text{N}_G(V')|$, with $\text{N}_G[V']$ and $\text{N}_G(V')$ denoting the closed and open neighborhoods of *V'* respectively.

A graph *G* has a dominating set of size at most *k* if and only if there exists a vertex set *V'* with $\text{profit}_{\text{IPD}G}(V') \geq p = |V| - k$. Thus, *p*-Inclusive Profit Domination is a relational parameterization for Dominating Set, with the parameter being the function $p = |V| - k$.

Note that profit relaxation does not always lead to an alternative parameterization of a problem. For example, a graph *G* has a non-blocking set of size at most *k* if and only if there exists a vertex set *V'* with $\text{profit}_{\text{IPD},G}(V') \geq p = k$, and thus *p*-Inclusive Profit Domination is also a profit relaxation of Non-Blocker. In this case, however, the profit parameterization is the same as the natural parameterization *k*-Non-Blocker, since $p = k$.

A similar situation arises when we derive a profit relaxation of Independent Set as follows: Let us define the number of vertices in a set *V'* as *gain*, and the number edges with both endpoints in *V'* as *loss*. Then we obtain a problem called Profit Independence (Stege & van Rooij, 2001).

Profit Independence

Input: A graph $G = (V, E)$ and a positive integer *p*.

Question: Does there exist a vertex set $V' \subseteq V$ such that $\text{profit}_{\text{PI},G}(V') \geq p$? Here $\text{profit}_{\text{PI},G}(V') = |V'| - |\text{E}_G(V')|$.

A graph *G* has an independent set of size at most *k* if and only if there exists a vertex set *V'* with $\text{profit}_{\text{PI},G}(V') \geq p = k$ (for a proof see Lemma 4.3 in Section 4.6). Hence, the natural parameterization of Profit Independence is the same as the natural parameterization of Independent Set.

4.4.3. Multiple-parameter Parameterizations

We have considered explicit parameters, implicit parameters and relational parameters—but what about parameterizing a problem by more than one parameter at once? This type of parameterization we call *multiple-parameter parameterization*. In the case of multiple-parameter parameterization the parameter κ is a set consisting of at least two parameters (e.g. Downey & Fellows, 1999, Neidermeier, 2002).

Let $\kappa = \{k_1, k_2, \ldots. k_x\}$, with $x \geq 1$. We call κ a *parameter set*.[43] We denote a problem Π parameterized by parameter set κ in one of two ways: either as κ-Π with $\kappa = \{k_1, k_2, \ldots. k_x\}$, or as $\{k_1, k_2, \ldots. k_x\}$-$\Pi$. In the case that κ contains only one element, $\kappa = \{k\}$, we simply write k-Π instead of $\{k\}$-Π (as we have done so far).

Let Π be a problem and let k_1 and k_2 be two possible parameters for Π. It is important to realize that we can have $\{k_1, k_2\}$-$\Pi \in$ FPT even if k_1-$\Pi \notin$ FPT and k_2-$\Pi \notin$ FPT. Consider, for example, the problem Independent Set. As remarked above, k-Independent Set is W[1]-hard; and thus not in FPT (unless FPT = W[1]). Further, recall that we proved Δ-Vertex Cover $\notin$ FPT (unless P = NP) using the fact that Vertex Cover is NP-hard even for $\Delta = 4$ (Lemma 4.1, page 64). Since also Independent Set is NP-hard for $\Delta = 4$, we can conclude with the same argument that Δ-Independent Set $\notin$ FPT (unless P = NP). We will now show that $\{k, \Delta\}$-Independent Set is in FPT.

The argument is organized as follows: First, we define a branching rule (IS 1) for Independent Set. Second, we show that the rule (IS 1) constructs a search tree whose size is bounded by a function $f(k, \Delta) = (\Delta+1)^k$. Third, we show that each node in the search tree can be created in polynomial-time. Finally, we conclude a running time for $\{k, \Delta\}$-Independent Set that is $O((\Delta+1)^k\,|V|^2)$, which is fpt-time for parameter set $\kappa = \{k, \Delta\}$.

The branching rule (IS 1) uses the following observation: If V' is a largest independent set for graph G, and vertex v in G is not in V' then a neighbor v_i of v must be

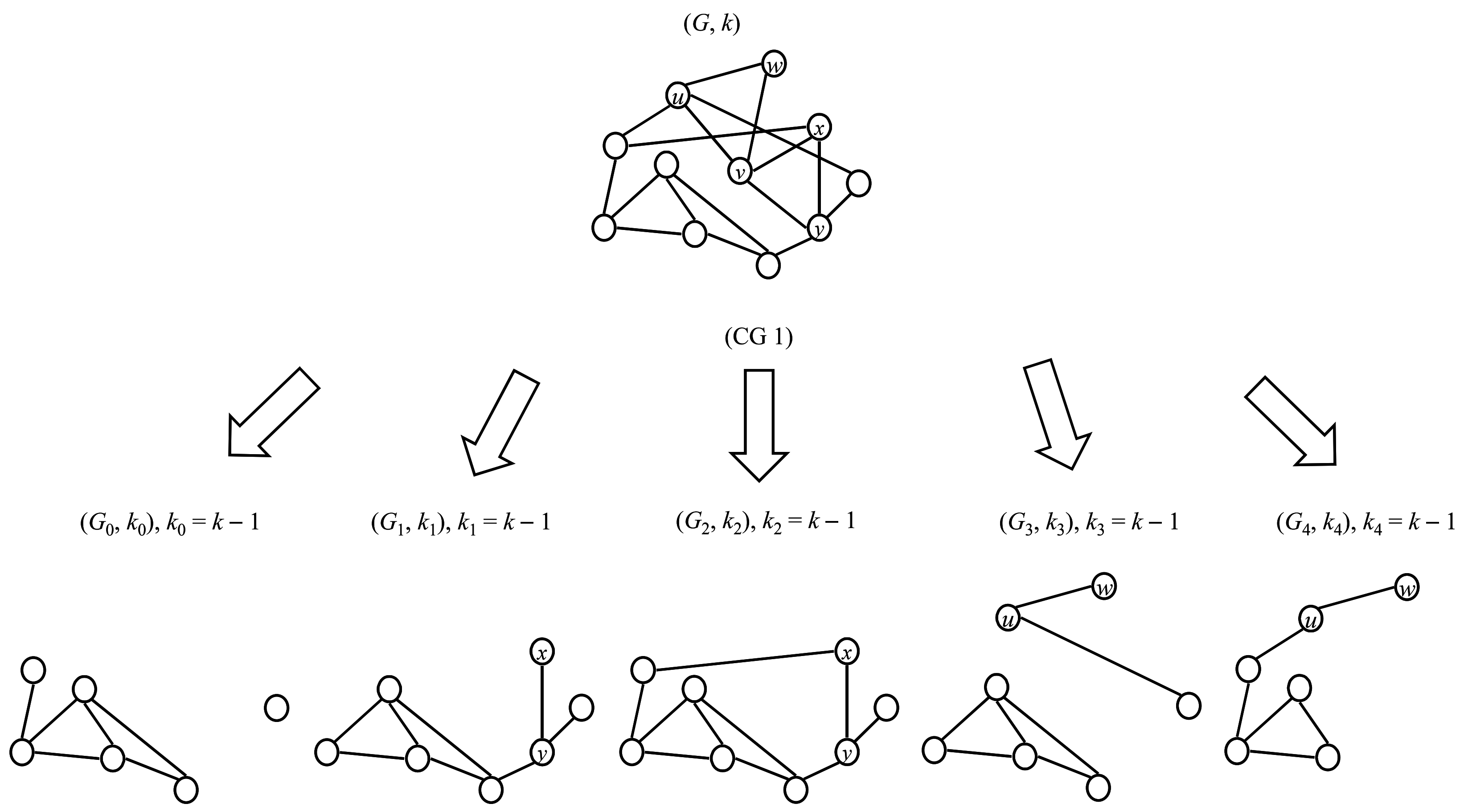
[43] Note that the definition of a parameter set incorporates the case where κ contains exactly one element as a special case.

in V' (otherwise, $V'' = V' \cup \{v\}$ would be an independent set that is larger than V', contradicting the fact that $|V'|$ is maximum). See Figure 4.4 for an illustration of (IS 1).

(IS 1) The Vertex-or-one-of-its-Neighbors Branching Rule: Let node s in the search tree be labeled by instance (G, k), $G = (V, E)$, for Independent Set, and let $v_0 \in V$. Then we create a child s_0 of s and label it by (G_0, k_0), with $G_0 = (V_0, E_0)$, $V_0 = V \backslash N_G[\{v_0\}]$, $E_0 = E \backslash R_G(\{v_0\})$, and $k_0 = k - 1$. Further, for each neighbor v_i of v, $i = 1, 2, \ldots, \deg_G(v)$, we create a child s_i of s and label it by (G_i, k_i), $G_\mathrm{i} = (V_i, E_i)$, $V_i = V \setminus N_G[\{v_i\}]$, $E_i = E \backslash R_G(\{v\})$, and $k_i = k - 1$.
To prove that (IS 1) is a valid branching rule as defined in Section 4.3, we need to show that (G, k) is a yes-instance for Independent Set if and only if (G_0, k_0) or (G_1, k_1) or (G_2, k_2) or … or $(G_{\deg(v)}, k_{\deg(v)})$ is a yes-instance for Independent Set.

Proof of (IS 1): $(\Rightarrow)$ Let (G, k) be a yes-instance. Then there exists an independent $V' \subseteq V$ for G, with $|V'| \geq k$. Since for each $j \in \{0, 1, 2, \ldots, \deg_G(v)\}$ at most one vertex in V' is also in $N_G[\{v_j\}]$ (i.e., $|N_G[\{v_j\}] \cap V'| = 1$; otherwise V' would not be an independent set) and $V_j = V \backslash N_G[\{v_j\}]$, we conclude that each G_j has an independent set of size at least $k - 1$. $(\Leftarrow)$ Assume that at least one instance (G_j, k_j), with $j \in \{0, 1, 2, \ldots, \deg_G(v)\}$, is a yes-instance for Independent Set. Then there exists an independent $V_j' \subseteq V$ for G_j, with $|V_j'| \geq k_j$. Since $N_G[\{v_j\}] \not\subset V_j$, and thus $N_G[\{v_j\}] \not\subset V_j'$, we conclude that $V_j' \cup \{v_j\}$ is an independent set for G of size $k_j + 1$. and $N_G[\{v_j\}]$. ■

We can use the branching rule (IS 1) to create a bounded search tree algorithm for $\{k, \Delta)$-Independent Set as follows. The algorithm takes as input an instance (G, k) and recursively applies (IS 1) to (G, k) until either an instance (G', k') is encountered with $k' \leq 0$ (in which case the algorithm returns the answer "yes") or (IS 1) cannot be applied anymore (this happens if $V' = \emptyset$). If the algorithm halts without returning the answer "yes" then we conclude that (G, k) is a no-instance for Independent Set (viz., we have shown in the proof above that the search tree will return the answer "yes" if and only if (G, k) is a yes-instance for Independent Set). This meets requirement (1) for a bounded search tree.



(G, k)
w
u
x
v
y
(CG 1)
$(G_0, k_0), k_0 = k - 1$
$(G_1, k_1), k_1 = k - 1$
$(G_2, k_2), k_2 = k - 1$
$(G_3, k_3), k_3 = k - 1$
$(G_4, k_4), k_4 = k - 1$

Figure 4.4. Illustration of branching rule (IS 1). See next page for description.

Figure 4.4. Illustration of branching rule (IS 1).
The instance before the application of the branching rule is denoted (G, k) and the instances obtained after the application of the branching rule on vertex v in G, are denoted (G_i, k_i) with $i = 0, 1, 2, \ldots, \deg_G(v)$. The rule (IS 1) assumes for every vertex v in G, that either v is in the independent set or it is not; further, if v is not in the independent set, then at least one of its neighbors, u, w, x, or y can be included in the independent set (otherwise v could be included in the independent set and we would have a larger independent set). Once a vertex is included in the independent set, we can delete it and all its neighbors from G. For each instance (G_i, k_i), $i = 0, 1, 2, \ldots, \deg_G(v)$, we define $k_i = k - 1$, to reflect the fact that a vertex has already been included in the independent set. As soon as we encounter an instance (G_i, k_i) with $k_i = 0$ we return the answer "yes" (i.e., we conclude that the original input has an independent set of size at least k) and if we cannot apply (IS 1) anymore, then we have exhausted the search tree and we return the answer "no" (i.e., we conclude that the original input does not have an independent set of size at least k). Note that in this example, $\deg_G(v) = 4$, and thus application of rule (IS 1) leads to the creation of five new instances, (G_0, k_0), (G_1, k_1), (G_2, k_2), (G_3, k_3), and (G_4, k_4). Further, note that these new instances are defined such that (G_0, k_0) or (G_1, k_1) or (G_2, k_2) or (G_3, k_3) or (G_4, k_4) is a yes-instance for Vertex Cover if and only if (G, k) is a yes-instance for Vertex Cover. This way, rule (IS 1) ensures that the branching algorithm returns the answer "yes" (or "no") if and only if the answer for the original input to the algorithm is "yes" ("no").

To see that the algorithm meets requirement (2) with respect to the parameter set $\kappa = \{\Delta, k\}$, first observe that each application of (IS 1), to a vertex v_0, leads to the creation of $1 + \deg_G(v_0) \leq 1 + \Delta$ new branches in the search tree T. Hence we have $\text{fan}(T) \leq 1 + \Delta$. Further, whenever (IS 1) creates a node labeled by (G', q') for a parent labeled by (G, q) then $k' \leq k - 1$, and thus $\text{depth}(T) \leq k$. We conclude that the size of the search tree is at most $O((1 + \Delta)^k)$.

Finally, with respect to requirement (3), we show that we can create each node in the search tree in time $O(|V|^2)$: First, to apply (IS 1) to an instance (G, k), $G = (V, E)$, we need to pick an arbitrary vertex $v_0 \in V$. This we can do in time $O(1)$. Then, for each new search tree node s_j that we create we spend at most $O(|V|^2)$ time to label it by (G_j, k_j). Namely, for each v_j we spend time at most $O(|V|^2)$ to delete all vertices in $N_G[\{v_j\}]$ and all edges adjacent to vertices in $N_G[\{v_j\}]$. Combined with the size of the search tree we can conclude that the algorithm for Independent Set runs in time $O((1 + \Delta)^k\ |V|^2)$.

4.5. Crucial Sources of Complexity[44]

In Section 4.4.3, I have illustrated how a problem (Independent Set), that is not in FPT (unless W[1] = FPT) for either of two parameters (Δ and k), can nevertheless be in FPT when both parameters are in the parameter set. Hence, knowing that two parameterizations κ-Π and κ'-Π, $\kappa \neq \kappa'$, are not in FPT, tells us nothing about the complexity class of the parameterization $(\kappa \cup \kappa')$-Π. On the other hand, if κ-Π or κ'-Π, $\kappa \neq \kappa'$, are in FPT then we can conclude that also $(\kappa \cup \kappa')$-$\Pi \in$ FPT. In other words, we have Observation 4.1.

Observation 4.1. If a parameterized problem κ-$\Pi \in$ FPT, then for any superset $\kappa' \supseteq \kappa$, κ'-$\Pi \in$ FPT.

To see why Observation 4.1 holds consider the following: The fact that κ-$\Pi \in$ FPT means that the problem Π can be solved in time $O(f(\kappa)|i|)$. Now note that $O(f(\kappa)|i|)$ is fpt-time for any parameter set $\kappa' \supseteq \kappa$. That the function $f(.)$ will be independent of some parameters in the set κ' does not matter; the only requirement for the function $f(.)$ is that it does not depend on $|i|$. With this observation we can conclude from $\{\Delta, k\}$-Independent Set $\in$ FPT that κ-Independent Set is in FPT for all parameters sets κ, with $\{\Delta , k\} \subseteq \kappa$. For completeness, I note that Observation 4.2 is a direct consequence of Observation 4.1.

Observation 4.2. If a parameterized problem κ-$\Pi \notin$ FPT, then for any $\kappa' \subseteq \kappa$, κ'-$\Pi \notin$ FPT.

Now consider a problem Π that is not in P. Say we know a parameter set κ such that κ-$\Pi \in$ FPT. Then we know that the parameters in the set κ are *sufficient* for confining the non-polynomial time complexity inherent in Π. From Observation 4.1, however, we know that κ may contain one or more parameters that are 'redundant,' in the

[44] I introduce the notion of 'crucial source of complexity' (see also van Rooij et al., 2003). To my knowledge, the notion has not yet been so defined in the parameterized complexity literature (but see e.g. Wareham, 1998, for a discussion of a related notion called 'intractability map'). My definition is motivated by the fact that some parameter sets may contain 'redundant' parameters, i.e., parameters that are not necessary for confining the exponential complexity, and thus, in a sense, do not contribute to the result that the problem is in FPT for that parameter set. It seems to me that in cognitive theory it may often be of interest to know whether or not a parameter set contains such redundant elements.

sense that the fpt-classification of κ-Π does not depend on those parameters. In those cases we say κ *not minimal*. More formally, if κ-$\Pi \in$ FPT and there does *not* exist a proper subset $\kappa' \subset \kappa$ such that κ'-$\Pi \in$ FPT, then we say κ is a *minimal* parameter set for Π.

If κ is a minimal parameter set for a classically intractable problem Π we call κ a *crucial source of (non-polynomial time) complexity* in Π, because then Π is "easy" for small values of parameters in κ *irrespective* the size of other input parameters. Note that a crucial source of complexity need not be unique. That is, there may exist different parameter sets κ and κ', with $\kappa' \not\subset \kappa$, such that κ-$\Pi \in$ FPT and κ'-$\Pi \in$ FPT. Also note that every problem has $\kappa = \{|i|\}$ as a crucial source of complexity. Thus the label 'crucial source of complexity' should be read as denoting a *minimal* set of parameters that is *sufficient*, but not necessary, for confining the non-polynomial time behavior in a problem.

To illustrate, we again consider $\{\Delta, k\}$-Independent Set. We have shown in Section 4.4.3 that $\{\Delta, k\}$-Independent Set $\in$ FPT. Thus, the parameter set $\kappa = \{\Delta, k\}$ is *sufficient* for confining the non-polynomial time complexity in the problem. Is $\{\Delta, k\}$ a *minimal* parameter set for Independent Set? Yes, it is (unless FPT = W[1]). As discussed in Section 4.4.3, Δ-Independent Set is not in FPT (unless P = NP) and k-Independent Set is not in FPT (unless FPT = W[1]). This establishes that there does not exist any subset $\kappa' \subseteq \{\Delta, k)$, such that κ-Independent Set is in FPT (unless FPT = W[1]). We conclude that $\{\Delta, k\}$ is a crucial source of complexity for Independent Set.

4.6. Parametric reduction

This section explains and illustrates the technique of parametric reduction. I restate the definition of parametric reduction from Section 3.4.1 (page 44): For parameterized problems κ_1-Π_1 and κ_2-Π_2 and functions f and g, we say a *parametric reduction* from κ_1-Π_1 to κ_2-Π_2 is a reduction that transforms any instance i_1 for κ_1-Π_1 into an instance i_2 for κ_2-Π_2 with $\kappa_2 = g(\kappa_1)$. Further, the reduction runs in time $f(\kappa_1)|i_1|^{\alpha}$, where α is a constant. In Chapter 3 we have already seen a parametric reduction: The polynomial-time reduction from Vertex Cover to Dominating Set in Lemma 3.1 (page 37; see also Figure

3.2) happens to be a parametric reduction from k_V-Vertex Cover to k_D-Dominating set. Namely, the reduction involves a transformation from an instance (G_V, k_V) for k_V -Vertex Cover to an instance (G_D, k_D) for k_D-Dominating Set, such that $k_D = g(k_V) = k_V$. Further, the transformation runs in time $O(|V_V|^2)$, which is polynomial time (and thus also fpt-time).

To illustrate a parametric reduction that runs in fpt-time, but not in polynomial-time, we adapt the reduction in Lemma 3.1 as follows: Instead of appending for each edge (u, v) in G one vertex x_{uv} in G^*, we append 2^k vertices (denoted $x_{uv,i}$ with $i = 1, 2, \ldots, 2^k$) per edge (u,v) in G. Lemma 4.2 presents this reduction. See Figure 4. for an illustration.

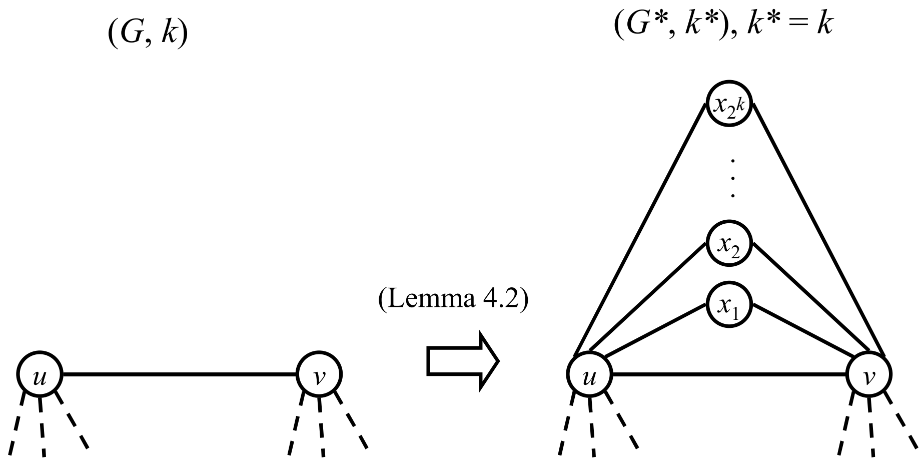


Figure 4.5. Illustration of the reduction in Lemma 4.2.
Every edge (u, v) in G (on the left) is copied to G^* (on the right). Further in G^* we append 2^k vertices on u and v, denoted $x_1, x_2, \ldots, x_{2^k}$. Note that ($G$, k) is a yes-instance for Vertex Cover if and only if (G^*, k^*), with $k^*= k$, is a yes-instance for Dominating Set.

Note that the transformation in Lemma 4.2 can be done in fpt-time: We can copy $G = (V, E)$ to $G^* = (V^*, E^*)$ in time $O(|V|^2)$, and we can add the $2^k\,|E|$ extra vertices and their $2^{k+1}|E|$ incident edges, in time $O(2^k\,|E|)$ which is $O(2^k\,|V|^2)$. Thus, the total time required for the transformation is $O(|V|^2 + 2^k\,|V|^2)$ or $O(2^k\,|V|^2)$, which is fpt-time for parameter k. Further, as in Lemma 3.1, Lemma 4.2 shows that (G, k) is a yes-instance for Vertex Cover if and only if (G^*, k^*), is a yes-instance for Dominating Set with $k^* = k$. Hence, the reduction is a parametric reduction.

Lemma 4.2. Let $G = (V, E)$ be a graph without singletons and let k be a positive integer. We create a graph $G^* = (V^*, E^*)$ as follows. For every $(u, v) \in E$, let $(u, v) \in E^*$. Further, for each $(u, v) \in E$ we include in G^* 2^k new vertices $x_{uv,i} \notin G$ and attach them to $u,v \in E^*$ using the edges $(u, x_{uv,i})$ and $(v, x_{uv,i})$, $i = 1, 2, \ldots, 2^k$. Finally, let $k^* = k$. Then G and k form a yes-instance for Vertex Cover if and only if G^* and k^* form a yes-instance for Dominating Set.

Proof: ($\Rightarrow$) Let (G, k) be a yes-instance for Vertex Cover. Then, we know that for every vertex $v \in V$, $v \in V'$ or v has at least one neighbor $u \in V'$ (Observation 3.1, page 36). Further, because every vertex $x_{uv,i} \in V^*\backslash V$ is adjacent to both $u \in V^*$ and $v \in V^*$, with $(u, v) \in E$, we also know every vertex $x_{uv} \in V^*\backslash V$ has at least one neighbor in V'. Thus V' is a dominating set for G^* with $|V'| \leq k = k^*$. We conclude (G^*, k^*) is a yes-instance for Dominating Set.

($\Leftarrow$) Let (G^*, k^*) be a yes-instance for Dominating Set, and let $V' \subseteq V^*$ be a minimum dominating V' for G^*. (1) We show V' contains only vertices in V (i.e., $V' \cap V^*\backslash V = \emptyset$). Namely, for every edge $(u, v) \in E$, to dominate in G^* each vertex in $\{u, v, x_{uv,1}, x_{uv,2}, \ldots, x_{uv,2^k}\} \subseteq V^*$, we can either include u in V' or v in V' or at least 2 vertices from $\{u, v, x_{uv,1}, x_{uv,2}, \ldots, x_{uv,2^k}\}$, since $k \geq 1$. Therefore, a minimum dominating set will include u or v, and no vertices from $\{x_{uv,1}, x_{uv,2}, \ldots, x_{uv,2^k}\}$. (2) Since $V' \subseteq V$, and for each edge $(u, v) \in E$, $u \in V'$ or $v \in V'$, we conclude that V' is a vertex cover for G with $|V'| \leq k = k^*$. We conclude that (G, k) is a yes-instance for Vertex Cover. ■

A word of warning on what can, and what cannot, be concluded from Lemma 4.2: Because the reduction in Lemma 4.2 does not run in polynomial-time, it does not imply anything for the classical complexity of Dominating Set. Thus, unlike Lemma 3.1, Lemma 4.2 is *not* a proof that Dominating Set is NP-hard. Further, because k-Vertex Cover is in FPT, all that Lemma 4.2 shows is that k-Dominating Set is, in the parameterized sense, "at least as hard" as k-Vertex Cover. It does not tell us, however, whether k-Dominating Set is strictly harder than k-Vertex Cover (Is it W[1]-hard?), or that it is equally hard (Is it in FPT?).[45]

[45] Of course, as noted in Section 4.4.2, it is known that k-Dominating Set is W[1]-hard. My argument here is that we cannot conclude this fact from Lemma 4.2.

Although Lemma 4.2 serves its purpose as an example of a parametric reduction, we are typically more interested in finding either (1) a parametric reduction from a known W[1]-hard problem Π to an unclassified problem Π', or (2) a parametric reduction from an unclassified problem Π' to a known FPT problem Π. In the first case we can conclude that Π' is W[1]-hard, while in the second case we can conclude that Π' is in FPT. To illustrate the first case I present a parametric reduction from k-Independent Set to p-profit Independence. This reduction transforms an instance (G, k) for Independent Set into an instance (G, p) for Profit Independence, with $p = k$.

Lemma 4.3. Let (G, k) be an instance for Independent Set. Further, let (G, p) be an instance for Profit Independence with $p = k$. Then (G, k) is a yes-instance for Independent Set if and only if (G, p) is a yes-instance for Profit Independence.

Proof: $(\Rightarrow)$ Let (G, k), $G = (V, E)$, be a yes-instance for Independent Set. Then there exists an independent set $V' \subseteq V$ for G with $|V'| \geq k$. This means that $E_G(V') = \varnothing$, and thus $\text{profit}_{PI,G}(V') = |V'| - |E_G(V')| = |V'| \geq k = p$. We conclude that (G, p) is a yes-instance of Profit Independence.

$(\Leftarrow)$ Let (G, p), $G = (V, E)$, be a yes-instance for Profit Independence. Then there exists a subset $V' \subseteq V$ with $\text{profit}_{PI,G}(V') \geq p$. We distinguish two cases: (1) If $E_G(V') = \varnothing$ then V' is an independent set for G, with $|V'| \geq p = k$. We conclude that G and k form a yes-instance of Independent Set. (2) If $E_G(V') \neq \varnothing$ then we transform V' into an independent set V'' for G using the following algorithm:

```
1.  $V'' \leftarrow V'$
2.  while $E_G(V'') \neq \varnothing$ do
    3.  pick an edge $(u, v) \in E_G(V'')$
    4.  $V'' \leftarrow V'' \backslash \{v\}$
5.  end while
6.  return $V''$
```

The algorithm considers each edge in G at most once and thus runs in time $O(|V|^2)$. Note that every call of line 4 results in the removal of at least one edge from $E_G(V'')$. Hence, $\text{profit}_{PI,G}(V'') \geq \text{profit}_{PI,G}(V') \geq p$. Furthermore, when the algorithm halts then $E_G(V'') = \varnothing$

and thus *V”* is an independent set of size at least $p = k$ for G. We conclude that G and k form a yes-instance for Independent Set. ■

Note that the transformation of (G, k) to (G, p), in Lemma 4.3, can be done in polynomial-time: time $O(1)$ to set $p = k$. Further, since p is a function of k only, the reduction is a parametric reduction. We know k-Independent Set is W[1]-hard, so we can conclude from Lemma 4.3 that p-Profit Independence is W[1]-hard as well.[46]

4.7. The Parametric Toolkit and Beyond

In this chapter I have reviewed a set of basic techniques for parameterized complexity analyses. The techniques discussed here do not exhaust all known techniques for parameterized complexity analysis. The reader interested in learning about other techniques is referred to the relevant literature (see e.g. Alber, Fan, Fellows, Fernau, Niedermeier, Rosamond & Stege, 2001; Fellows, 2002; Fellows, McCartin, Rosamond, & Stege, 2000; Downey & Fellows, 1999; Downey, Fellows & Stege, 1999a, 1999b; Gottlob, Scarcello, & Sideri, 2002, Khot & Raman, 2000; Prieto & Sloper, forthcoming; Niedermeier, 2002; Niedermeier & Rossmanith, 1999, 2000; Stege et al., 2002; Stege & van Rooij, 2003). The relatively informal introduction that I have presented in this chapter can also aid understanding of this more formal work.

The techniques I have presented in this chapter are simple but powerful. In the application of complexity theory to cognitive theory, as pursued here, these techniques can already bring us a long way. I will illustrate this in the following chapters (Chapters 5, 6 and 7), by illustrating each technique in the context of existing cognitive theories.

[46] Note that this parametric reduction is also a polynomial-time reduction. Thus, since Independent Set is NP-hard, Lemma 4.2 also proves that Profit Independence is NP-hard (see also Section 6.3).

Chapter 5. Coherence

In this chapter we consider the problem Coherence as defined by Thagard (2000) and Thagard and Verbeurgt (1998). We start with the problem's definition and the motivation for studying this problem. The main part of this chapter will be devoted to complexity analyses of the problem Coherence (and its variants), both from a classical and from a parameterized perspective. Note that in interpreting the results we will assume P ≠ NP and FPT ≠ W[1]. I close with a brief discussion and suggestions for future research.

5.1. Coherence as Constraint Satisfaction

This section presents the notation and terminology to formally define the Coherence problem. The next section (Section 5.2) describes applications of this problem in cognitive theory as proposed by, among others, Thagard and Verbeurgt (1998) and Thagard (2000).

Two sets S_1 and S_2 are called a *partition* of a set S if $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \varnothing$. We denote a partition of a set S into sets S_1 and S_2 by $(S_1 \cup S_2)^S$. In general, if S partitions into sets $S_1, S_2 \ldots, S_x$, with $x \geq 2$ and $S_1, S_2 \ldots, S_x$ all pairwise disjoint, we write $(S_1 \cup S_2 \cup \ldots \cup S_x)^S$. Let P be a set of elements,[47] and let $C \subseteq P \times P$ denote a set of unordered pairs of elements in P. A pair $(p, q) \in C$ we call a *constraint* between p and q. The set C is partitioned into two sets of constraints C^+ and C^-. We call a constraint $(p, q) \in C^+$ a *positive constraint* and we say that p and q *cohere*. We call a constraint $(p, q) \in C^-$ a *negative constraint* and we say that p and q *incohere*. Each constraint $(p, q) \in C$ has an associated positive integer weight $\mathrm{w}(p, q)$.

Note that this information can be thought of as an edge-weighted graph $N = (P, C)$, with vertex set P and weighted edge set C (cf. Thagard & Verbeurgt, 1998). To stay as close as possible to the terminology of Thagard and Verbeurgt, I will refer to $N = (P, C)$ as a *network*, to the vertices in P as *elements* and to the edges in C as *constraints*. However, I will use standard graph terminology to refer to properties of, and

[47] Thagard and Verbeurgt (1998) use the symbol E to refer to the set of elements. I do not use the symbol E here to avoid confusion with the edge set E in a graph $G = (V, E)$. I choose to use the symbol P instead because in many applications described by Thagard (2000) elements in this set are *propositions*.

relationships between, elements, constraints, and networks (cf. Appendix A); e.g., I will speak of the *degree* of an element, a constraint being *incident* to an element, the *endpoints* of a constraint, two elements being *neighbors*, the number of *components* in a network, networks that are *trees*, a network being a *subgraph* of another network, etc.

Elements in P can be either *accepted* or *rejected*. A denotes the set of accepted elements in P and R denotes the set of rejected elements. The sets A and R are mutually disjoint (i.e., we cannot accept and reject one and the same element). We say the sets A and R form a *partition* of P if $P = A \cup R$, and write $(A \cup R)^P$. A partition $(A \cup R)^P$ can satisfy one or more constraints in C. A positive constraint $(p, q) \in C^+$ is *satisfied* by partition $(A \cup R)^P$ if and only if ($p \in A$ and $q \in A$) or ($p \in R$ and $q \in R$). A negative constraint $(p, q) \in C^-$, is *satisfied* by partition $(A \cup R)^P$ if and only if ($p \in A$ and $q \in R$) or ($p \in R$ and $q \in A$). The set of all (positive and negative) constraints in network N that are satisfied by partition $(A \cup R)^P$ is denoted by $\mathrm{S}_N(A, R)$. Partitions can differ in the amount of *coherence*. Coherence is measured by the function $\mathrm{Coh}_N(A, R) = \sum_{(p,q)\in \mathrm{S}_N(A,R)} \mathrm{w}(p,q)$. Note that the subscript $_N$ on the functions $\mathrm{S}_N(.,.)$ and $\mathrm{Coh}_N(.,.)$ denotes that they are to be evaluated in the context of a specific network N.

Having introduced the necessary terminology, we can now state the problem Coherence (Thagard, 2000; Thagard & Verbeurgt, 1998):

Coherence (*optimization version*)

Input: A network $N = (P, C)$, with $(C^+ \cup C^-)^C$. For each constraint $(p, q) \in C$ there is an associated positive integer weight $\mathrm{w}(p, q)$.

Output: A partition of P into A and R such that $\mathrm{Coh}_N(A, R)$ is maximized.

In the remainder of this chapter we will work with the decision version of Coherence.

Coherence (*decision version*)

Input: A network $N = (P, C)$, with $(C^+ \cup C^-)^C$. For each constraint $(p, q) \in C$ there is an associated positive integer weight $\mathrm{w}(p, q)$. A positive integer c.

Question: Does there exist a partition of P into A and R such that $\mathrm{Coh}_N(A, R) \geq c$?

Keep in mind that there is a close relationship between the decision version and optimization version. If we know an optimal partition $(A \cup R)^P$ (i.e., a partition $(A \cup R)^P$ such that $\mathrm{Coh}_N(A, R)$ is maximum), then we also know if there exists a partition with

$\mathrm{Coh}_N(A, R) \geq c$, for any c. Conversely, we can determine an optimal partition by solving the decision version of Coherence for different c. If the answer “yes” is returned for c, while for $c + 1$ the answer is “no,” then we can conclude that c is maximum.

5.2. Coherence as Cognitive Theory

According to Thagard (2000; see also Eliasmith & Thagard, 1997; O'Laughlin & Thagard, 2000; Thagard, 1989, 1999, 2001; Thagard, Eliasmith, Rusnock, & Shelley, 2002; Thagard & Kunda, 1998; Thagard & Shelley, 1997; Thagard and Verbeurgt, 1998), the Coherence problem (and variants thereof) can be used to model many different cognitive tasks in various cognitive domains; including domains such as scientific explanation, legal justification, social judgment, and visual perception. I briefly sketch applications in each domain below. See Thagard (2000), and the references above, for more detailed discussions of these and other areas of applications.

Scientific Explanation:[48] According to Thagard (2000; see also Eliasmith & Thagard, 1997; Thagard, 1989, 2000; Thagard and Verbeurgt, 1998), the task of a scientist (or group of scientists) to decide upon a set of hypotheses that satisfactorily explains a set of empirical data can be modeled as follows. Let the elements in P represent propositions describing the empirical observations as well as the different possible hypotheses about what brought about the data. The set C models the coherence relationships between pairs of propositions; e.g., if hypothesis q explains observation p then p and q cohere; if hypothesis q and r each explains observation p, then q and r incohere; if hypothesis q and r together explain observation p, then q and r cohere. The task of the scientist is to decide on a set of hypotheses that provides the most coherent explanation of the data.

Legal Justification: According to Thagard (2000; see also Eliasmith & Thagard, 1997; Thagard, 1989, 2000; Thagard and Verbeurgt, 1998), the task of a jury member to decide whether a defendant is innocent or guilty can be modeled as follows. Let the elements in P represent propositions describing factual evidence brought to trial as well as the different hypotheses one may entertain about the case. Let $s \in P$ denote the

[48] In this context it is of interest to note that Thagard also proposes that his notion of coherence explains why computer scientists believe that P ≠ NP (Thagard, 1993).

proposition 'defendant is innocent.' The set C models the conceptual relationships between pairs of propositions; e.g., if evidence p is explained by hypothesis q then p and q cohere; if hypothesis p and q contradict then p and q incohere. Then, the task of a jury member is to decide whether it is more coherent to believe that $s \in A$ or that $s \in R$.

Social Judgment: Given the example above it is not hard to see how also a social judgment task can be conceptualized as a coherence problem. Take any form of social attribution; e.g., 'X is honest,' 'X believes Y.' Then the task of deciding whether the attribution should (or should not) be made—based upon one's experiences with a person and one's knowledge about states of the world—can be seen as analogous to the task of deciding whether a defendant is innocent or not (see also O'Laughlin & Thagard, 2000; Thagard, 1989, 2000; Thagard & Kunda, 1998; Thagard and Verbeurgt, 1998).

Visual Perception: Because sensory inputs typically give incomplete and/or ambiguous information about a visual scene the visual system faces a problem: How to interpret sensory information in a reasonable and coherent way? The task of interpreting a visual scene can be modeled as follows (Thagard, 1989, 2000; Thagard & Shelley, 1997; Thagard and Verbeurgt, 1998). Let the elements in P represent the set of sensory data and/or possible interpretations of the data. Then set C models the coherence relationships between data and interpretation and between different interpretations: e.g., if sensory datum p supports interpretation q then p and q cohere; if interpretation p and q are inconsistent then p and q incohere. The task of the visual system is to decide on a set of interpretations that maximizes coherence. Figure 5.1 gives an example of this model using the Necker Cube (cf. Thagard 1989, p. 439).

In summary, given the wide variety of domains in which Coherence finds application, the problem is clearly of importance to cognitive science. The remainder of this chapter is devoted to studying the problem's complexity. To be clear, I will not evaluate Coherence, other than with respect to complexity. For other types of evaluations and critical discussions on Coherence see, for example, Schoch (2000), Millgram (2000), and Wiedemann (1999).

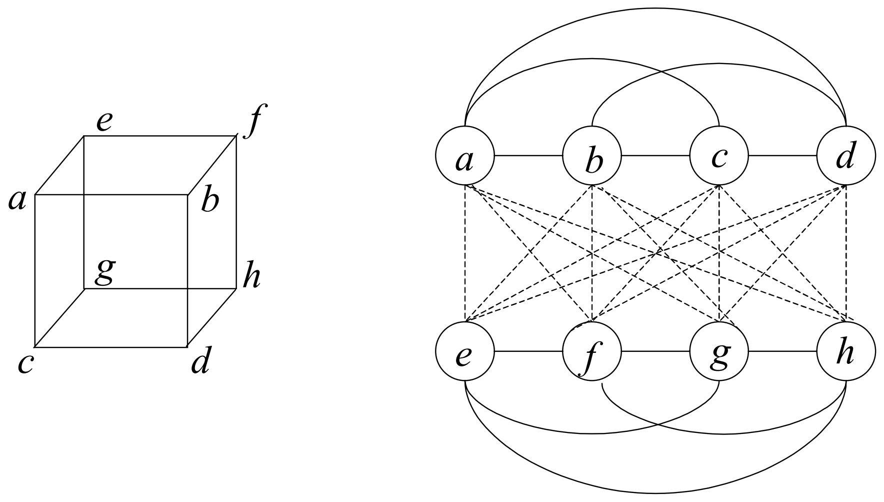


Figure 5.1. Example of a Coherence poblem
(left) The Necker cube can be interpreted in two qualitatively different ways: Either the face with vertices *a*, *b*, *c* and *d* is in the front (in which case the face with vertices *e*, *f*, *g* and *h* is in the back) or the face with vertices *e*, *f*, *g* and *h* is in the front (in which case the face with vertices *a*, *b*, *c* and *d* is in the back). The task of disambiguating between the two possible interpretations can be modeled as a coherence problem: (right) Every vertex $p \in \{a, b, c, d, e, f, g, h\}$ of the Necker cube is modeled by an element $p \in P$. If two vertices *p* and *q* in *P* are part of the same interpretation of the Necker cube then $p, q \in C^+$ (solid lines), else $p, q \in C^-$ (dotted lines). Now a partition of *P* of maximum coherence into vertices that are "in the front", *A*, and vertices that are "in the back," *R*, will correspond to one of the two possible interpretations of the Necker cube.

5.3. Coherence is NP-hard

This section recapitulates Thagard and Verbeurgt's (1998) proof that Coherence is NP-hard. The reduction is from the graph problem Max-Cut. Max-Cut takes as input an edge weighted graph $G = (V, E)$. In its optimization version, the goal is to partition the vertex set *V* into sets *A* and *R* such that the sum of the weights on edges that have an endpoint in *A* and an endpoint in *R* is maximum. The decision version of this problem is as follows:

Max-Cut (*decision version*)

Input: An edge weighted graph $G = (V, E)$. For each edge $(u, v) \in E$ there is an associated positive integer weight $\mathrm{w}(u, v)$. A positive integer *k*.

Question: Does there exist a partition of *V* into sets *A* and *R* such that $\mathrm{W}_G(A, R) = \sum_{(u,v)\in \mathrm{Cut}_G(A,R)} \mathrm{w}(u,v) \geq k$? Here $\mathrm{Cut}_G(R, A) = \{(u, v) \in E : u \in A \text{ and } v \in R\}$.

Max-Cut is known to be NP-complete (Garey & Johnson, 1979). The following lemma presents a reduction from Max-Cut to Coherence.

Lemma 5.1. (Thagard & Verbeurgt, 1998)[49] Let graph $G = (V, E)$ and positive integer k form an instance for Max-Cut. We define an instance, consisting of N and c, for Coherence as follows: Let $N = (P, C)$ be a network such that $P = V$, $C = E$ and $C = C^-$ (i.e., all constraints in C are defined to be negative constraints). Further for all $(p, q) \in C$ set the weight $\mathrm{w}_N(p, q) = \mathrm{w}_G(p, q)$. Finally, let $c = k$. Then (G, k) is a yes-instance for Max-Cut if and only if (N, c) is a yes-instance for Coherence.

Proof: Since $C = C^-$, we know that $(p, q) \in \mathrm{Cut}_G(A, R)$ if and only if $(p, q) \in \mathrm{S}_N(A, R)$. Further, the edge weights in G and N are identical, and thus $\mathrm{Coh}_N(A, R) = \mathrm{W}_G(A, R)$ for any partition $(A \cup R)^P$. We conclude that (G, k) is a yes-instance for Max-Cut conclude if and only if (N, c) is a yes-instance for Coherence. ■

The transformation from the instance (G, k) for Max-Cut into a corresponding instance (N, c) for Coherence described in Lemma 5.1 can be done in polynomial time. Namely, we can copy every element in V and E to P and $C = C^-$ respectively, in time $O(n^2)$, we set $c = k$ in $O(1)$, and we assign each constraint in C its weight in time $O(n^2)$. We conclude, the reduction in Lemma 5.1 is a polynomial-time reduction, and thus:

Corollary 5.1. (Thagard & Verbeurgt, 1998) Coherence is NP-hard.

Note that the instance for Coherence, created in lemma 5.1, contains only negative constraints. This means that Lemma 5.1 also describes a reduction from Max-Cut to Coherence on networks with negative constraints only. Hence we also have:

Corollary 5.2. (Thagard & Verbeurgt, 1998) Coherence on a network $N = (P, C)$, with $C = C^-$, is NP-hard.

Further, it is known that Max-Cut problem remains NP-hard even if all constraints have weight '1' (Garey & Johnson, 1979). Hence, from the reduction in Lemma 5.1, we also conclude:

Corollary 5.3. Coherence on a network $N = (P, C)$, with $C = C^-$ and $\mathrm{w}(p, q) = 1$ for all $(p, q) \in C$, is NP-hard.

[49] It should be noted that Thagard and Verbeurgt (1998) give the same reduction as I do here with the exception that they set $c = 2k$, instead of $c = k$. I could not verify the correctness of their reduction (unless Thagard and Verbeurgt count all (negative) constraint weights twice in the computation of coherence; which does not seem to fit the problem's definition). In any case, Lemma 5.1 gives the correct reduction from Max-Cut to the Coherence problem, as defined in Section 5.1.

5.4. Reflections on the NP-hardness of Coherence

We have seen that the problem Coherence, as defined in Section 5.1, is NP-hard. Note that this NP-hardness result only holds for the general problem Coherence (with or without positive constraints). It does not imply that all problems modeled by Coherence are NP-hard. This difference between 'a general problem being NP-hard' and 'a problem being generally NP-hard' is often confused. Consider, for example, the following synopsis by Thagard (2000, p. 15):[50]

> "Coherence problems are inherently computational intractable, in the sense that, (…) [assuming P ≠ NP], there are no efficient (polynomial-time) procedures for solving them."

Here Thagard uses the words "coherence problems" to refer to a general class of problems that can be modeled by Coherence. Hence, the statement suggests that coherence problems are of exponential-time complexity *across-the-board*. Clearly the finding that Coherence is NP-hard does not warrant this conclusion: NP-hardness is not automatically inherited by every special case, or variant, of an NP-hard problem.

This section discusses in more detail when an NP-hardness result automatically generalizes to another problem (if the other problem involves a generalization of the original problem) and when it does not (if the other problem is a special case or a variant of the original problem). Motivated by the applications described in Section 5.2, I consider some special cases of Coherence (Section 5.4.1), some generalized versions of Coherence (Section 5.4.2) and some variants of Coherence (Section 5.4.3).

5.4.1. Special Cases of Coherence

Thagard and Verbeurgt (1998) remark that the special case of Coherence, in which the network contains only positive constraints (no negative constraints), is not NP-hard.

[50] Similarly, Oaksford and Chater (1998) write: "Consistency checking constitutes a general class of problems in complexity theory called satisfiability problems" (p. 76). Further, because Cook (1971) has shown that the satisfiability problem SAT is NP-complete (see also Section 3.2), Oaksford and Chater (1998) conclude that "consistency checking, like all satisfiability problems, is NP-complete" (p. 77). This statement, like Thagard's quoted in the text, is misleading. Clearly, there exist satisfiability problems that are in P (e.g. 2-SAT; Garey & Johnson, 1979; see also Appendix B).

Namely, in that case, we always satisfy all constraints if we set $A = P$ and $R = \varnothing$ (or, equivalently, if we set $A = \varnothing$ and $R = P$). Hence we have the following result:

Observation 5.1. (Thagard & Verbeurgt, 1998). Coherence on a network $N = (P, C)$, with $C = C^+$, is in P.

Proof: Let $N = (P, C)$ be a network such that $C = C^+$. Then we can solve Coherence as follows: Set $A = P$, and $R = \varnothing$ (or, alternatively, set $R = P$, and $A = \varnothing$). Since, $C = C^+$, we have $S_N(A, R) = C$ and thus $\mathrm{Coh}_N(A, R)$ is maximum (Note: This solves the optimization version of Coherence; to solve the decision version of coherence, we add the step that compares $\mathrm{Coh}_N(A, R)$ to c). This procedure runs in time $O(|P|)$. ■

Arguably, many interesting coherence problems that arise in practice contain at least some negative constraints, and thus Observation 5.1 seems to be a trivial special case of Coherence. Note, however, that there also exist special cases of Coherence in which the network may contain (possible even many) negative constraints that are also in P. To illustrate consider again Figure 5.1.

The network depicted in Figure 5.1 has the special property that all its constraints can be satisfied simultaneously. In general, let us call a network $N = (P, C)$ *consistent* if it is possible to satisfy all constraints in N; in other words, N is consistent if there exists a partition of P into A and R such that $S_N(A, R) = C$. We note the following theorem.

Theorem 5.1. Coherence on consistent networks is in P.

Proof: (*Non-constructive*) Let a consistent network $N = (P, C)$ and positive integer c form an instance for Coherence. By the definition of 'consistent,' if $c \leq \sum_{(p,q)\in C} \mathrm{w}(p,q)$ the answer is "yes." On the other hand, if $c > \sum_{(p,q)\in C} \mathrm{w}(p,q)$ then the answer is "no." ■

The non-constructive proof of Theorem 5.1 is not very helpful, since typically we would like to know how to partition P in order to obtain a partition of coherence c, for all $c \leq \sum_{(p,q)\in C} \mathrm{w}(p,q)$. In the following I present a constructive argument.

The argument is organized as follows. First, we describe a polynomial-time algorithm called Consistent Coherence algorithm that takes as input a connected network and outputs a partition. Second, we show that the Consistent Coherence algorithm always computes an optimal partition on consistent connected networks (Lemma 5.2). Finally, we conclude a constructive proof of Theorem 5.1.

Consistent Coherence algorithm [51]

Input: A connected network $N = (P, C)$.

Output: A partition $(A \cup R)^P$.

[Description of algorithm:] The Consistent Coherence algorithm performs a breath-first search (BFS) on the input network N. The BFS search considers each element in N in a particular order as follows: It starts by (0) considering an arbitrary element $p \in N$; then (1) it considers each neighbor of p; then (2) for each neighbor q of p, it considers the neighbors of q that have not yet been considered; then (3) for each neighbor r of each neighbor q of p, it considers the neighbors of r have not yet been considered; and so on, … (4) …, (5) …, (j) …, until in (k) all elements in P are considered.[52]

Note that this BFS search can be thought of as a traversal of the network that builds a spanning tree T of N of depth k. Further, we can see T as a search tree with each node of T being labeled by an element in P. Since each element in P and each constraint in C need not be considered more than a constant number of times, the traversal can be done in time $O(|P| + |C|)$, or $O(|P|^2)$.

The Consistent Coherence algorithm assigns elements to A or R in the order in which they are considered by the BFS procedure. It does so as follows. Let s denote the element whose neighbors are presently being considered by the BFS procedure. The algorithm checks whether s is assigned to A or to R; then for each neighbor t of s the algorithm checks the type of the constraint (s, t) and assigns t to A or R so as to satisfy constraint (i.e., if $s \in A$ and $(s, t) \in C^+$, then t is assigned to A; if $s \in A$ and $(s, t) \in C^-$, then t is assigned to R; if $s \in R$ and $(s, t) \in$

[51] I do not give a description of the algorithm in pseudo code because I think the procedure is easier to understand when described as I have done here. Furthermore, pseudo code for Breath First Search (BFS) can be found in many standard computer science textbooks (see e.g. Goodrich & Tamassia, 2002)

[52] Consider, for example, the graph in Figure A2 in the Appendix A. A BFS search starting at, say, vertex a, would first (0) consider vertex a, then (1) consider vertices b, c, d and e, then (2) vertices f, h, and i; then (3) vertices g and j; then (4) k, and finally (5) vertex l.

C^+, then t is assigned to R; and if $s \in R$ and $(s, t) \in C^-$, then t is assigned to A). Since this assignment procedure runs in $O(1)$ per assigned element t, we conclude a total running time of $O(|P|^2)$ for the Consistent Coherence algorithm. [End of Description]

The following lemma establishes that, if the input N is a *consistent* network, then Consistent Coherence algorithm outputs a *maximum* coherence partition for N. In other words, we show that the Consistent Coherence algorithm solves the optimization version of Coherence for consistent connected networks.

Lemma 5.2. Given a consistent connected network $N = (P, C)$ as input, the Consistent Coherence algorithm outputs a partition of P into sets A and R such that $\mathrm{Coh}_N(A, R)$ is maximum.

Proof: Let $N = (P, C)$ be a consistent connected network, and let partition $(A \cup R)^P$ be the partition obtained by running Consistent Coherence algorithm on N. Further, let $T = (P, C_T)$ denote the spanning tree of N that was traced by the run of the Consistent Coherence algorithm on N. We show that $\mathrm{S}_N(A, R) = C$.

First, we observe that all constraints in $C_T \subseteq C$ are satisfied; this is ensured by the assignment procedure run during the BFS search. It remains to be shown that also all other constraints are satisfied. We prove this by contradiction. Let p be the first element in P considered by the algorithm—i.e., p is the root of T. Further, let $(q, r) \in C \backslash C_T$ be a constraint in $C \backslash C_T$ that is not satisfied. Consider the cycle $X \subseteq N$ with $X = <(p, q_k), (q_k, q_{k-1}), \ldots, (q_3, q_2), (q_2, q_1), (q_1, q), (q, r), (r, r_1), (r_1, r_2), (r_2, r_3), \ldots, (r_{j-1}, r_j), (r_j, p)>$ such that all edges in the cycle, other than (q, r), are in C_T. We now prove that it is not possible to satisfy all constraints in X. Imagine traversing the edges in X in order, starting at r and ending at q, and while we visit the elements $r, r_1, r_2, r_3, \ldots, r_{j-1}, r_j, p, q_k, q_{k-1}, \ldots, q_3, q_2, q_1, q$, we assign them to A or R as follows: We start by assigning r to either A or R (it does not matter for the argument which is the case). Every time we visit an element s_i, with preceding element s_{i-1}, such that $(s_{i-1}, s_i) \in C^+$ we do one of the following: (1) if $s_{i-1} \in A$ then we assign s_i to A, (2) if $s_{i-1} \in R$ then we assign s_i to R. Every time we visit an element s_i such that $(s_{i-1}, s_i) \in C^-$ we do one of the following: (3) if $s_{i-1} \in A$ then we assign s_i to R, (4) if $s_{i-1} \in R$ then we assign s_i to A. Note that rules (1) – (4) are necessary to ensure that each traversed edge in $\{(r, r_1), (r_1, r_2), (r_2, r_3), \ldots, (r_{j-1}, r_j), (r_j, p)\ (p, q_k)\ (q_k, q_{k-1}), \ldots, (q_3, q_2), (q_2, q_1), (q_1, q)\}$ is satisfied by the assignment. The last edge (q, r) is satisfied if only if it was satisfied by the original partition made by the Consistent Coherence algorithm. We conclude that the cycle X is *not* consistent. Since $X \subseteq N$, we also conclude that N is *not* consistent, contradicting the fact that N is consistent. ■

We can now derive a constructive proof of Theorem 5.1.

Proof of Theorem 5.1: (*Constructive*) Let $N = (P, C)$ be a consistent network, and let $N_i = (P_i, C_i)$, with $i = 1, 2, \ldots, x$, be the components of N. Then we solve Coherence for N, by running the Consistent Coherence algorithm on each component N_i. This returns for each component N_i an optimal partition $(A_i \cup R_i)^{P_i}$. We define $A = A_1 \cup A_2 \cup \ldots \cup A_x$ and $R = R_1 \cup R_2 \cup \ldots \cup R_x$, and conclude that $\mathrm{Coh}_N(A, R)$ is maximum.

For each component the procedure runs in time $O(|P_i|^2)$, and thus the whole procedure runs in time $O(|P_1|^2 + |P_2|^2 + \ldots + |P_x|^2)$. Since $|P_1| + |P_2| + \ldots + |P_x| = |P|$, we conclude a total running time of $O(|P|^2)$. ■

Interestingly, Theorem 5.1 implies that, on Thagard and Verbeurgt's definition of Coherence, coherence reasoning is difficult only if one's belief system is *inconsistent*. As long as one maintains a consistent belief base, coherence reasoning is tractable.

I close this subsection with a couple of related observations:

Lemma 5.3. Coherence on an instance (N, c) with $c = \sum_{(p,q)\in C} \mathrm{w}(p,q)$ is in P.[53]

Proof: Let (N, c), with $c = \sum_{(p,q)\in C} \mathrm{w}(p,q)$, be an instance for Coherence. We run Consistent Coherence algorithm on N as described in the constructive proof of Theorem 5.1. Let the resulting partition be $(A \cup R)^P$. If $\mathrm{Coh}_N(A, R) = \sum_{(p,q)\in C} \mathrm{w}(p,q)$ then (N, c) is a yes-instance, otherwise it is a no-instance. ■

From Lemma 5.3 we also conclude that we can decide in polynomial-time whether or not a network N is consistent.

Corollary 5.4. Checking whether or not a network is consistent is in P.

Lemma 5.4. If network $N = (P, C)$ is a tree, then N is consistent.

[53] Lemma 5.3 also follows from a straightforward reduction from Coherence on consistent networks to the polynomial-time problem 2-SAT (see Appendix B for problem definition).

Proof: (*Sketch*). If we run Consistent Coherence algorithm on a tree $N = (P, C)$, the algorithm will always trace, and thus satisfy, every constraint in C. ■

Since a tree on $|P|$ vertices has exactly $|P| - 1$ edges (Gross & Yellen, 1999), we also conclude the following corollary.

Corollary 5.5. A network $N = (P, C)$ that is a tree has coherence $c = |P|-1$. Lemma 5.4 and Corollary 5.5 will also prove useful in Section 5.6.

5.4.2. Generalizations of Coherence

Recall that in the application of Coherence to scientific reasoning, a distinction was made between elements representing data (D) and elements representing scientific hypotheses (H) (cf. the applications to legal justification and social judgment). In this context, Thagard (2000) adopts the *data priority principle*. This principle states that elements in D are "favored" to be in the set A. Thagard (2000) proposes that this principle can work in at least two different ways: Either loosely, by assigning weights to elements in D and for each $d \in D$ counting its weight towards the total coherence of a partition only if $d \in A$; or strictly, by requiring that all $d \in D$ be assigned to A. The first version is called Discriminating Coherence and the second is called Foundational Coherence.

Discriminating Coherence (*decision version*)

Input: A network $N = (P, C)$, with $(H \cup D)^P$ and $(C^+ \cup C^-)^C$. For each $d \in D$ there is an associated positive integer weight $\mathrm{w}_D(d)$, and for each $(p, q) \in C$ there is an associated positive integer weight $\mathrm{w}_C(p, q)$. A positive integer c.

Question: Does there exist a partition of P into A and R such that $\mathrm{DCoh}_N(A, R) \geq c$? Here $\mathrm{DCoh}_N(A, R) = \sum_{(p,q)\in \mathrm{S_N}(A,R)} \mathrm{w}_C(p,q) + \sum_{d\in(D\cap A)} \mathrm{w}_D(d)$.

Foundational Coherence (*decision version*)

Input: A network $N = (P, C)$, with $(H \cup D)^P$ and $(C^+ \cup C^-)^C$. For each $(p, q) \in C$ there is an associated positive integer weight $\mathrm{w}(p, q)$. A positive integer c.

Question: Does there exist a partition of P into A and R such that $D \subseteq A$ and $\mathrm{Coh}_N(A, R) \geq c$?

Does the fact that Coherence is NP-hard automatically imply that Discriminating Coherence and Foundational Coherence are NP-hard? Yes! Each of these problems encompassed Coherence as a special case. Namely, Coherence is equivalent to Discriminating Coherence on inputs with $D = \varnothing$. Also, Coherence is equivalent to Foundational Coherence on inputs with $D = \varnothing$. In other words, Discriminating Coherence and Foundational Coherence are generalizations of Coherence, and thus they are "at least as hard" as Coherence. We conclude:

Corollary 5.6. Discriminating Coherence is NP-hard

Corollary 5.7. Foundational Coherence is NP-hard

The fact that Discriminating Coherence and Foundational Coherence are generalizations of Coherence means that the results discussed in Section 5.4.1 do not necessarily apply to these more general problems (though in some cases they might). It also means that fpt-results, in Sections 5.5 – 5.7, obtained for Coherence do not necessarily generalize to comparable parameterizations for Discriminating Coherence and Foundational Coherence (though, again, in some cases they might).

5.4.3. Variations on Coherence

Thagard (2000) not only considers special cases and generalizations of Coherence, but also variations on the problem (see also Thagard, 1989; Thagard and Verbeurgt, 1998). Such variations, although seemingly related to Coherence, may have very different properties, and hence very different complexity, than the Coherence problem itself.

To illustrate I consider the application of Coherence in the domain of legal justification. Here a jury member wishes to decide whether the belief that the defendant is innocent coheres at least as much with the evidence as does the belief that the defendant is guilty.[54] These competing beliefs can be modeled as follows (cf. Thagard, 1998, 2000; Thagard and Verbeurgt, 1998): There is a special element in the network, denoted s, representing the hypothesis that the defendant is innocent. We interpret $s \in A$ as the

[54] Note that we are giving the defendant the benefit of the doubt if both propositions cohere equally with the evidence. One might argue that in jury trials the verdict "guilty" is justified only if the defendant is found guilty beyond reasonable doubt. Of course, this aspect of jury decision-making can be incorporated into the model presented here by defining a critical amount λ that represents "reasonable doubt." Then the jury member's task is to decide whether the belief 'the defendant is innocent' is at most λ less coherent than the belief 'the defendant is guilty.' Note that the model we work with in this section is a special case of this problem with $\lambda = 0$.

belief "the defendant is innocent" and $s \in R$ as the belief "the defendant is guilty." If we again adopt the data priority principle, either in a strict or loose sense, we obtain the following two problems:

Single-Element Discriminating Coherence (*decision version*)
Input: A network $N = (P, C)$, with $(H \cup D)^P$ and $(C^+ \cup C^-)^C$. For each $d \in D$ there is an associated positive integer weight $w_D(d)$, and for each $(p, q) \in C$ there is an associated positive integer weight $w_C(p, q)$. A special element $s \in H$ and a positive integer c.
Question: Does there exist a partition of P into A and R such that $\text{DCoh}_N(A, R)$ is maximum and $s \in A$?

Single-Element Foundational Coherence (*decision version*)
Input: A network $N = (P, C)$, with $(H \cup D)^P$ and $(C^+ \cup C^-)^C$. For each $(p, q) \in C$ there is an associated positive integer weight $w(p, q)$. A special element $s \in H$ and a positive integer c.
Question: Does there exist a partition of P into A and R such that $\text{Coh}_N(A, R)$ is maximum and $D \cup s \subseteq A$?

Note that both problems are decision problems.

Does the fact that Coherence is NP-hard automatically imply that Single-Element Discriminating Coherence and Single-Element Foundational Coherence are NP-hard? No! Coherence is not a special case of any of these problems.

Let us consider Single-Element Discriminating Coherence first. Despite the apparent similarity between Single-Element Discriminating Coherence and Discriminating Coherence (page 90), the two problems are of quite different flavor. In Discriminating Coherence the goal is to find a partition of at least coherence c, while in Single-Element Discriminating Coherence we want to know if there exists an optimal partition for N such that $s \in A$. Hence, in Single-Element Discriminating Coherence we are not interested in finding a partition, nor in how much coherence it might have; all we care about is whether or not we should accept s.

The difference between Discriminating Coherence and Single-Element Discriminating Coherence also becomes apparent when we consider the special case of

Single-Element Discriminating Coherence with $D = \varnothing$. Recall that if we set $D = \varnothing$, then Coherence and Discriminating Coherence are equivalent, and thus Discriminating Coherence is NP-hard even if $D = \varnothing$. On the other hand, if we set $D = \varnothing$, then solving Single-Element Discriminating Coherence becomes trivial. In that case, the answer is always "yes," because there always exist at least two maximum coherence partitions, one with $s \in A$ and one with $s \in R$, and thus the answer is always "yes." We conclude this from the fact that the function $\text{Coh}_N(.,.)$ is symmetric, as shown in Observation 5.2.

Observation 5.2. Let $N = (P, C)$ be a network. Then for any partition $(A \cup R)^P$, and its complement $(A' \cup R')^P$, with $A' = R$ and $R' = A$, $\text{Coh}_N(A, R) = \text{Coh}_N(A', R')$.

Proof: The claim follows directly from the definition of the coherence function $\text{Coh}_N(A, R)$. Namely, a constraint is satisfied (and counts towards coherence) if both endpoints are in the same set of the partition (regardless of whether they are both in R or both in A), and a negative constraint is satisfied if both endpoints are in opposite sets of the partition (it does not matter which of the two is in A and which is in R). Thus, a partition $(A \cup R)^P$, and its complement $(A' \cup R')^P$ satisfy exactly the same set of constraints. ■

Corollary 5.8. Single-Element Discriminating Coherence with $D = \varnothing$ is in P.
Note that all observations made so far for Single-Element Discriminating Coherence also apply for Single-Element Foundational Coherence. That is, like Single-Element Discriminating Coherence, Single-Element Foundational Coherence is not a special case of Coherence. Further, for $D = \varnothing$, Single-Element Discriminating Coherence and Single-Element Foundational Coherence are the same problem. Hence we also have:

Corollary 5.9. Single-Element Foundational Coherence with $D = \varnothing$ is in P.
Are Single-Element Discriminating Coherence and Single-Element Foundational Coherence computationally easier to compute than Coherence? At present time, I do not know the answer to this question. And *this* is the point of my illustration. Thagard cannot conclude that problems such as Single-Element Discriminating Coherence and Single-Element Foundation Coherence are NP-Hard simply because Coherence is NP-hard. Of course these problems may very well be NP-hard, but knowing so requires a proof.

5.5. c-Coherence is in FPT

In this section we consider the parameterized complexity of Coherence when parameterized by integer c. We show that c-Coherence is in FPT. The argument is build up as follows. Section 5.5.1 introduces a generalization of the problem Coherence that allows for more than one constraint between two elements in P. This generalization, called Double-Constraint Coherence, is then used to formulate a set of reduction rules in Section 5.5.2. After the application of the reduction rules we know that the input network is of minimum degree 3. In Section 5.5.3, we show that for each element in P we can satisfy at least half of its incident constraints, and conclude that every network on n or more elements has at least c coherence. This allows us to conclude a kernel of $|P| \leq c$ and thus a running time $O(2^c + |P|)$.

I remark that the fpt-algorithm presented in this section is not constructive—i.e., it solves the decision version but not the search version of Coherence. A constructive, but slower fpt-algorithm for c-Coherence will be discussed in Section 5.6.

5.5.1. Double-Constraint Coherence

In Coherence, for each pair $p, q \in P$ there is at most one constraint $(p, q) \in C$, with either $(p, q) \in C^+$ or $(p, q) \in C^-$. In the following generalization of Coherence, called Double-Constraint Coherence, for each pair $p, q \in P$ there may be two constraints; a positive constraint $(p, q)^+ \in C^+$ and a negative constraint $(p, q)^- \in C^-$.

Double-Constraint Coherence

Input: A double-constraint network $N = (P, C^+ \cup C^-)$. For each $(p, q)^+ \in C^+$ there is an associated positive integer weight $w(p, q)^+$, and for each $(p, q)^- \in C^-$ there is an associated positive integer weight $w(p, q)^-$. A positive integer c.

Question: Does there exist a partition $(A \cup R)^P$ such that $\mathrm{Coh}_N(A, R) \geq c$?

Note that Coherence is a special case of Double-Constraint Coherence. Hence, an fpt-result for c-Double-Constraint Coherence also applies to c-Coherence.

In the following, if for two elements $p, q \in P$ there exists exactly one constraint $(p, q) \in C^+ \cup C^-$ then we may write (p, q) instead of $(p, q)^+$ or $(p, q)^-$. As for networks, the *degree* of an element p in a double-constraint network N, $\deg_N(p)$, is the number of constraints incident to p, and the *neighborhood* of an element p in a double-constraint network N, $\mathrm{N}_N(p)$, is the set of neighbors of p in N. Note that in a double-constraint network $\deg_N(p)$ may be larger than $|\mathrm{N}_N(p)|$. Further, for two vertices p and q in a double-constraint network we say p and q are *independent* if $(p, q)^+ \notin C^+$ and $(p, q)^- \notin C^-$.

5.5.2. Reduction Rules

We present a set of reduction rules for Double-Constraint Coherence. We only apply a rule if none of the preceding rules applies. Rules (DC 2) – (DC 9) are illustrated in Figure 5.2.

We start by removing small components from the input with rule (DC 1) as follows: For each component $N_i = (P_i, C_i^+ \cup C_i^-)$, with $|P_i| \leq 3$, we compute a partition of P_i with maximum coherence using an exhaustive search: i.e., we compute all $2^{|P_i|}$ possible partitions of P_i, compute their coherence, and pick the one with maximum coherence. Since rule (DC 1) is only applied if $|P_i| \leq 3$, this rule runs in time $O(2^{|P_i|}) = O(2^3) = O(8) = O(1)$, which is constant time.

(DC 1) Small Component Rule: Let the double-constraint network $N = (P, C^+ \cup C^-)$ and positive integer c form an instance for Double-Constraint Coherence. If N has a component $N_i = (P_i, C_i^+ \cup C_i^-)$ with $|P_i| \leq 3$, then we determine a partition $(A_i \cup R_i)^P$ such that $\mathrm{Coh}_N(A_i, R_i)$ is maximum. Then let $N^* = N \backslash N_i$ and let $c^* = c - \mathrm{Coh}_N(A_i, R_i)$.

Proof: Since N_i is a component of N it is not connected to any element outside P_i, and thus we can determine a maximum coherence partition of P_i independent of the maximum coherence partition of $P \backslash P_i$. ■

Rules (DC 2) and (DC 3) eliminate elements that are connected to at most one other element.

(DC 2) Pendant Rule: Let double-constraint network $N = (P, C^+ \cup C^-)$ and positive integer c form an instance for Double-Constraint Coherence. If there exists an element $p \in P$ such that $\deg_N(p) = 1$, with $\mathrm{N}_N(p) = \{q\}$, then let $N^* = (P^*, C^{+*} \cup C^{-*})$, with $P^* = P \backslash \{p\}$, and $C^{+*} \cup C^{-*} = (C^+ \cup C^-) \backslash \{(p, q)\}$, and $c^* = c - \mathrm{w}(p, q)$. Finally, let (N^*, c^*) be the new instance for Double-Constraint Coherence.

Proof: We show (N, c) is a yes-instance for Double-Constraint Coherence if and only if (N^*, c^*) is a yes-instance for Double-Constraint Coherence, by showing that constraint (p, q) is satisfied by an optimal partition. The proof is by contradiction.

Assume that $(A \cup R)^P$ is partition such that $\mathrm{Coh}_N(A, R)$ is maximum and $(p, q) \notin \mathrm{S}_N(A, R)$. We distinguish two cases: (1) Let $p \in A$. We set $A' = A \backslash \{p\}$ and $R' = R \cup \{p\}$. Since (p, q) is the only constraint incident to p, we conclude that $\mathrm{S}_N(A', R') = \mathrm{S}_N(A, R) \cup \{(p, q)\}$. But that means that $\mathrm{Coh}_N(A', R') > \mathrm{Coh}_N(A, R)$, contradicting that $\mathrm{Coh}_N(A, R)$ is maximum. (2) Let $p \in R$. We set $A' = A \cup \{p\}$ and $R' = R \backslash \{p\}$. Since (p, q) is the only constraint incident to p, we conclude that $\mathrm{S}_N(A', R') = \mathrm{S}_N(A, R) \cup \{(p, q)\}$. But that means that $\mathrm{Coh}_N(A', R') > \mathrm{Coh}_N(A, R)$, contradicting that $\mathrm{Coh}_N(A, R)$ is maximum. ■

(DC 3) Degree-2 Single Neighbor Rule: Let the double-constraint network $N = (P, C^+ \cup C^-)$ and positive integer c form an instance for Double-Constraint Coherence. Let element $p \in P$, with $\deg_N(p) = 2$, have one neighbor q, such that $(p, q)^+ \in C^+$ and $(p, q)^- \in C^-$. Then let $N^* = (P^*, C^{+*} \cup C^{-*})$, with $P^* = P \backslash \{p\}$, and $C^{+*} \cup C^{-*} = (C^+ \cup C^-) \backslash \{(p, q)^+, (p, q)^-\}$ and $c^* = c - \mathrm{MAX}(\mathrm{w}(p, q)^+, \mathrm{w}(p, q)^-)$. Finally, let (N^*, c^*) be the new instance for Double-Constraint Coherence.

Proof: Let $(A \cup R)^P$ be a partition such that $\mathrm{Coh}_N(A, R)$ is maximum. Then we know that either $(p, q)^+ \in \mathrm{S}_N(A, R)$ or $(p, q)^- \in \mathrm{S}_N(A, R)$. Since the partition has maximum coherence we can assume that the satisfied constraint in $\{(p, q)^+, (p, q)^-\}$ is the one with largest weight. Hence, (N, c) is a yes-instance for Double-Constraint Coherence if and only if (N^*, c^*) is a yes-instance for Double-Constraint Coherence. ■

Rules (DC 4) – (DC 9) deal with all elements p in N that have exactly two neighbors, q and r. Each of these rules has the following general form. The part of N that consists of p, q, r and their incident constraints, is replaced in N^* by q and r connected by a positive constraint $(q, r)^+$ and a negative constraint $(q, r)^-$. Further, the weights in N^* on $(q, r)^+$ and $(q, r)^-$ are set so as to exactly capture the coherence value obtained in a optimal partition $(A \cup R)^P$ if (1) both q and r would be in the same set (both in A or both in R); captured by the weight $(q, r)^+$ and if (2) q and r would be in different sets of the partition (one in A and the other in R); captured by the weight $(q, r)^-$.

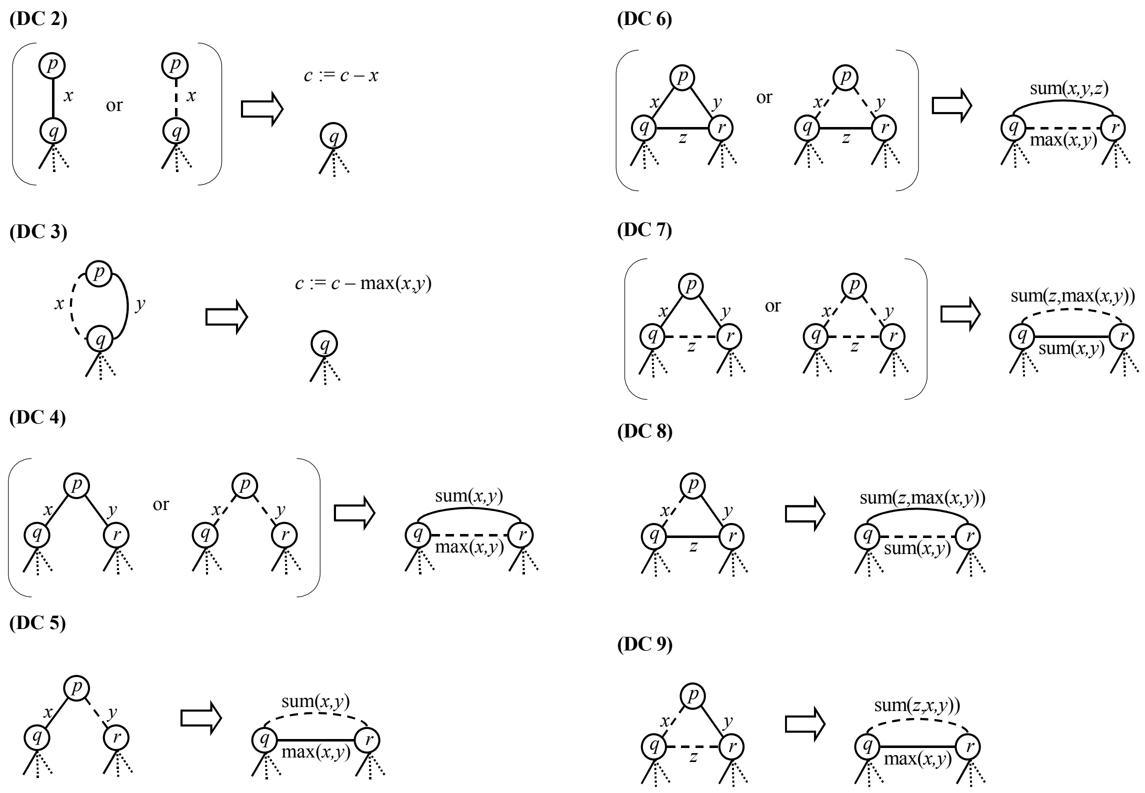


Figure 5.2. Illustration of the reduction rules for Double-Constraint Coherence. Reduction rules (DC 2)–(DC 9) are illustrated. Solid lines represent positive constraints and dotted lines represented negative constraints. The symbols *x*, *y* and *z* represent weights. See text for details.

(DC 4) First Degree-2 Disconnected Neighbor Rule: Let double-constraint network $N = (P, C^+ \cup C^-)$ and positive integer c form an instance for Double-Constraint Coherence. Let element $p \in P$, with $\deg_N(p) = 2$, have two neighbors q and r such that q and r are independent and $[(p, q), (p, r) \in C^+$ or $(p, q), (p, r) \in C^-]$. Then let $N^* = (P^*, C^{+*} \cup C^{-*})$, with $P^* = P - p$, $C^{+*} = (C^+\backslash\{(p, q), (p, r)\}) \cup (q, r)^+$, and $C^{-*} = (C^- \backslash\{(p, q), (p, r)\}) \cup (q, r)^-$. Further, we set the weights $\mathrm{w}^*(q, r)^+ = \mathrm{w}(p, q) + \mathrm{w}(p, r)$ and $\mathrm{w}^*(q, r)^- = \mathrm{MAX}(\mathrm{w}(p, q), \mathrm{w}(p, r))$. Finally, let (N^*, c^*), with $c^* = c$, be the new instance for Double-Constraint Coherence.

Proof: Let $(A \cup R)^P$ be a partition such that $\mathrm{Coh}_N(A, R)$ is maximum. We distinguish two cases: (1) Let q, r both be in the same set of the partition (i.e., either $q, r \in A$ or $q, r \in R$). Since (p, q) and (p, r) are the same type of constraint (both positive or both negative), setting $p \in A$, or setting $p \in R$, will either satisfy both (p, q) and (p, r), or neither of them. Since $\text{Coh}_N(A, R)$ is maximum we conclude it must be the first option, and thus $\{(p, q), (p, r)\} \subseteq \text{S}_N(A, R)$. (2) Let q and r be in different sets of the partition (i.e., either $q \in A$ and $r \in R$, or $q \in R$ and $r \in A$). Since (p, q) and (p, r) are the same type of constraint (both positive or both negative), setting $p \in A$, or setting $p \in R$, can satisfy at most one of (p, q) and (p, r). Since $\text{Coh}_N(A, R)$ is maximum we conclude the satisfied constraint is the one with the largest weight. We conclude that (N, c) is a yes-instance for Double-Constraint Coherence if and only if (N^*, c^*) is a yes-instance for Double-Constraint Coherence. ■

(DC 5) Second Degree-2 Disconnected Neighbors Rule: Let double-constraint network $N = (P, C^+ \cup C^-)$ and positive integer c form an instance for Double-Constraint Coherence. Let $p \in P$, with $\text{deg}_N(p) = 2$, be an element with exactly two neighbors q, r that are independent, $(p, q) \in C^+$ and $(p, r) \in C^-$. Then let $N^* = (P^*, C^{+*} \cup C^{-*})$, with $P^* = P \backslash \{p\}$, $C^{+*} = (C^+ \backslash \{(p, q), (p, r)\}) \cup \{(q, r)^+\}$, and $C^{-*} = (C^- \backslash \{(p, q), (p, r)\}) \cup \{(q, r)^-\}$. Further, we set the weights $\text{w}^*(q, r)^+ = \text{MAX}(\text{w}(p, q), \text{w}(p, r))$ and $\text{w}^*(q, r)^- = \text{w}(p, q) + \text{w}(p, r)$. Finally, let (N^*, c^*), with $c^* = c$, be the new instance for Double-Constraint Coherence.

Proof: Let $(A \cup R)^P$ be a partition such that $\text{Coh}_N(A, R)$ is maximum. We distinguish two cases: (1) Let q, r both be in the same set of the partition (i.e., either $q, r \in A$ or $q, r \in R$). Since (p, q) and (p, r) are different types of constraint (one is positive and the other negative), setting $p \in A$, or setting $p \in R$, can satisfy at most one of (p, q) and (p, r). Since $\text{Coh}_N(A, R)$ is maximum we conclude that the satisfied constraint is the one with the largest weight. (2) Let q and r be in different sets of the partition (i.e., either $q \in A$ and $r \in R$, or $q \in R$ and $r \in A$). Since (p, q) and (p, r) are different types of constraint (one is positive and the other negative), setting $p \in A$, or setting $p \in R$, will either satisfy both (p, q) and (p, r), or neither of them. Since $\text{Coh}_N(A, R)$ is maximum we conclude it must be the first option, and thus $\{(p, q), (p, r)\} \subseteq \text{S}_N(A, R)$. We conclude that (N, c) is a yes-instance for Double-Constraint Coherence if and only if (N^*, c^*) is a yes-instance for Double-Constraint Coherence. ■

(DC 6) First Degree-2 Connected Neighbors Rule: Let double-constraint network $N = (P, C^+ \cup C^-)$ and positive integer c form an instance for Double-Constraint

Coherence. Let element $p \in P$, with $\deg_N(p) = 2$, have two neighbors q and r such that $(q, r)^+ \in C^+$ and $(q, r)^- \notin C^-$, and $[(p, q), (p, r) \in C^+$ or $(p, q), (p, r) \in C^-]$. Then let $N^* = (P^*, C^{+*} \cup C^{-*})$, with $P^* = P\backslash\{p\}$, $C^{+*} = C^+\backslash\{(p, q)^+, (p, r)^+\}$, and $C^{-*} = (C^- \backslash\{(p, q)^-, (p, r)^-\}) \cup \{(q, r)^-\}$. Further, we set the weights $w^*(q, r)^+ = w(p, q) + w(p, r) + w(q, r)$ and $w^*(q, r)^- = \text{MAX}(w(p, q), w(p, r))$. Finally, let (N^*, c^*), with $c^* = c$, be the new instance for Double-Constraint Coherence.

Proof: Let $(A \cup R)^P$ be a partition such that $\text{Coh}_N(A, R)$ is maximum. We distinguish two cases: (1) Let q, r be in the same set of the partition (i.e., either $q, r \in A$ or $q, r \in R$). Then $(q, r) \in S_N(A, R)$. Further, since (p, q) and (p, r) are the same type of constraint (both positive or both negative), setting $p \in A$, or setting $p \in R$, will either satisfy both (p, q) and (p, r), or neither of them. Since $\text{Coh}_N(A, R)$ is maximum we conclude it must be the first option. In sum, we have $\{(q, r), (p, q), (p, r)\} \subseteq S_N(A, R)$. (2) Let q and r be in different sets of the partition (i.e., either $q \in A$ and $r \in R$, or $q \in R$ and $r \in A$). Then $(q, r) \notin S_N(A, R)$. Further, since (p, q) and (p, r) are the same type of constraint (both positive or both negative), setting $p \in A$, or setting $p \in R$, can satisfy at most one of (p, q) and (p, r). Since $\text{Coh}_N(A, R)$ is maximum we conclude the satisfied constraint is the one with the largest weight. We conclude that (N, c) is a yes-instance for Double-Constraint Coherence if and only if (N^*, c^*) is a yes-instance for Double-Constraint Coherence. ■

(DC 7) Second Degree-2 Connected Neighbors Rule: Let double-constraint network $N = (P, C^+ \cup C^-)$ and positive integer c form an instance for Double-Constraint Coherence. Let element $p \in P$, with $\deg_N(p) = 2$, have two neighbors q and r such that $(q, r)^- \in C^-$ and $(q, r)^+ \notin C^+$, and $[(p, q), (p, r) \in C^+$ or $(p, q), (p, r) \in C^-]$. Then let $N^* = (P^*, C^{+*} \cup C^{-*})$, with $P^* = P\backslash\{p\}$, $C^{+*} = (C^+\backslash\{(p, q)^+, (p, r)^+\}) \cup \{(q, r)^+\}$, and $C^{-*} = C^- \backslash\{(p, q)^-, (p, r)^-\}$. Further, we set the weights $w^*(q, r)^+ = w(p, q) + w(p, r)$ and $w^*(q, r)^- = w(q, r) + \text{MAX}(w(p, q), w(p, r))$. Finally, let (N^*, c^*), with $c^* = c$, be the new instance for Double-Constraint Coherence.

Proof: Let $(A \cup R)^P$ be a partition such that $\text{Coh}_N(A, R)$ is maximum. We distinguish two cases: (1) Let q, r be in the same set of the partition (i.e., either $q, r \in A$ or $q, r \in R$). Then $(q, r) \notin S_N(A, R)$. Further, since (p, q) and (p, r) are the same type of constraint (both positive or both negative), setting $p \in A$, or setting $p \in R$, will either satisfy both (p, q) and (p, r), or neither of them. Since $\mathrm{Coh}_N(A, R)$ is maximum we conclude it must be the first option, and thus $\{(p, q), (p, r)\} \subseteq \mathrm{S}_N(A, R)$. (2) Let q and r be in different sets of the partition (i.e., either $q \in A$ and $r \in R$, or $q \in R$ and $r \in A$). Then $(q, r) \in \mathrm{S}_N(A, R)$. Since (p, q) and (p, r) are the same type of constraint (both positive or both negative), setting $p \in A$, or setting $p \in R$, can satisfy at most one of (p, q) and (p, r). Since $\mathrm{Coh}_N(A, R)$ is maximum we conclude the satisfied constraint is the one with largest weight. We conclude that (N, c) is a yes-instance for Double-Constraint Coherence if and only if (N^*, c^*) is a yes-instance for Double-Constraint Coherence. ■

(DC 8) Third Degree-2 Connected Neighbors Rule: Let double-constraint network $N = (P, C^+ \cup C^-)$ and positive integer c form an instance for Double-Constraint Coherence. Let element $p \in P$, with $\deg_N(p) = 2$, have two neighbors q and r such that $(q, r)^+ \in C^+$ and $(q, r)^- \notin C^-$, and $(p, r)^+ \in C^+$ and $(p, q)^- \in C^-$. Then let $N^* = (P^*, C^{+*} \cup C^{-*})$, with $P^* = P\backslash\{p\}$, $C^{+*} = C^+\backslash\{(p, r)^+\}$, and $C^{-*} = (C^- \backslash\{(p, q)^-\}) \cup \{(q, r)^-\}$. Further, we set the weights $w^*(q, r)^+ = w(q, r)^+ + \mathrm{MAX}(w(p, q)^-, w(p, r)^+)$, and $w^*(q, r)^- = w(p, q)^- + w(p, r)^+$. Finally, let (N^*, c^*), with $c^* = c$, be the new instance for Double-Constraint Coherence.

Proof: Let $(A \cup R)^P$ be a partition such that $\mathrm{Coh}_N(A, R)$ is maximum. We distinguish two cases: (1) Let q and r be in the same set of the partition (i.e., either $q, r \in A$ or $q, r \in R$). Then $(q, r) \in \mathrm{S}_N(A, R)$. Further, since (p, q) and (p, r) are different types of constraint (one is positive and the other negative), setting $p \in A$, or setting $p \in R$, can satisfy at most one of (p, q) and (p, r). Since $\mathrm{Coh}_N(A, R)$ is maximum we conclude that the satisfied constraint is the one with the largest weight. (2) Let q and r be in different sets of the partition (i.e., either $q \in A$ and $r \in R$, or $q \in R$ and $r \in A$). Then $(q, r) \notin \mathrm{S}_N(A, R)$. Since (p, q) and (p, r) are different types of constraint (one is positive and the other negative), setting $p \in A$, or setting $p \in R$, will either satisfy both (p, q) and (p, r), or neither of them. Since $\mathrm{Coh}_N(A, R)$ is maximum we conclude it must be the first option, and thus $\{(p, q), (p, r)\} \subseteq \mathrm{S}_N(A, R)$. We conclude that (N, c) is a yes-instance for Double-Constraint Coherence if and only if (N^*, c^*) is a yes-instance for Double-Constraint Coherence. ■

(DC 9) Third Degree-2 Connected Neighbors Rule: Let double-constraint network $N = (P, C^+ \cup C^-)$ and positive integer c form an instance for Double-Constraint Coherence. Let element $p \in P$, with $\deg_N(p) = 2$, have two neighbors q and r such that $(q, r)^- \in C^-$ and $(q, r)^+ \notin C^+$, and $(p, r)^+ \in C^+$ and $(p, q)^- \in C^-$. Then let $N^* = (P^*, C^{+*} \cup C^{-*})$, with $P^* = P \backslash \{p\}$, $C^{+*} = (C^+ \backslash \{(p, r)^+\}) \cup \{(q, r)^+\}$, and $C^{-*} = C^- \backslash \{(p, q)^-\}$. Further, we set the weights $w^*(q, r)^+ = \text{MAX}(w(p, q)^-, w(p, r)^+)$, and $w^*(q, r)^- = w(q, r)^- + w(p, q)^- + w(p, r)^+$. Finally, let (N^*, c^*), with $c^* = c$, be the new instance for Double-Constraint Coherence.

Proof: Let $(A \cup R)^P$ be a partition such that $\text{Coh}_N(A, R)$ is maximum. We distinguish two cases: (1) Let q and r be in the same set of the partition (i.e., either $q, r \in A$ or $q, r \in R$). Then $(q, r)^- \notin S_N(A, R)$. Further, since (p, q) and (p, r) are different types of constraint (one is positive and the other negative), setting $p \in A$, or setting $p \in R$, can satisfy at most one of (p, q) and (p, r). Since $\text{Coh}_N(A, R)$ is maximum we conclude that the satisfied constraint is the one with the largest weight. (2) Let q and r be in different sets of the partition (i.e., either $q \in A$ and $r \in R$, or $q \in R$ and $r \in A$). Then $(q, r) \in S_N(A, R)$. Since (p, q) and (p, r) are different types of constraint (one is positive and the other negative), setting $p \in A$, or setting $p \in R$, will either satisfy both (p, q) and (p, r), or neither of them. Since $\text{Coh}_N(A, R)$ is maximum we conclude it must be the first option, and thus $\{(q, r), (p, q), (p, r)\} \subseteq S_N(A, R)$. We conclude that (N, c) is a yes-instance for Double-Constraint Coherence if and only if (N^*, c^*) is a yes-instance for Double-Constraint Coherence. ■

When given an instance (N, c) for Double-Constraint Coherence, we can apply the polynomial-time reduction rules (DC 1) – (DC 9) until none of the rules applies anymore. If none of the rules (DC 1) – (DC 9) applies anymore, then we say that the resulting instance (N', c') is *reduced* for Double-Constraint Coherence. Note that we can reduce any instance for Double-Constraint Coherence in time $O(|P|)$. Namely, each reduction rule runs in constant time. Further, each rule causes an element to be deleted from P. Thus we never apply more than $|P|$ reduction rules.

5.5.3. A Problem Kernel

Let (N, c) be a reduced instance for Double-Constraint Coherence. Then we know that N is of minimum degree 3. Namely, the reduction rules (DC 1) – (DC 9) ensure that all vertices with degree 0, 1 or 2 are removed from the network. The following lemma uses this fact to infer a problem kernel for c-Double-Constraint Coherence.

Lemma 5.5. Let (N, c), with $N = (P, C^+ \cup C^-)$, be a reduced instance for Double-Constraint Coherence. If $|P| \geq c$ then (N, c) is a yes-instance Double-Constraint Coherence.

Proof: We show that there exists a partition such that for every element $p \in P$ at least half of the constraints incident to p are satisfied. Let $(A \cup R)^P$ be a partition that satisfies the most constraints; i.e., $|S_N(A, R)|$ is maximum. We now prove that for every $p \in P$ at least half of the constraints incident to p are in $S_N(A, R)$. The proof is by contradiction. Assume that there exists an element $p \in P$ such that strictly less than half of the constraints incident to p are in $S_N(A, R)$. We define a new partition $(A' \cup R')^P$ and we distinguish between two cases: (1) Let $p \in A$. Then let $A' = A \backslash \{p\}$ and $R' = R \cup \{p\}$; (2) Let $p \in R$. Then let $A' = A \cup \{p\}$ and $R' = R \backslash \{p\}$. Since, in both (1) and (2), partition $(A' \cup R')^P$ satisfies all constraints incident to p that are not satisfied by the partition $(A \cup R)^P$, we conclude that strictly more than half of the constraints incident to p are in $S_N(A', R')$. Further, the change from partition $(A \cup R)^P$ to $(A' \cup R')^P$ does not affect any of the other constraints in N. We conclude that the partition $|S_N(A', R')| > |S_N(A, R)|$, contradicting the fact that $|S_N(A, R)|$ is maximum.

We have shown that for each element $p \in P$ half of its incident constraints can be satisfied. Further, since the input is reduced, the minimum number of constraints incident to any element is at least 3. This means that for each element p at least 2 of its incident constraints can be satisfied.

We note that each constraint has two elements as its endpoints. Thus we conclude that for any reduced instance on $|P| = n$ elements there exists a partition $(A \cup R)^P$, such that $\mathrm{Coh}_N(A, R) \geq \frac{2n}{2} = n$. Thus, if (N, c) is a reduced instance for Double-Constraint Coherence and $c \leq n$, then the answer is "yes" for (N, c). ■

From (DC 1)–(DC 9) and Lemma 5.5 we conclude the following theorem.

Theorem 5.2. c-Double-Constraint Coherence is in FPT and solvable in time $O(2^c + |P|)$.

Proof: We describe an fpt-algorithm for c-Double-Constraint Coherence. The algorithm takes as input an instance (N, c) for c-Double-Constraint Coherence and first reduces it using (DC 1)–(DC 9). For the resulting reduced instance (N', c') for c-Double-Constraint Coherence we know that either $|P'| < c$ or $|P'| \geq c$. If $|P'| \geq c$ then, using Lemma 5.5, we conclude that (N, c) is a yes-instance for c-Double-Constraint Coherence; in which case the computation has terminated in time $O(n)$. If $|P'| < c$ then we perform an exhaustive search on all $2^{|P'|}$ possible partitions of P'. This search runs in $O(2^{|P'|})$ which is $O(2^c)$, since $|P'| < c$. The exhaustive search time, combined with the polynomial time to reduce the instance, gives a total running time of $O(2^c + |P|)$ for c-Double-Constraint Coherence. ■

Since Coherence is a special case of Double-Constraint Coherence we also have:

Corollary 5.10. c-Coherence is in FPT and solvable in time $O(2^c + |P|)$.

5.6. A Constructive fpt-Algorithm for c-Coherence

In the previous section I have presented an fpt-algorithm solving c-Coherence in time $O(2^c + |P|)$. As remarked earlier, the algorithm is not constructive. Namely, the reduction rules (DC 2) – (DC 9) all delete element p from N without specifying whether $p \in A$ or $p \in R$. These rules simply use the knowledge that one or the other must be the case to reduce the instance. Similarly, Lemma 5.5 proves that a reduced instance with more than c elements has at least coherence c, but it does not specify how a partition with that amount of coherence may be obtained.

In this section I present a constructive fpt-algorithm for c-Coherence. This algorithm has a running time of $O(2.52^c + |P|^2)$, and thus it is somewhat slower than the algorithm presented in the previous section. However, it has the advantage that it not only solves the decision version of Coherence, but also its search version. The argument is organized as follows. In Section 5.6.1, we first consider connected networks (i.e., networks that consist of a single component). Using the fact that a network on $|P|$ elements that is a tree has coherence $|P| - 1$ (Corollary 5.5), we conclude that a connected network on $|P|$ elements has at least coherence $|P| - 1$. This implies a problem kernel of size $|P| \leq c$ for c-Coherence on connected inputs. Then, in Section 5.6.2, we use the reduction rule (DC 1) to obtain a bound on the number of components. This allows us to conclude a problem kernel of size $|P| \leq 2.52c - 2$ for c-Coherence.

5.6.1. A Problem Kernel for Connected Networks

Lemma 5.6. Let (N, c), with $N = (P, C)$, be an instance for c-Coherence, such that N is connected. If $|P| \geq c + 1$, then (N, c) is a yes-instance for c-Coherence.

Proof: Since N is connected, N has a spanning tree $T = (P_T, C_T)$ with $P_T = P$ and $C_T \subseteq C$. We consider T. From Corollary 5.5 we know that (T, c) is a yes-instance for c-Coherence. Since T is a subgraph of N, we conclude that also (N, c) is a yes-instance for c-Coherence. ■

From Lemma 5.6 we conclude a constructive fpt-algorithm for c-Coherence on connected input networks that runs in time $O(2^c + |P|)$.

Lemma 5.7. c-Coherence for connected networks is solvable by a constructive fpt-algorithm that runs in time $O(2^c + |P|)$.

Proof: We describe an fpt-algorithm for c-Coherence on connected networks. The algorithm takes as input an instance (N, c) for c-Coherence, with connected network $N = (P, C)$. First it counts the elements in P in time $O(|P|)$. We distinguish two cases: (1) Let $|P| \geq c + 1$. Then we conclude from Lemma 5.6 that (N, c) is a yes-instance for c-Coherence. For a constructive result we additionally determine a spanning tree for N in time $O(|P|)$ (Lemma 5.4, page 89), and we build a partition for T with coherence c using the Consistent Coherence algorithm in time $O(|P|)$. The constructive computation also terminates in time $O(3|P|) = O(|P|)$. (2) Let $|P| \leq c$. Then we perform an exhaustive search on all possible partitions of P. This exhaustive search runs in $O(2^{|P|})$ which is $O(2^c)$, since $|P| \leq c$. The exhaustive search time, combined with the time to count the elements in P, gives a total running time of $O(2^c + |P|)$ for c-Coherence. ■

5.6.2. A General Problem Kernel

We show that using reduction rule (DC 1) and Lemma 5.6 we can conclude a problem kernel for c-Coherence for general inputs. In the following, we call a network N *reduced** if (DC 1) does not apply to N.

We start by observing that a reduced* network N with more than c components has at least coherence c.

Lemma 5.8. Let (N, c), with $N = (P, C)$, be an instance for c-Coherence, such that N is reduced* and the number of components in N is at least $\frac{1}{3}c$. Then (N, c) is a yes-instance for c-Coherence.

Proof: Let $N_1, N_2, \ldots, N_x$, with $x \geq \frac{1}{3}c$, be the components in N. Since N is reduced* we know that $|N_i| \geq 4$ for each component N_i, $i = 1, 2, \ldots, x$. From Lemma 5.6 we conclude that the minimum coherence per component is 3. Since N has at least $\frac{1}{3}c$ components we know N has at least coherence $3\frac{1}{3}c = c$. Hence, (N, c) is a yes-instance for c-Coherence. ■

We now show that, if Lemma 5.8 does not apply, a reduced* network $N = (P, C)$ with $|P| \geq 1\frac{1}{3}c - 1$ has at least coherence c.

Lemma 5.9. Let (N, c), with $N = (P, C)$, be an instance for c-Coherence, such that N is reduced*, the number of components in N is at most $\frac{1}{3}c - 1$ and $|P| \geq 1\frac{1}{3}c - 1$. Then (N, c) is a yes-instance for c-Coherence.

Proof: Let $N_1, N_2, \ldots, N_x$, with $x \leq \frac{1}{3}c - 1$, be the components in N. From $N = (P, C)$ we construct a connected network $N^* = (P^*, C^*)$ as follows: We set $P^* = P$ and $C^* = C \cup B$. Here B is a set of *bridges*, connecting the components in N, and is defined as $B = \{(v_i, v_{i+1}) : v_i \in N_i \text{ and } v_{i+1} \in N_{i+1}, i = 1, 2, \ldots, x - 1\}$. Note that N^* is a connected network and has exactly $x - 1$ edges more than N. From Lemma 5.3 we conclude that N^* has at least coherence $|P^*| - 1 = |P| - 1$. Since, there are $x - 1 \leq \frac{1}{3}c - 2$ edges more in N^* then there are in N, we conclude that N has at least coherence $|P| - 1 - \frac{1}{3}c + 2 = |P| - \frac{1}{3}c + 1$. Since $|P| \geq 1\frac{1}{3}c - 1$ it follows that N has at least coherence $1\frac{1}{3}c - 1 - \frac{1}{3}c + 1 = c$. We conclude that (N, c) is a yes-instance for c-Coherence. ■

From Lemma 5.8 and 5.9 we conclude a constructive fpt-algorithm for c-Coherence that runs in time $O(2.52^c + |P|^2)$.

Lemma 5.10. c-Coherence is solvable by a constructive fpt-algorithm that runs in time $O(2.52^c + |P|^2)$.

Proof: We describe an fpt-algorithm for c-Coherence. The algorithm takes as input an instance (N, c) for c-Coherence, with $N = (P, C)$, and first reduces* (N, c) to (N', c'), with $N' = (P', C')$. Then it counts the number of components and the number of elements in P'. All this can be done in time $O(|P|)$. We distinguish three cases: (1) Let the number of components in N' be at least $\frac{1}{3}c'$. From Lemma 5.8 we conclude that (N', c'), and thus also (N, c), is a yes-instance for c-Coherence. (2) Let the number of components in N' be at most $\frac{1}{3}c'-1$ and $|P'| \geq 1\frac{1}{3}c'-1$. From Lemma 5.9 we conclude that (N', c'), and thus also (N, c), is a yes-instance for c-Coherence. To obtain a constructive result for cases (1) and (2), we determine a spanning forest F for N', and build a partition for F with c' coherence using the Consistent Coherence algorithm. (3) Let the number of components in N' be at most $\frac{1}{3}c'-1$ and $|P'| \leq 1\frac{1}{3}c'-2$. Then we perform an exhaustive search on all possible partitions of P. This exhaustive search runs in $O(2^{|P'|})$. Since $|P'| \leq 1\frac{1}{3}c'-2$ and $c' \leq c$, we know that $O(2^{|P'|})$ is $O(2^{\left(1\frac{1}{3}c-2\right)})$. Further, note that $O(2^{\left(1\frac{1}{3}c-2\right)}) = O(2^{1\frac{1}{3}c}) - O(2^2)$, which is $O(2^{1\frac{1}{3}c}) \approx O(2.52^c)$. The exhaustive search time, combined with the time required for the reduction from (N, c) to (N', c'), gives a total running time of $O(2.52^c + |P|^2)$ for c-Coherence. ■

5.7. $|C^-|$-Coherence is in FPT

In this section we consider the parameterized complexity of Coherence when the parameter is the number of negative constraints in the input network, $|C^-|$. The choice of this parameter is motivated by two observations made at the beginning of the chapter: If the input network N contains only negative constraints then Coherence is NP-hard (Corollary 5.2, page 84), but if the input network N contains only positive constraints then Coherence is in P (Observation 5.1, page 86). A question that naturally arises is the following. If the input network contains both negative and positive constraints, does the presence of positive constraints add non-polynomial time complexity *over and above* the non-polynomial time complexity due to the presence of negative constraints? In other words, we ask: Is the parameter $|C^-|$ sufficient for confining the non-polynomial time complexity in the general Coherence problem? I will show that the answer is "yes," by proving that $|C^-|$-Coherence $\in$ FPT.

The proof is organized as follows. First, Section 5.7.1 presents a generalization of Coherence, called Annotated Coherence. Section 5.7.2, presents a branching rule that can be used to build a search tree, with leaves labeled by instances for Annotated Coherence such that all constraints incident to elements in *P* are positive constraints. Section 5.7.3 describes how Annotated Coherence for such special instances can be reduced to the polynomial-time problem Min-Cut. In Section 5.7.4, we conclude an fpt-algorithm for $|C^-|$-Coherence.

For completeness, I note that a direct consequence of Corollary 5.2 is that $|C^+|$-Coherence is not in FPT (unless P = NP).

Corollary 5.11. $|C^+|$-Coherence is not in FPT (unless P = NP).

Proof: Assume that $|C^+|$-Coherence is in FPT and P ≠ NP. Then there exists an algorithm that solves Coherence in time $O(f(|C^+|)\ n^\alpha)$, for some function *f* and some constant α. But that means that we can solve Coherence on networks with only negative constraints in time $O(f(0)\ n^\alpha)$, which is $O(n^\alpha)$; meaning Coherence on networks with only negative constraints is in P. But then, from Corollary 5.2, we have P = NP. ■

5.7.1. Annotated Coherence

In Coherence, any element may be assigned to either side of the partition: i.e., to *A* or to *R*. We have seen in Section 5.4.2, that in Foundational Coherence—a generalization of Coherence—some elements (the ones in *D*) are pre-determined to be in the set *A*. Here, I generalize Coherence even further by also allowing that some elements are pre-determined to be in the set *R*. This generalized problem we call Annotated Coherence.

Annotated Coherence (*decision version*)

Input: A network $N = (P, C)$, with $(P' \cup A' \cup R')^P$ and $(C^+ \cup C^-)^C$. For each $(p, q) \in C$ there is an associated positive integer weight $w(p, q)$. A positive integer *c*.

Question: Does there exist a partition of *P* into *A* and *R* such that $A' \subseteq A$, $R' \subseteq R$, and $\text{Coh}_N(A, R) \geq c$?

One way to think of Annotated Coherence is as the problem that arises when one is in the middle of the process of solving Coherence; i.e., some elements in *P* have already been assigned (the ones in *A'* and *R'*), but other elements in *P* still remain to be assigned (the ones in P').[55] Using this interpretation of Annotated Coherence I will present a branching algorithm for Coherence. This algorithm uses the branching rule described in the next subsection.

5.7.2. Branching into Pos-Annotated Coherence

Let us distinguish between two types of elements in P: the elements that are connected to at least one negative constraint, denoted P^-, and the elements that are connected to positive constraints only, P^+. Note that $P = P^- \cup P^+$. We now define a branching rule, called (AC 1), that uses the observation that for each $p \in P^-$, and an optimal partition $(A \cup R)^P$, either $p \in A$ or $p \in R$. (cf. branching rule (VC 7) in Section 4.3).

(AC 1) The Min-Element-In-*A*-or-*R* Branching Rule: Let node s in the search tree be labeled by instance (N, c) for Annotated Coherence, with $N = (P, C)$, $P = P' \cup A' \cup R'$, and $P = P^- \cup P^+$. Further, let $p \in P'$ such that $p \in P^-$. Then we create two children of s in the search tree, called s_1 and s_2, and label s_1 by (N_1, c_1) and label s_2 by (N_2, c_2). Here $N_1 = (P_1, C_1)$ with $P_1' = P' \backslash \{p\}$, $A_1' = A' \cup \{p\}$, $R_1' = R'$, $c_1 = c$, and $N_2 = (P_2, C_2)$ with $P_2' = P' \backslash \{p\}$, $A_2' = A'$, $R_2' = R' \cup \{p\}$, $c_2 = c$.

Proof: We need to show that (N, c) is a yes-instance for Annotated Coherence if and only if and only if (N_1, c_1) or (N_2, c_2) is a yes-instance for Annotated Coherence. $(\Rightarrow)$ Let (N, c) be a yes-instance for Annotated Coherence. Then there exists a partition $(A \cup R)^P$, with $A' \subseteq A$ and $R' \subseteq R$ such that $\text{Coh}_N(A, R) \geq c$. We distinguish two cases: (1) Let $p \in A$. Then (N_1, c_1) is a yes-instance for Annotated Coherence. (2) Let $p \in R$. Then (N_2, c_2) is a yes-instance for Annotated Coherence. $(\Leftarrow)$ Let (N_1, c_1) or (N_2, c_2) be a yes-instance for Annotated Coherence. We distinguish two cases: (1) Let (N_1, c_1) be a yes-instance for Annotated Coherence. Then there exists a partition $(A_1 \cup R_1)^{P_1}$, with $A_1' \subseteq A_1$ and $R_1' \subseteq R_1$ such that $\text{Coh}_{N_1}(A_1, R_1) \geq c_1$. But then also $\text{Coh}_N(A_1, R_1) \geq c_1 = c$. We conclude (N, c) is a yes-instance for Annotated Coherence. (2) Let (N_2, c_2) be a yes-instance for Annotated Coherence. Then there exists a partition $(A_2 \cup R_2)^{P_2}$, with $A_2' \subseteq$

[55] In the design of fpt-algorithms it is often practical to work with annotated versions of a problem (see e.g. Alber, et al., 2001, Stege and van Rooij, 2003).

A_2 and $R_2' \subseteq R_2$ such that $\text{Coh}_{N_2}(A_2, R_2) \geq c_2$. But then also $\text{Coh}_N(A_2, R_2) \geq c_2 = c$. We conclude (N, c) is a yes-instance for Annotated Coherence. ■

The branching rule (AC 1) can be used to construct a search tree as follows. We take as input an instance (N, c) for Annotated Coherence and apply (AC 1) to (N, c) until it cannot be applied anymore (in which case we know that $P = P^+$). When branching terminates, each leaf s_i of the search tree T is labeled by an instance (N_i, c_i) with all elements in P_i' connected to *positive* constraints only. Further, the application of (AC 1) results in a search tree T with fan$(T) = 2$, and the depth$(T) \leq |P^{\neg}|$, and thus, the size of T is $O(2^{|P^{\neg}|})$.

Note that we have not yet solved Annotated Coherence: all we did so far is create a search tree with $2^{|P^{\neg}|}$ leaves such that each leaf is labeled by a special case of Annotated Coherence in which all "unassigned" elements in P are incident to positive constraints only. We call this special case Pos-Annotated Coherence.

Pos-Annotated Coherence

Input: A network $N = (P, C)$, with $(P' \cup A' \cup R')^P$ and $(C^+ \cup C^-)^C$. For every $(p, q) \in C^-$, $p, q \in A' \cup R'$. For each $(p, q) \in C$ there is an associated positive integer weight $w(p, q)$. A positive integer c.

Question: Does there exist a partition of P' into A and R such that $A' \subseteq A$, $R' \subseteq R$, and $\text{Coh}_N(A, R) \geq c$?

Next we show that Pos-Annotated Coherence is solvable in polynomial time $O(|P^+|^3)$ by reducing it to the known polynomial-time problem Min-Cut.

5.7.3. Pos-Annotated Coherence is in P[56]

Here we show a polynomial-time reduction from Pos-Annotated Coherence to a problem called Min-Cut (cf. Max-Cut discussed in Section 5.3). The problem Min-Cut takes as input an edge-weighted graph $G = (V, E)$, with two special vertices $s, t \in V$, where s is called *source* and t is called *sink*. In its optimization version, the goal is to partition the vertex set V into sets A and R, such $s \in A$, $t \in R$, and the sum of the weights on edges that

[56] I thank Allan Scott and Parissa Agah for useful discussions on the problem Pos-Annotated Coherence. I am indebted to Allan Scott for the idea that Pos-Annotated Coherence reduces to Min-Cut.

have one endpoint in A and one endpoint in R is minimum. The decision version of this problem is as follows:

Min-Cut (*decision version*)

Input: An edge weighted graph $G = (V, E)$. A source $s \in V$ and a sink $t \in V$. For each edge $(u,v) \in E$ there is an associated positive integer weight $\mathrm{w}(u, v)$. A positive integer k.

Question: Does there exist a partition of V into disjoint sets A and R such that, $s \in A$, $t \in R$, and $W_G(A, R) = \sum_{(u,v)\in \mathrm{Cut}_G(A,R)} \mathrm{w}(u,v) \leq k$? Here $\mathrm{Cut}_G(R, A)= \{(u, v) \in E : u \in A \text{ and } v \in R\}$.

It is known that Min-Cut is solvable in time $O(|V|^3)$ (see e.g. Cormen, Leiserson, & Rivest, 1990).[57]

The reduction from Pos-Annotated Coherence to Min-Cut involves a couple of steps. I first present two reduction rules, (AC 2) and (AC 3); see Figure 5.3 for an illustration. After having applied these reduction rules to an instance (N, c) for Pos-Annotated Coherence we know that for the reduced instance (N_i, c_i) there only exist one element $s \in A_i'$ and one element $t \in R_i'$. I then present a reduction from reduced instances for Pos-Annotated Coherence to Min-Cut.

The reduction rule (AC 2) for Annotated Coherence is based on the observation that for constraints with both endpoints in $A' \cup R'$ we can simply check whether or not they are satisfied by the assignment $A' \cup R'$, delete them from the network, and update c accordingly. Namely, for those constraints it is predetermined whether or not they will be satisfied in the final partition.

[57] I thank Minko Markov for bringing to my attention the classic result that Min-Cut is equivalent to a problem called Max-Flow. Algorithms for Min-Cut/Max-Flow can be found in many introductory textbooks on algorithms and/or graph theory (e.g. Cormen, Leiserson, & Rivest, 1990; Gross & Yellen, 1990; Gould, 1988; Foulds, 1992).

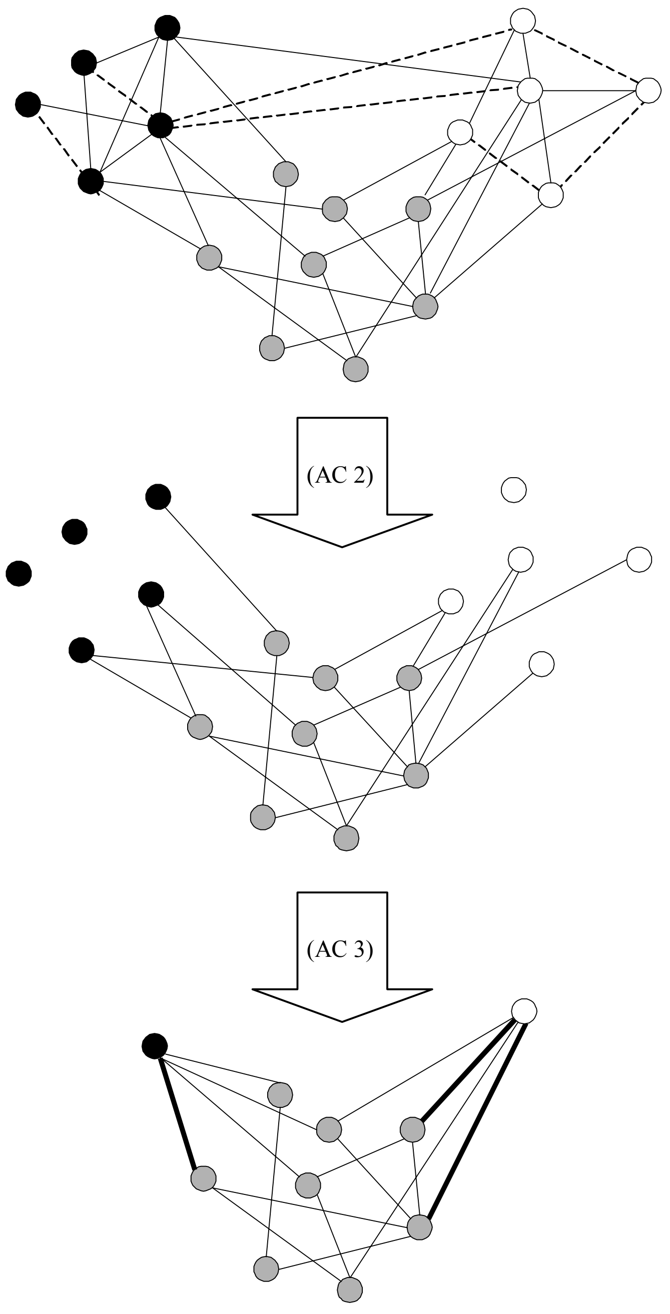


Figure 5.3. Illustration of reduction rules (AC 2) and (AC 3).
Here the rules (AC 2) and (AC 3) are applied to an instance for a Pos-Annotated Coherence. Positive constraints are indicated by solid lines and negative constraints by dotted lines. The black dots represent elements in *A'*, the white dots represent elements in *R'*, and the gray dots represent elements in *P'*. Application of (AC 2) to the instance results in the removal of all constraints with both endpoints in $A' \cup R'$, and subsequent application of (AC 3) results in merging all elements in *A'* into one element *s*, and in merging all elements in *R'* into one element *t*. The bold lines in the bottom figure indicate that the weights of those constraints have been changed in the merge procedure of (AC 3). Note that the instance at the bottom is an instance for the Min-Cut problem.

(AC 2) Delete Pre-Determined Constraints Rule: Let (N, c) be an instance for Annotated Coherence. If there exists a constraint $(p, q) \in A' \cup R'$, then let $P^* = P$, and $C^* = C \backslash (p, q)$ and further,

(1) if $(p, q) \in C^+$ and $p, q \in A'$ or $p, q \in R'$ then $c^* = c - \mathrm{w}(p, q)$,

(2) if $(p, q) \in C^-$ and $p, q \in A'$ or $p, q \in R'$ then $c^* = c$,

(3) if $(p, q) \in C^+$ and ($p \in A'$ and $q \in R'$) or ($p \in R'$ and $q \in A'$) then $c^* = c$,

(4) if $(p, q) \in C^-$ and ($p \in A'$ and $q \in R'$) or ($p \in R'$ and $q \in A'$) then $c^* = c - \mathrm{w}(p, q)$.

Finally, let the resulting instance, (N^*, c^*), be the new instance for Annotated Coherence.

Proof: Annotated Coherence is defined such that for any candidate solution $(A \cup R)^P$, $A' \subseteq A$ and $R' \subseteq R$. Thus the weight of an edge $(p, q) \in A' \cup R'$ will count towards coherence if and only if condition (1) or condition (4) is met. ■

The second reduction rule, (AC 3), is applied only if rule (AC 2) does not apply. The rule (AC 3) is based on the observation that for all elements in A' we can merge them into one single element s without affecting the amount of coherence in the network; similarly, for all elements in R' we can merge them into one single element t.

(AC 3) Merge Pre-Assigned Elements Rule: Let (N, c) be an instance for Annotated Coherence such that (AC 2) does not apply. If there exist two elements p and q such that $p, q \in A'$ or $p, q \in R'$, then let $P^* = P \backslash \{q\}$, $C^* = (C \backslash \mathrm{R}_N(q)) \cup C_{qp}$ where $C_{qp} = \{(p, r) : (q, r) \in \mathrm{R}_N(q)\}$, and $c^* = c$. Further, we update the weight function w*(.) for each constraint $(x, y) \in C^*$ as follows:

(1) if $(x, y) \in C^* \backslash C_{qp}$ then $\mathrm{w}^*(x, y) = \mathrm{w}(x, x)$,

(2) if $(x, y) \in C_{qp}$ and $(x, y) \notin \mathrm{R}_N(q)$ then $\mathrm{w}^*(x, y) = \mathrm{w}(x, y)$

(3) if $(x, y) \in C_{qp}$ and $(q, y) \in \mathrm{R}_N(q)$ then $\mathrm{w}^*(x, y) = \mathrm{w}(x, y) + \mathrm{w}(q, y)$.

(4) if $(x, y) \in C_{qp}$ and $(x, q) \in \mathrm{R}_N(q)$ then $\mathrm{w}^*(x, y) = \mathrm{w}(x, y) + \mathrm{w}(x, q)$.

Finally, let the resulting instance, (N^*, c^*), be the new instance for Annotated Coherence.

Proof: Since (AC 2) does not apply, we know that for any two elements $p, q \in A' \cup R'$, $(p, q) \notin C$. Hence, deleting q from P does not cause deletion of a constraint between p and q. By (1) re-connecting all constraints that previously connected P' to q such that they now connect to p instead of q, and (2) setting the weight for each constraint

$(p, r) \in C^*$ as $w^*(p, r) = w(p, r) + w(q, r)$ (here we define $w(q, r) = 0$, if $(q, r) \notin C$), we ensure that (N, c) is a yes-instance for Annotated Coherence if and only if (N^*, c^*) is a yes-instance for Annotated Coherence. ■

We say an instance (N, c) for Annotated Coherence is *reduced* if and only if (AC 2) and (AC 3) do not apply (N, c). The following lemma presents a polynomial-time reduction from reduced instances for Pos-Annotated Coherence to Min-Cut.

Lemma 5.11. Let (N, c), with $N = (P, C)$, be a reduced instance for Pos-Annotated Coherence, with $A' = \{s\}$ and $R' = \{t\}$. Let (G, k), with $G = (V, E)$ such that $V = P$ and $E = C$, for every $(u, v) \in E$, $w_G(u, v) = w_N(u, v)$ and $k = \sum_{(p,q)\in C} w(p,q) - c$.

Then (N, c) is a yes-instance for Pos-Annotated Coherence if and only if (G, k) is a yes-instance for Min-Cut.

Proof: Since (N, c) is a reduced instance for Pos-Annotated Coherence we know $C = C^+ = E$. Let $(A \cup R)^P = (A \cup R)^V$ be any partition. Then for every $(p, q) \in C = E$, $(p, q) \in \text{Cut}_G(A, R)$ if and only if $(p, q) \notin S_N(A, R)$. In other words, $C = \text{Cut}_G(A, R) \cup S_N(A, R)$. Thus we have $\text{Coh}_N(A, R) = \sum_{(p,q)\in C} w(p,q) - W_G(A, R)$. We conclude that (N, c) is a yes-instance for Pos-Annotated Coherence if and only if (G, k) is a yes-instance for Min-Cut. ■

5.7.4. An fpt-algorithm for $|C^{\neg}|$-Annotated Coherence

Given the observations made in Sections 5.7.2, and 5.7.3, we are now in a position to construct an fpt-algorithm for $|C^{\neg}|$-Annotated Coherence. The algorithm works as follows: It takes as input an instance (N, c) for Annotated Coherence, and recursively applies branching rule (AC 1) until no longer possible. Then each leaf i, with $i = 1, 2, \ldots, 2^{|P^{\neg}|}$, in the resulting search tree is labeled by an instance (N_i, c_i) for Pos-Annotated Coherence. Subsequently, the algorithm reduces each (N_i, c_i) using rules (AC 2) and (AC 3). Finally, it solves the Min-Cut problem for each reduced instance (N_i', c_i').

Since the search tree size is bounded by $O(2^{|P^{\neg}|})$, the reduction rules (AC 2) and (AC 3) can be applied in time $O(|P|^2)$, and Min-Cut can be solved in time $O(|P|^3)$, we conclude that the whole algorithm runs in time $O(2^{|P^{\neg}|}(|P|^2 + |P|^3))$ which is $O(2^{|P^{\neg}|}|P|^3)$. Because this running time is fpt-time for parameter $|P^{\neg}|$ we conclude Theorem 5.3.

Theorem 5.3. $|P^{-}|$-Annotated Coherence is in FPT.

Since $|P^{-}| \leq |C^{-}|$ we know that $O(2^{|P^{-}|}|P|^3)$ is $O(2^{|C^{-}|}|P|^3)$,

Corollary 5.12. $|C^{-}|$-Annotated Coherence is in FPT.

Since, Coherence is a special case of Annotated Coherence, we also conclude:

Corollary 5.13. $|P^{-}|$-Coherence is in FPT.

Corollary 5.14. $|C^{-}|$-Coherence is in FPT

Note that Theorem 5.3 presents a stronger result than Corollary 5.12. It shows that although the number of negative constraints is sufficient for capturing the non-polynomial time complexity inherent in Annotated Coherence, it is not necessary—i.e., the number of elements that are incident to at least one negative constraint suffices as well.

5.8. Conclusion

In this chapter I have illustrated techniques for complexity analysis by considering the problem Coherence as defined by Thagard (2000) and Thagard and Verbeurgt (1998). Section 5.3 illustrated the use of polynomial-time reduction to prove that Coherence is NP-hard. Section 5.4 discussed and illustrated the conditions under which this NP-hardness result generalizes to other coherence problems. Sections 5.5 and 5.6 presented two different ways to use reduction and kernelization rules to derive a problem kernel for *c*-Coherence. Finally, Section 5.7 illustrated how a branching rule and reduction rules can be combined to construct a bounded search tree for $|C^{-}|$-Coherence.

Besides illustrating techniques, the analyses have lead to some interesting observations about Coherence. Among other things, we have found that Coherence is computationally easy (1) if it is possible to satisfy all constraints, (2) if the amount of coherence that one is aiming for is not too high, and (3) if the number of negative constraints in the network is not too large. Further, we observed that result (2) also holds for the more general problem Double-Constraint Coherence, and that result (3) also holds for the more general problem Annotated Coherence (including also Foundational Coherence as a special case).

Many open questions remain. As I noted in Section 5.4, it remains to be shown whether or not Single-Element Discriminating Coherence and Single-Element

Foundational Coherence are NP-hard. Also, it is of interest to study the classical complexity of other special cases and variants of Coherence than the ones considered in Sections 5.4.1 and 5.4.3. In this chapter, I have presented several fpt-algorithms for Coherence: two for parameter c, one for parameter $|P^-|$, and one for parameter $|C^-|$. Future research may investigate whether it is possible to obtain even faster fpt-algorithms for these parameterizations. Further, Coherence and its generalizations have many implicit parameters that I have not considered. For some such parameters it is easily determined whether or not they are sufficient to capture the non-polynomial time complexity in Coherence (e.g., the parameter $|D|$ in Discriminating Coherence and Foundational Coherence) but for others this is far from trivial (e.g., the parameter $|H|$ in Discriminating Coherence and Foundational Coherence).[58] Lastly, since one ideally aims for low incoherence (i.e., a small number of unsatisfied constraints), a parameter that may be of particular interest is the relational parameter $\iota = \sum_{(p,q) \in C} w(p,q) - c$. Is ι-Coherence in FPT? As explained in Section 4.4.2, the fact that c-Coherence $\in$ FPT does not answer this question.

[58] Note, however, that if elements in D are connected to elements in H by *positive* constraints only (as generally seems to be the case in applications discussed by Thagard (2000), then $|H|$-Foundational Coherence is in FPT by Theorem 5.3 (page 114), since then $D \subseteq P^+$.

Chapter 6. Subset Choice[59]

In this chapter we consider the problem Subset Choice, a generalization of a model by Fishburn and LaValle (1996). I start by describing the general problem Subset Choice as defined by van Rooij, Stege and Kadlec (2003). Then I discuss applications of Subset Choice in human decision-making. The main part of this chapter is again devoted to classical and parameterized complexity analyses of the problem. As before, in interpreting the results we assume P ≠ NP and FPT ≠ W[1]. I will close with a brief discussion and suggestions for future research.

6.1. Subset Choice as Hypergraph Problem

Subset choice denotes the situation in which a decision maker is presented with a set of choice alternatives and is asked to choose a subset from the available set. Here we consider subset choice problems in which the goal is to choose a subset with satisfactory (subjective) value. Fishburn and LaValle (1996) presented a model of the value of sets (and subsets) using weighted graphs. In the following, I present a generalization of their model using weighted hypergraphs (see also van Rooij et al., 2003).

A hypergraph is a generalization of the concept of a graph. In a graph $G = (V, E)$, the set E consists of unordered *pairs* of distinct vertices, i.e., $E \subseteq V \times V$. In a *hypergraph* $H = (V, E)$, the set E consists of unordered h-tuples of distinct vertices, $2 \leq h \leq |V|$ (i.e., $E \subseteq \bigcup_{2 \leq h \leq |V|} V^h$, where V^h denotes the h-fold product of V). In other words, a graph is a special type of hypergraph—viz., one in which E contains only 2-tuples of vertices. In a hypergaph, we call an element in E a *hyperedge*.

In the context of Subset Choice, every vertex $v \in V$ represents a choice alternative, and a hyperedge $(v_1, v_2, ..., v_h) \in E$, $2 \leq h \leq |V|$, represents the presence of an h-way interaction between choice alternatives $v_1, v_2, ..., v_h$. Each vertex $v \in V$ has an associated *vertex weight* $w_V(v)$, and each hyperedge $e = (v_1, v_2, ..., v_h)$, $e \in E$, has associated *hyperedge weight* $w_E(e)$. For simplicity, we assume that $w_V(v) \in \mathbb{Z}$, and $w_E(e)$

[59] Parts of this chapter also appear in a manuscript submitted for publication by I. van Rooij, U. Stege and H. Kadlec (2003), entitled *Sources of Complexity in Subset Choice*.

$\in \mathbb{Z}\backslash\{0\}$ (here $\mathbb{Z}$ denotes the set of integers, and $\mathbb{Z}\backslash\{0\}$ denotes the set of non-zero integers).[60] A vertex weight $w_V(v)$ represents the value of choice alternative v when evaluated in isolation. A hyperedge weight $w_E(e)$, with $e = (v_1, v_2, ..., v_h)$, represents the added value of the combination of vertices $v_1, v_2, ..., v_h$ over and above the value of the singular elements $v_1, v_2, ...,$ and v_h and over and above the value of all hyperedges that are combinations of at most $h - 1$ vertices in $\{v_1, v_2, ..., v_h\}$.

We assume that a decision-maker, when presented with a set of choice alternatives V, embodies a (latent) value-structure that can be modeled by a weighted hypergraph $H = (V, E)$. The value associated with choosing subset $V' \subseteq V$ for a value-structure H is then given by:

$$\text{value}_H(V') = \sum_{v \in V'} w_V(v) + \sum_{e \in E_H(V')} w_E(e), \qquad (1)$$

where $E_H(V') = \{(v_1, v_2, ..., v_h) \in E \mid v_1, v_2, ..., v_h \in V'\}$.

Having introduced the necessary terminology, we can now define the general Subset Choice problem:

Subset Choice

Input: A *weighted* hypergraph $H = (V, E)$, $E \subseteq \bigcup_{2 \le h \le |V|} V^h$. For every $v \in V$ there is a weight $w_V(v) \in \mathbb{Z}$ and for every $e \in E$ there is a weight $w_E(e) \in \mathbb{Z}\backslash\{0\}$. A positive integer p.

Question: Does there exist a subset $V' \subseteq V$ such that $\text{value}_H(V') \ge p$?

6.1.1. Notation and Terminology

We define notation and terminology specifically for hypergraphs: We say a vertex $v \in V$ is *incident* to hyperedge $e = (v_1, v_2, ..., v_h)$ and, conversely, e is incident to v, if $e \in E$ and $v \in \{v_1, v_2, ..., v_h\}$. The *degree* of a vertex $v \in V$ is the total number of hyperedges in H that are incident to v, denoted $\deg_H(v)$. The *span* of an hyperedge $e = (v_1, v_2, ..., v_h)$ is denoted by $\text{span}(e) = h$. Two vertices $u, v \in V$ are *neighbors* in H if there exists an hyperedge $(v_1, v_2, ..., v_h) \in E$ with $u, v \in \{v_1, v_2, ..., v_h\}$. The (open) *neighborhood* $N_H(v)$

[60] All results reported in this chapter generalize straightforwardly to value-structures with any values of fixed precision, simply by scaling the weights and results by the precision factor.

is the set of neighbors of v in H, and the *closed neighborhood* is denoted $N_H[v] = N_H(v) \cup \{v\}$. Note that in graphs we have $\deg_G(v) = |N_G(v)|$ for every $v \in V$, but in hypergraphs $\deg_H(v)$ may be larger or smaller than $|N_H(v)|$ for every $v \in V$. Further note that for any specific hypergraph $H = (V, E)$ there exists a positive integer $\varepsilon \le |V|$ such that $\text{span}_H(e) \le \varepsilon$.

In our analyses we will sometimes consider the special case that $\varepsilon = 2$ (i.e., the special case that the hypergraph is a graph; cf. Fishburn & LaValle, 1996). Further, we will consider value-structures that can be represented by hypergraphs with special weighting functions. We define the following classes for hypergraphs. Let $H = (V, E)$ be a weighted hypergraph. Then we say: (1) H is a *unit-weighted* hypergraph if $w_V(v) \in \{-1, +1\}$ for all $v \in V$ and $w_E(e) \in \{-1, +1\}$ for all $e \in E$; (2) H is an *edge-weighted* hypergraph if $w_V(v) \in \{-1, +1\}$ for all $v \in V$ and $w_E(e) \in \mathbb{Z}\backslash\{0\}$ for all $e \in E$; (3) H is a *vertex-weighted* hypergraph if $w_V(v) \in \mathbb{Z}$ and $w_E(e) \in \{-1, +1\}$ for all $e \in E$; (4) H is a *conflict* hypergraph if $w_V(v) \ge 0$ for all $v \in V$ and $w_E(e) \le -1$ for all $e \in E$; (5) H is a *surplus* hypergraph if $w_V(v) \le 0$ for all $v \in V$ and $w_E(e) \ge +1$ for all $e \in E$. Intersections of these classes define further special cases considered in our analyses. Table 6.1 gives an overview of the special value-structures that we will consider.

Table 6.1. Overview of special value-structures.

value-structure	$w_V(v)$	$w_E(e)$	span(e)
unit-weighted conflict graph (UCG)	+1	−1	2
unit-weighted surplus graph (USG)	−1	+1	2
edge-weighted conflict graph (ECG)	+1	≤ -1	2
vertex-weighted conflict graph (VCG)	≥ 1	−1	2
conflict graph (CG)	≥ 1	≤ -1	2
unit-weighted conflict hypergraph (UCH)	+1	−1	$\le \|V\|$
conflict hypergraph (CH)	≥ 1	≤ -1	$\le \|V\|$
surplus hypergraph (CH)	≤ -1	≥ 1	$\le \|V\|$

Note: $w_V(v)$ denotes the weight of vertices, $w_E(e)$ denotes the weight on hyperedges, and span(e) denotes the span of hyperedges in the value-structure.

In our analyses we also consider several explicit, implicit and relational parameters of the input. Table 6.2 presents an overview of all parameters considered in this chapter.

Table 6.2. Overview of input parameters for Subset Choice.

Parameter	Definition
ε	For all $e \in E$, $\text{span}_H(e) \leq \varepsilon$
Ω_V	For all $v \in V$, $w_V(v) \leq \Omega_V$
ω_V	For all $v \in V$, $w_V(v) \geq -\omega_V$
Ω_E	For all $e \in E$, $w_E(e) \leq \Omega_E$
ω_E	For all $e \in E$, $w_E(e) \leq -\omega_E$
Δ	For all $v \in V$, $\text{deg}_H(v) \leq \Delta$
θ	For all $v \in V$, $N_H(v) \leq \theta$
p	A positive integer
q	$q = p - \text{value}_H(V)$

6.2. Subset Choice as Cognitive Theory

The problem Subset Choice arises in a many different settings (see e.g. Bossert, 1989, Farquhar & Rao, 1976; Fisburn & LaValle, 1993, 1996; Kannai & Peleg, 1984), including medical decision-making, management, voting and consumer choice. Below I will briefly sketch applications in each domain. See also Table 1 in Farquhar and Rao (1976) for an overview of many more applications.

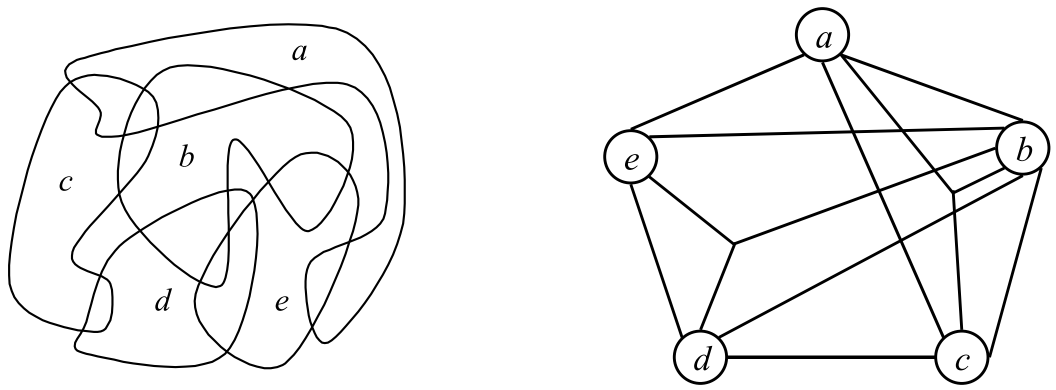
Medical Decision-making: The task of a physician to prescribe a combination of medications to a patient with multiple ailments can be modeled as follows. Let each vertex $v \in V$ represent a medication, and its vertex weight $w_V(v)$ represent the benefits for the patient of taking medication v when considered in isolation. Further, let each hyperedge $e = (v_1, v_2, \ldots, v_h)$, $e \in E$ and its weight $w_E(e)$ model the beneficial or detrimental effects of taking medications $v_1, v_2, \ldots, v_h$ in combination. The physician's task may then be to prescribe a subset of medications such that the overall benefit to the patient is satisfactory. Similarly, a physician's choice of medical tests to diagnose a patient can be modeled as a Subset Choice problem (see also Farquhar & Rao, 1976).

Management: The task of forming a committee (or a work team) can be modeled as follows. Let each vertex $v \in V$ represent a candidate for the committee, and its vertex weight $w_V(v)$ represents the (judged) individual contribution of candidate v to the group (e.g., his/her individual skills and abilities). Further, each hyperedge $e = (v_1, v_2, \ldots, v_h)$, $e \in E$, models an interdependency between candidates $v_1, v_2, \ldots, v_h$, with a positive weight $w_E(e)$ meaning that combining the candidates $v_1, v_2, \ldots, v_h$ leads to increased productivity, and a negative weight $w_E(e)$ meaning that combining $v_1, v_2, \ldots, v_h$ leads to reduced productivity (cf. Fishburn & LaValle, 1996). The goal is to choose a set of candidates such that the overall level of productivity is satisfactory. Similarly, the problem of selecting a set of job applicants can be modeled as a Subset Choice problem (see also Figure 6.1 for an illustration).

Voting: The task of electing a set of political representatives can be modeled analogously to the task of forming a committee and/or selecting job applicants described above (cf. Haynes, Hedetniemi, & Slater, 1998). Also, voting in an approval voting system constitutes a subset choice problem (e.g. Falmagne & Regenwetter, 1996; Regenwetter, Marley, & Joe, 1998).[61]

Consumer Choice: Many consumer choice problems are subset choice problems. Consider, for example, a consumer that wants to buy a pizza and must decide on a set of pizza toppings (or, any other menu selection task; see also Farquhar & Rao, 1976). Then each vertex weight $w_V(v)$ represents the individual taste value of topping v. Further, for each hyperedge $e = (v_1, v_2, \ldots, v_h)$, a positive weight $w_E(e)$ means that toppings $v_1, v_2, \ldots, v_h$ compliment each other, while a negative weight $w_E(e)$ means that toppings $v_1, v_2, \ldots, v_h$ clash. The goal of the consumer is to choose a set of toppings of satisfactory tastiness. Similarly, when purchasing a computer, the task of choosing a subset of satisfactory value from among all computer options (monitor, software, printer, scanner, DVD, CD burner, etc.) is a subset choice problem (see also Fishburn & LaValle, 1996).

[61] In an approval voting system, the voter is to indicate for each political candidate whether or not s/he approves of the candidate, and the candidate with the most approval votes wins.



$w_V(a) = 20.2$
$w_V(b) = 22.5$
$w_V(c) = 19.6$
$w_V(d) = 18.4$
$w_V(e) = 16.3$

$w_E(a, b) = -3.8$
$w_E(a, c) = -2.4$
$w_E(a, e) = -1.2$
$w_E(b, c) = -1.2$
$w_E(b, d) = -4.1$

$w_E(b, e) = -0.8$
$w_E(c, d) = -0.9$
$w_E(d, e) = -3.8$
$w_E(a, b, c) = 0.8$
$w_E(b, d, e) = 0.7$

Figure 6.1. Example of a Subset Choice problem.
A university department has available a set of 3 professor positions, and a search committee has to choose from among 5 applicants, called *a*, *b*, *c*, *d*, and *e*, to fill the positions. The example assumes that applicants are being evaluated solely in terms of the new contributions that they bring to a department. The figure on the left gives a schematic representation of the applicants' fields of specialization and their overlap. The hypergraph *H* (on the right) and the vertex and hyperedge weights (on the bottom) model the situation as follows. Each vertex *p* in *H* represents an applicant, and each hyperedge $(p_1, p_2, \ldots, p_h)$ in *H* represents an overlap between the fields of applicants $p_1, p_2, \ldots, p_h$. Each applicant *p* has an associated value, denoted $w_V(p)$, representing the judged contribution of person *p* to the department when evaluated independently of the other applicants. If there is overlap between the fields of two applicants *p* and *q*, then hiring both *p* and *q* will contribute less to the department than the sum of their respective contributions (viz., then a part of what *p* contributes is also contributed by *q*, and vice versa). This is represented by a negative weight $w_E(p, q)$. If there is an overlap of three fields (e.g., in this example, fields of *a*, *b* and *c* overlap), we define a positive weight for their combination. This weight corrects for the over-counting due to the lower order weights. For example, weight $w_E(a, b, c)$ is set so as to compensate in the computation of $\text{value}_H(\{a, b, c\})$ for the fact that in the sum $w_V(a) + w_V(b) + w_V(c) + w_E(a, b) + w_E(a, c) + w_E(b, c)$ the overlap between *a*, *b* and *c* has been removed once to often.

I close this section with a comment on the relationship between Subset Choice and Coherence. Note that Coherence can be seen as a variant of Subset Choice; viz., one in which the value function depends on both the set of chosen alternatives (the set *A* in Coherence) and the set of rejected alternatives (the set *R*). Further, like Subset Choice, Coherence can be naturally extended to networks that are hypergraphs (see e.g. Schoch, 2000). Due to the similarities between the two problems, their respective areas of application may overlap and they may then serve as competing models. The relationship between Coherence and Subset Choice will be further illustrated in a reduction from Coherence to Subset Choice presented in Section 6.3 (Lemma 6.3, page 126).

6.3. Subset Choice is NP-hard

In this section we prove the following theorem.

Theorem 6.1. Subset Choice is NP-hard.

For illustrative purposes, I present three different proofs of Theorem 6.1. The first of these proofs, in Lemma 6.1, is an adaptation of the reduction from Independent Set to Profit Independence as presented in Section 4.6 (page 77). The second, in Lemma 6.2, is due to Fishburn and LaValle (1996) and also involves a polynomial-time reduction from the problem Independent Set. The third proof involves a reduction from Coherence (Lemma 6.3). Later in this chapter, even a fourth proof of Theorem 6.1 will appear (in Lemma 6.5, page 138). This latter proof involves yet another reduction from Independent Set.

I present multiple proofs of Theorem 6.1 for several reasons. First, I wish to illustrate that many different polynomial-time reductions are possible to prove that a problem is NP-hard. This is not surprising, of course, since all NP-hard problems are all directly or indirectly reducible to each other, and thus we know there exist at least as many reductions between any two NP-hard problems as there are NP-hard problems. What Lemma 6.1 and 6.2 (see also Lemma 6.5 in Section 6.5.2) illustrate, however, is that even multiple more-or-less direct reductions from one problem to another may exist. Second, the first proof (unlike the proof due to Fishburn & LaValle, 1996) has as a corollary that Subset Choice is NP-hard even in the very restricted case when the value-structure is a unit-weighted conflict graph. This result serves as the basis for the parameterized complexity analyses in Sections 6.4 and 6.5. Third, in the present context it is useful to illustrate the close relationship between Coherence and Subset Choice. This also sets up the possibility for generalizing certain results for Subset Choice to Coherence.

We start with the first reduction from Independent Set to Subset Choice (cf. Lemma 4.3, page 77).

Lemma 6.1. Let the graph $G = (V, E)$ and the positive integer k form an instance for Independent Set. Then we define an instance for Subset Choice, consisting of a weighted hypergraph G^* with span $\varepsilon = 2$ (i.e., G^* is a graph) and positive integer p, as follows. Let $G^* = (V^*, E^*)$ with $V^* = V$ and $E^* = E$. Further, for every $v \in V^*$ let $\mathrm{w_V}(v) = +1$ and for every $e \in E^*$ let $\mathrm{w_E}(e) = -1$. Let $p = k$. Then G and k form a yes-instance for Independent Set if and only if G^* and p form a yes-instance for Subset Choice.

Proof: ($\Rightarrow$) Let (G, k) be a yes-instance for Independent Set. Then there exists an independent set $V' \subseteq V$ for G with $|V'| \geq k$. This means that $\mathrm{E}_G(V') = \emptyset$ and therefore $\mathrm{E}_{G^*}(V') = \emptyset$. Thus $\mathrm{value}_{G^*}(V') = \sum_{v \in V'} \mathrm{w_V}(v) + \sum_{e \in \mathrm{E}_{G^*}(V')} \mathrm{w_E}(e) = |V'| - |\mathrm{E}_{G^*}(V')| = |V'| \geq k = p$. Therefore, (G^*, p) is a yes-instance of Subset Choice.

($\Leftarrow$) Let (G^*, p) be a yes-instance for Subset Choice. Then there exists a subset $V' \subseteq V^*$ with $\mathrm{value}_{G^*}(V') \geq p$. We distinguish two cases: (1) If $\mathrm{E}_{G^*}(V') = \emptyset$ then $\mathrm{E}_G(V') = \emptyset$ and therefore V' is an independent set for G, with $|V'| \geq p = k$. We conclude that (G, k) is a yes-instance of Independent Set. (2) If $\mathrm{E}_{G^*}(V') \neq \emptyset$ then $\mathrm{E}_G(V') \neq \emptyset$. We transform V' into an independent set V'' for G using the following algorithm:

```
1. $V'' \leftarrow V'$
    2. while $\mathrm{E}_G(V'') \neq \emptyset$ do
        3. pick an edge $(u,v) \in \mathrm{E}_G(V'')$
        4. $V'' \leftarrow V'' \backslash \{v\}$
    5. end while
6. return $V''$
```

The algorithm considers each edge in G at most once and thus runs in time $O(|E|)$ or $O(|V|^2)$. Note that every call of line 4 results in the removal of at least one edge from $\mathrm{E}_G(V'')$. Hence, $\mathrm{value}_{G^*}(V'') \geq \mathrm{value}_{G^*}(V') \geq p$. Furthermore, when the algorithm halts then $\mathrm{E}_G(V'') = \emptyset$ and thus V'' is an independent set of size at least $p = k$ for G. We conclude that (G, k) is a yes-instance for Independent Set. ■

Note that the reduction in lemma 6.1 is a polynomial-time reduction. Namely, we can copy every element in V and E to V^* and E^* respectively in time $O(|V|^2)$, we can set $p = k$ in time $O(1)$, and we can assign each vertex in V the weight '1' and assign each edge in E the weight '−1' in time $O(|V|^2)$. Further, the algorithm that, given a subset with value at least p, computes an independent set of size at least $k = p$, runs in time $O(|V|^2)$. Since Independent Set is known to be NP-complete (e.g. Garey & Johnson, 1979), Lemma 6.1 proves Theorem 6.1.

Observe that, in the proof of Theorem 6.1, G^* is always a unit-weighted conflict graph (i.e., $\mathrm{w_V}(v) = 1$ for all $v \in V$ and $\mathrm{w_E}(e) = -1$ for all $e \in E$). Thus the proof shows that Subset Choice is NP-hard even in the restricted case where the value-structure can be represented by a unit-weighted conflict graph.[62] I will refer to this special case as Unit-weighted Conflict Graph (UCG) Subset Choice.

Corollary 6.1. UCG Subset Choice is NP-hard.

I remark that, since for any unit-weighted conflict graph $G = (V, E)$ and any subset $V' \subseteq V$, $\mathrm{value}_G(V') = \sum_{v \in V'} \mathrm{w_V}(v) + \sum_{e \in \mathrm{E}_G(V')} \mathrm{w_E}(e) = |V'| - |\mathrm{E}_G(V')| = \mathrm{profit}_{\mathrm{PI},G}(V')$, the problem UCG Subset Choice is equivalent to the problem Profit Independence introduced in Chapter 4 (page 68).

Lemma 6.2 presents the reduction from Independent Set to Subset Choice proposed by Fishburn and LaValle (1996, p. 189). I note that Fishburn and LaValle did not present a proof of their reduction. It turns out that the proof of Lemma 6.1 also works identically for Lemma 6.2. It is possible, however, to prove the "only if" direction in

[62] Note that any value-structure $G = (V, E)$, $E \subseteq V^2$, with integer weight $\mathrm{w_V}(v) = \alpha$, for constant $\alpha \geq 1$, and $\mathrm{w_E}(e) = -\mathrm{w_V}(v)$, for all $v \in V$ and all $e \in E$, can be represented by a unit-weighted conflict graph $G^* = (V^*, E^*)$, with $\mathrm{w^*_V}(v) = 1$ for all $v \in V$ and $\mathrm{w^*_E}(e) = -1$ for all $e \in E$. In general, any value structure $H = (V, E)$ with $\mathrm{w_V}(v) \in \mathbb{Z}$ for all $v \in V$ and $\mathrm{w_E}(e) \in \mathbb{Z} \setminus \{0\}$ for all $e \in E$, can be modeled by a hypergraph $H^* = (V^*, E^*)$, with $\mathrm{w^*_V}(v) = \frac{\mathrm{w_V}(v)}{d}$ for all $v \in V^*$ and $\mathrm{w^*_E}(e) = \frac{\mathrm{w_E}(e)}{d}$ for all $e \in E^*$, where d is a *common divisor* of all vertex and hyperedge weight values in H. There exist a subset $V' \subseteq V$ with $\mathrm{value}_H(V') = p$ for H if and only if $\mathrm{value}_{H^*}(V') = \frac{p}{d}$ for H^*.

Lemma 6.2 with a simpler argument than the one used for Lemma 6.1. I will present this simpler argument in the proof below.

Lemma 6.2. Let the graph $G = (V, E)$ and the positive integer k form an instance for Independent Set. Then we define an instance for Subset Choice, consisting of a weighted hypergraph G^* with span $\varepsilon = 2$ (i.e., G^* is a graph) and positive integer p, as follows. Let $G^* = (V^*, E^*)$ with $V^* = V$ and $E^* = E$. Further, for every $v \in V^*$ let $w_V(v) = +1$ and for every $e \in E^*$ let $w_E(e) = -|V|^2$. Let $p = k$. Then G and k form a yes-instance for Independent Set if and only if G^* and p form a yes-instance for Subset Choice.

Proof: ($\Rightarrow$) Identical to the proof of Lemma 6.1. ($\Leftarrow$) Let (G^*, p) be a yes-instance for Subset Choice. Then there exists a subset $V' \subseteq V^*$ with $\text{value}_{G^*}(V') \geq p$. We observe that the size of V' is bounded by $|V|$. We show that $E_{G^*}(V') = \emptyset$. Assume that $E_{G^*}(V') \neq \emptyset$ then $\text{value}_{G^*}(V') < 0$. Since p is a positive integer and $\text{value}_{G^*}(V') \geq p$ we reach a contradiction. We conclude that $E_{G^*}(V') = \emptyset$. Then $E_G(V') = \emptyset$ and therefore V' is an independent set for G, with $|V'| \geq p = k$. We conclude that (G, k) is a yes-instance for Independent Set. ■ [63]

The next lemma presents a reduction from Coherence to Subset Choice (Lemma 6.3). The reduction in Lemma 6.3 is illustrated in Figure 6.2, and uses the strategy of *local replacement* (Garey & Johnson, 1979, cf. the reduction from Vertex Cover to Dominating Set presented in Chapter 3, page 37). The basic idea is as follows: Given an instance (N, c) for Coherence with network $N = (P, C)$, we define a graph $G = (V, E)$ with $V \supseteq P$ and $E \supseteq C$. For each constraint (u, v) in N we define a special substructure in G on edge $(u, v) \in E$; this substructure is of one form for positive constraints and of another form for negative constraints. The transformation ensures that (N, c) is a yes-instance for Coherence if and only if (G, p), with $p = c$, is a yes-instance for Subset Choice.

[63] Note that the proof of Lemma 6.2 also supports a reduction from G to G^* in which all hyperedge weights in G^* are set to $-|V|$ instead of $-|V|^2$.

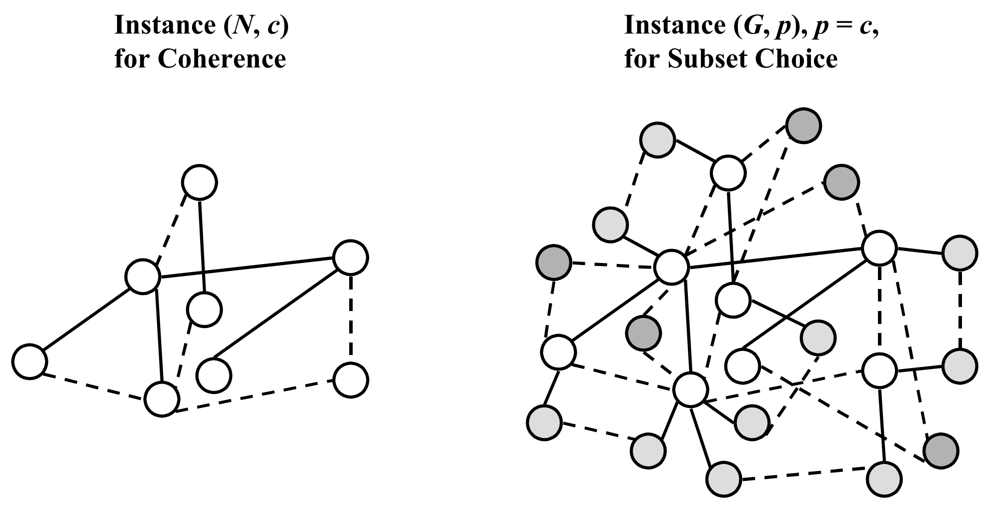


Figure 6.2. Illustration of the reduction in Lemma 6.3.
(Left) An instance (N, c) for Coherence. For simplicity, the example assumes that all constraints in N have weight '1'. Note that this special case of Coherence is also NP-hard (Corollary 5.3, page 84). The solid lines represent positive constraints in N, and the dotted lines represent negative constraints in N. (Right) The instance (G, p) for Subset Choice obtained on the transformation from (N, c) in Lemma 6.3. Here white and light gray vertices have weight '0' and dark gray vertices have weight '+1'; solid edges have weight '+1' and dotted edges have weight '−1'. Note that (N, c) is a yes-instance for Coherence if and only if (G, p) is a yes-instance for Subset Choice.

Lemma 6.3. Let network $N = (P, C)$, with weight function $w_C(.)$, and the positive integer c form an instance for Coherence. Then we define an instance for Subset Choice, consisting of a weighted graph $G = (V, E)$ and positive integer p, as follows.

(1) For each element $v \in P$ there is a corresponding vertex $v \in V$.

(2) For each positive constraint $(u, v) \in C^+$, there is a corresponding edge $(u, v) \in E$. Further we create in G a vertex x_{uv}, and we attach x_{uv} to both u and v with the edges (x_{uv}, u) and (x_{uv}, v). Finally we set vertex weights $w_V(u) = w_V(v) = 0$, and $w_V(x_{uv}) = w_C(u, v)$, and we set edge weights $w_E(u, v) = w_C(u, v)$, and $w_E(x_{uv}, u) = w_E(x_{uv}, v) = -w_C(u, v)$.

(3) For each negative constraint $(u, v) \in C^-$, there is a corresponding edge $(u, v) \in E$. Further we create in G two vertices x_u and x_v. We attach x_u to u with edge (x_u, u); we attach x_v to v with edge (x_v, v); and we attach x_u and x_v to each other with edge (x_u, x_v).

Finally we set vertex weights $w_V(u) = w_V(v) = w_V(x_u) = w_V(x_v) = 0$, and we set edge weights $w_E(u, v) = w_E(x_u, x_v) = - w_C(u, v)$, and $w_E(x_u, u) = w_E(x_v, v) = w_C(u, v)$.

(4) Finally, we set $p = c$.

Then (N, c) is a yes-instance for Coherence if and only if (G, p) is a yes-instance for Subset Choice.

Proof: $(\Rightarrow)$ Let (N, c) be a yes-instance for Coherence. Then there exists a partition $(A \cup R)^P$, with $\mathrm{Coh}_N(A, R) = \sum_{(u,v)\in S_N(A,R)} w_C(u, v) \geq c$. We consider the constraints in $S_N(A, R)$ one by one, and we build a set $V' \supseteq A$, $V' \subseteq V$, with $\mathrm{value}_G(V') \geq c$. Let $(u, v) \in S_N(A, R)$: (a) if $u, v \in A$, then let $u, v \in V'$; (b) if $u, v \in R$, then let $x_{uv} \in V'$; (c) if $u \in A$ and $v \in R$, then let $u, x_u \in V'$; (d) if $u \in R$ and $v \in A$, then let $v, x_v \in V'$. Note that in case (a) (u, v) is a positive constraint (otherwise it would not be in $S_N(A, R)$), and thus $\mathrm{value}_G(\{u, v\}) = w_V(u) + w_V(v) + w_V(x_{uv}) = w_V(x_{uv}) = w_C(u, v)$; in case (b) (u, v) is a positive constraint, and thus $\mathrm{value}_G(\{x_{uv}\}) = w_C(u, v)$; in case (c) (u, v) is a negative constraint, and thus $\mathrm{value}_G(\{u, x_u\}) = w_C(u, v)$; and, in case (d) (u, v) is a negative constraint, and thus $\mathrm{value}_G(\{v, x_v\}) = w_C(u, v)$. We conclude that $\mathrm{value}_G(V') = \mathrm{Coh}_N(A, R) \geq c = p$, and thus (G, p) is a yes-instance for Subset Choice.

$(\Leftarrow)$ Let (G, p) be a yes-instance for Subset Choice. Then there exists a subset $V' \subseteq V$ with $\mathrm{value}_G(V') \geq p$. Assume that the size of V' maximum. We consider the elements in P, and we build a partition $(A \cup R)^P$ as follows. Let $v \in P$. If $v \in V'$, then let $v \in A$ otherwise let $v \in R$. We now consider each edge $(u, v) \in C$: (a) Let $(u, v) \in C^+$. By the construction of G, the weight $w_C(u, v)$ counts towards $\mathrm{value}_G(V')$ if and only if $[u, v \in V'$ and $x_{uv} \notin V']$ or $[u, v \notin V'$ and $x_{uv} \in V']$, and thus $w_C(u, v)$ counts towards $\mathrm{Coh}_N(A, R)$ if and only if $[u, v \in V'$ and $x_{uv} \notin V']$ or $[u, v \notin V'$ and $x_{uv} \in V']$. (b) Let $(u, v) \in C^-$. By the construction of G, the weight $w_C(u, v)$ counts towards $\mathrm{value}_G(V')$ if and only if $[u, x_u \in V'$ and $v, x_x \notin V']$ or $[u, x_u \notin V'$ and $v, x_v \in V']$, and thus $w_C(u, v)$ counts towards $\mathrm{Coh}_N(A, R)$ if and only if $[u, x_u \in V'$ and $v, x_x \notin V']$ or $[u, x_u \notin V'$ and $v, x_v \in V']$. We conclude that $\mathrm{Coh}_N(A, R) = \mathrm{value}_G(V') = p \geq c$, and thus (N, c) is a yes-instance for Coherence. ■

6.4. Subset Choice on Unit-weighted Conflict Graphs

In the previous section we have seen that Subset Choice is NP-hard even for the special case where the value-structure can be modeled by a unit-weighted conflict graph, called UCG Subset Choice (Corollary 6.1). In this section we investigate the parameterized complexity of UCG Subset Choice for two different parameters. The first parameter is the natural parameter for UCG Subset Choice, the positive integer p. The second parameter is the relational parameter q, where q is defined as follows: Let (H, p), with $H = (V, E)$, be an instance for Subset Choice; then $q = p - \text{value}_H(V)$. As we can rewrite $p = \text{value}_H(V) + q$, the criterion q is naturally interpreted as the requirement that the value of the chosen subset V' should exceed the value of V by amount q.

First, in Section 6.4.1, we will show that p-UCG Subset Choice is not in FPT (unless FPT = W[1]). In Section 6.4.2, I explain in more detail how the relational parameter q for Subset Choice relates to the natural parameter p, by introducing a problem called Subset Rejection. Then, in Section 6.4.3, we will show that q-UCG Subset Choice is in FPT.

6.4.1. p-UCG Subset Choice is W[1]-hard

The following theorem shows that p-UCG Subset Choice is not in FPT (unless FPT = W[1]). The proof involves a reduction from the known W[1]-complete problem, k-Independent Set (Downey & Fellows, 1999).

Theorem 6.2. p-UCG Subset Choice $\notin$ FPT (unless FPT = W[1]).

Proof: Reconsider the proof of Lemma 6.1. Lemma 6.1 presents a polynomial-time reduction in which we transform any instance (G, k) for Independent Set to and instance (G^*, p) for Subset Choice, with G^* a unit weighted conflict graph and $p = k$. In other words, Lemma 6.1 presents a parametric reduction from k-Independent Set to p-UCG Subset Choice. Since, the problem k-Independent Set is known to be W[1]-complete, we conclude that p-UCG Subset Choice is W[1]-hard. ■

Since UCG Subset Choice is a special case of Subset Choice we conclude:

Corollary 6.2. p-Subset Choice $\notin$ FPT (unless FPT = W[1]).

Corollary 6.2 shows that the desire to obtain a satisfactorily large subset value (i.e., we want a value of at least p) is, in itself, not a crucial source of complexity in Subset Choice.

6.4.2. Subset Rejection and Parameter q

This section explains in more detail the relational parameterization of Subset Choice, when parameterized by $q = p - \text{value}_H(V)$. To facilitate thinking in terms of the parameter q (instead of p) we define a new value function on subsets of vertices in a weighted hypergraph: Let $H = (V, E)$ be a hypergraph and let $V' \subseteq V$ be a subset. Then the improvement in value of $V \backslash V'$, relative to the value of V, is called *rejection value* of V' and is defined as:

$$\begin{aligned}\text{reject}_H(V') &= \text{value}_H(V/V') - \text{value}_H(V) \\ &= \left(\sum_{v \in V \backslash V'} \text{w}_\text{V}(v) + \sum_{e \in \text{E}_H(V \backslash V')} \text{w}_\text{E}(e) \right) - \left(\sum_{v \in V} \text{w}_\text{V}(v) + \sum_{e \in \text{E}_H(V)} \text{w}_\text{E}(e) \right) \\ &= -1 \cdot \left(\sum_{v \in V'} \text{w}_\text{V}(v) + \sum_{e \in \text{R}_H(V')} \text{w}_\text{E}(e) \right)\end{aligned}$$

where $\text{R}_H(V') = \{(v_1, v_2, ..., v_h) \in E \mid v_1 \text{ or } v_2 \text{ or } ... \text{ or } v_h \in V'\}$.

Note that a vertex v has positive rejection-value if its weight *plus* the sum of the weights of its incident hyperedges is negative. In other words, a choice alternative has positive rejection-value if it strongly clashes with other choice alternatives in the set of available alternatives. More generally, a subset $V' \subseteq V$ has positive rejection-value if the sum of the values of its elements *plus* the sum of the weights of all hyperedges incident to a vertex $v \in V$ is negative. Thus, removing a rejection set V' with positive rejection-value from V entails the removal of negative value from H.

We can now state a new problem, called Subset Rejection:

Subset Rejection

Input: A *weighted* hypergraph $H = (V, E)$, $E \subseteq \bigcup_{2 \leq h \leq |V|} V^h$, for every $v \in V$ a weight $\text{w}_\text{V}(v) \in \mathbb{Z}$, for every $e \in E$ a weight $\text{w}_\text{E}(e) \in \mathbb{Z} \backslash \{0\}$, and a positive integer q.

Question: Does there exist a subset $V' \subseteq V$ such that $\text{reject}_H(V') \geq q$?

While the problem Subset Choice asks for the subset V' that we want to *choose*, the problem Subset Rejection asks for the subset V' that we want to *reject* (in the latter case $V \backslash V'$ is the subset that we want to choose). Note that there exists a subset $V' \subseteq V$ with $\text{reject}_H(V') \geq q = p - \text{value}_H(V)$ if and only if there exists a subset $V^* = V \backslash V'$ with $\text{value}_H(V^*) \geq p = q + \text{value}_H(V)$. Thus, the natural parameterization of Subset Rejection (denoted q-Subset Rejection) is a relational parameterization for Subset Choice (denoted $(p - \text{value}_H(V))$-Subset Choice or, simply, q-Subset Choice).

6.4.3. q-UCG Subset Choice is in FPT

Above we have seen that q-Subset Choice is a relational parameterization for Subset Choice. In this section, we consider the parameterized complexity of q-Subset Choice on unit-weighted conflict graphs, i.e., q-UCG Subset Choice. We prove the following theorem.

Theorem 6.3. q-UCG Subset Choice $\in$ FPT.

For simplicity, we work with the version Subset Rejection instead of Subset Choice. That is, we consider the problem as one of deciding whether or not a subset V' with $\text{reject}_H(V') \geq p - \text{value}_H(V) = q$ exists; instead of deciding whether or not a subset $V^* = V \backslash V'$ with $\text{value}_H(V^*) \geq p$ exists. Keep in mind, though, that the two conceptualizations are equivalent; i.e., the answer is "yes" for the one if and only if the answer is "yes" for the other.[64]

The proof of Theorem 6.3 is organized as follows. We start with two observations: The first is a general observation that holds for conflict hypergraphs (Observation 6.1), the second applies specifically to unit-weighted conflict graph (Observation 6.2). Using Observations 6.1 and 6.2, we define a branching rule, (UCG 1), that can be used to construct a bounded search tree for q-Subset Rejection, and thus also

[64] Although it is true that the *search* versions of Subset Choice and Subset Rejection ask for a different solution subset (i.e., if $V' \subseteq V$ is a solution for the one problem, then $V \backslash V'$ is a solution for the other), this difference is insubstantial for present purposes. Namely, the transformation from a set V' to its complement $V \backslash V'$ can be done in polynomial-time and thus does not affect the (classical and parameterized) complexity classification of Subset Choice/Subset Rejection. Further, since V is given as part of the input, one can build the sets V' and $V \backslash V'$ simultaneously by deleting a vertex v from V as soon as v is included in V'; the remaining vertices in V together form $V \backslash V'$.

for q-Subset Choice. Finally, we conclude an fpt-algorithm that solves q-Subset Choice in time $O(2^q|V|)$.

Observation 6.1 applies to general conflict hypergaphs. It shows that a vertex v with non-positive rejection-value (i.e., its weight plus the weights of its incident edges is positive), in a conflict hypergraph, never needs to be rejected (i.e., always can be chosen). Namely, if $\text{reject}_H(\{v\}) \leq 0$, then there always exists a subset V' with maximum rejection-value such that $v \notin V'$.

Observation 6.1. Let conflict hypergraph $H = (V, E)$ and positive integer q form an instance for Conflict Hypergraph (CH) Subset Rejection. Further, let $v \in V$ be a vertex such that $\text{reject}_H(\{v\}) \leq 0$. Then (H, q) is a yes-instance for CH Subset Rejection if and only if there exist a subset V' with $\text{reject}_H(V') \geq q$ and $v \notin V'$.

Proof: ($\Rightarrow$) Let (H, q) be a yes-instance for CH Subset Rejection. Then there exists a subset $V^* \subseteq V$ with $\text{reject}_H(V^*) \geq q$. We show that there exist a subset $V' \subseteq V^*$ with $\text{reject}_H(V') \geq \text{reject}_H(V^*) \geq q$ and $v \notin V'$. We distinguish two cases (1) Let $v \notin V^*$. Then $V' = V^*$ proves the claim. (2) Let $v \in V^*$. Then consider the subset $V^*\backslash\{v\}$. Since H is a conflict hypergraph (i.e., it has only negative edges and positive vertices) and $\text{reject}_H(\{v\}) \leq 0$, we know that $\text{reject}_H(V^*\backslash\{v\}) \geq \text{reject}_H(V^*) \geq q$. Then $V' = V^*\backslash\{v\}$ proves the claim. ($\Leftarrow$) Let (H, q) be an instance for CH Subset Rejection and let $V' \subseteq V$ with $\text{reject}_H(V') \geq q$ and $v \notin V'$. Then (H, q) is a yes-instance for CH Subset Rejection. ■

Observation 6.2 applies to unit-weighted conflict graphs. It shows that for every edge (u, v) in a unit-weighted conflict graph we may reject u or v. Namely, in that case, there always exists a subset V' with maximum rejection-value with at least one of u or v in V'.

Observation 6.2. Let $G = (V, E)$ and q form an instance for UCG Subset Rejection and let $(u, v) \in E$. Then (G, q) is a yes-instance for UCG Subset Rejection if and only if there exists $V' \subseteq V$ with $\text{reject}_G(V') \geq q$ and $u \in V'$ or $v \in V'$.

Proof: ($\Rightarrow$) Let (G, q) be a yes-instance for UCG Subset Rejection and let $(u, v) \in E$. Then there exist a subset $V^* \subseteq V$ with $\text{reject}_G(V^*) \geq q$. We show that there exist a subset $V' \supseteq V^*$ with $\text{reject}_G(V') \geq \text{reject}_G(V^*) \geq q$ such that $v \notin V'$. We distinguish two cases (1) Let $u \in V^*$ or $v \in V^*$. Then the $V' = V^*$ proves the claim. (2) Let $u, v \notin V^*$.

Then $(u, v) \notin R_G(V^*)$. Since $w_V(u) = 1$ and $w_E(u,v) = -1$ we can conclude $\text{reject}_G(V^* \cup \{u\}) \geq \text{reject}_G(V') + (-w_V(u) - w_E(u,v)) = \text{reject}_G(V') \geq q$. Then $V' = V^* \cup \{u\}$ proves the claim. ($\Leftarrow$) Let (G, q) be an instance for UCG Subset Rejection and let $V' \subseteq V$ with $\text{reject}_G(V') \geq q$ and $u \in V'$ or $v \in V'$. Then (G, q) is a yes-instance for UCG Subset Rejection. ■

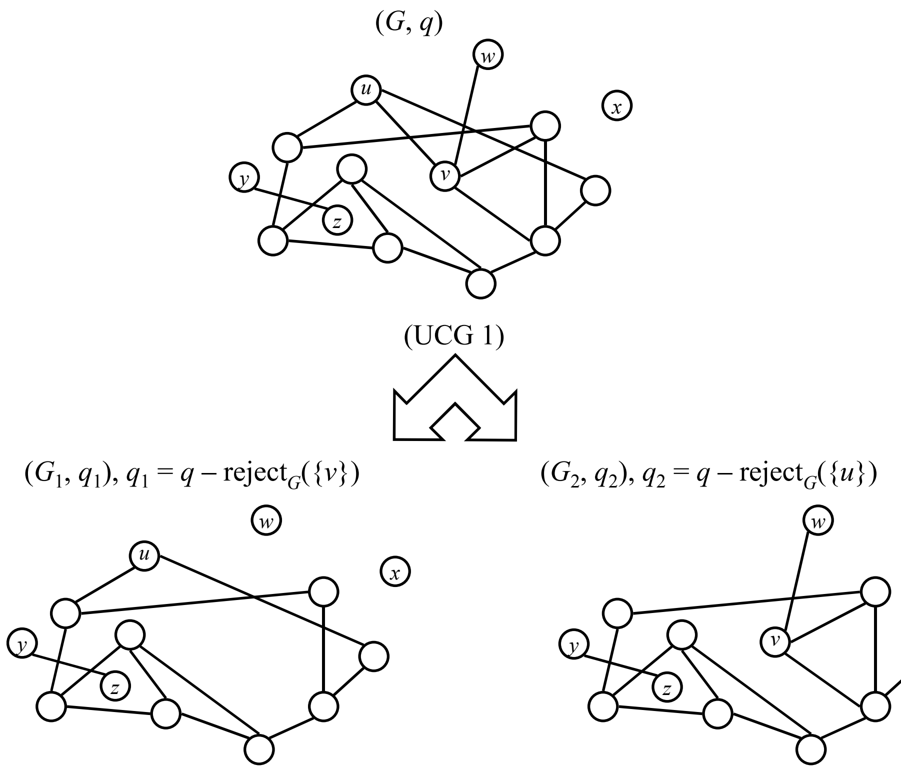
From Observation 6.2, we know that for every edge in a unit-weighted conflict graph we can include at least one of its endpoints in V' without loss in rejection-value. From Observation 6.1 we now that we only need to include endpoints if they have positive rejection-value in V'. The rule (UCG 1) uses these observations to branch on edges in a unit-weighted conflict graph.

(UCG 1) The Positive Endpoint Edge-Branching Rule. Let s be a search tree node labeled by an instance (G, q), $G = (V, E)$, for UCG Subset Rejection and let $(v_1, v_2) \in E$ with $\text{reject}_G(\{v_i\}) > 0$ for at least one vertex $v_i \in \{v_1, v_2\}$. Then for each $v_i \in \{v_1, v_2\}$ with $\text{reject}_G(\{v_i\}) > 0$ we create a child s_i of s and label it by (G_i, q_i), where $G_i = (V\backslash\{v_i\}, E\backslash R_G(\{v_i\}))$, $q_i = q - \text{reject}_G(\{v_i\})$.

Note that (UCG 1) only applies if there exists an edge (v_1, v_2) in G with $\text{reject}_G(\{v_1\}) > 0$ or $\text{reject}_G(\{v_2\}) > 0$. Thus application of (UCG 1) to a node in the search tree always leads to the creation of at least one child of that node. If only one of v_1 and v_2 has positive rejection-value then exactly one child is created, and if both v_1 and v_2 have positive rejection-value then exactly two children are created.

To prove that (UCG 1) is a valid branching rule for UCG Subset Rejection, we need to show that (G, q) is a yes-instance for UCG Subset Rejection if and only if at least one of the children of s is labeled by a yes-instance for UCG Subset Rejection.

Proof of (UCG 1): Let (G, q) be a yes-instance for UCG Subset Rejection and let $(v_1, v_2) \in E$ with $\text{reject}_G(\{v_i\}) > 0$ for at least one vertex $v_i \in \{v_1, v_2\}$. We distinguish two cases. (1) Let both v_1 and v_2 have positive rejection-value. Then application of rule (UCG 1) to edge (v_1, v_2) leads to the creation of two instances (G_1, q_1) and (G_2, q_2), where (G_1, q_1) represents the possibility that $v_1 \in V'$ and (G_2, q_2) represents the possibility that $v_2 \in V'$. From Observation 6.2 we know that there exists a subset $V' \subseteq V$ with maximum rejection-value such that $v_1 \in V'$ or $v_2 \in V'$. We conclude that (G, q) is a yes-instance for


(G, q)
w
u
x
v
y
z
(UCG 1)
(G1, q1), q1 = q – rejectG({v})
(G2, q2), q2 = q – rejectG({u})

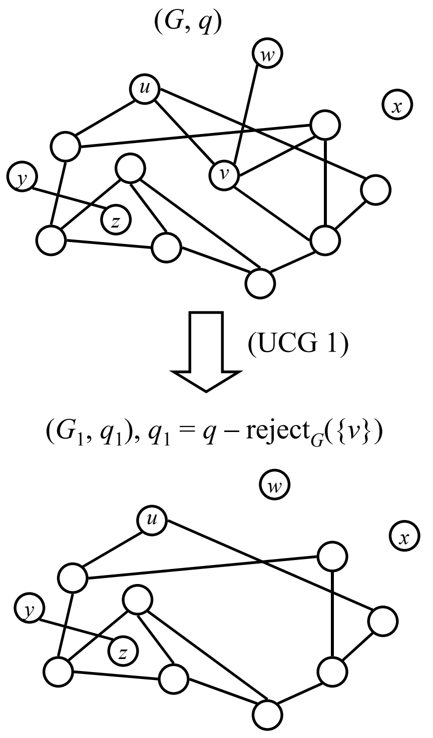


Figure 6.3. Illustration of branching rule (UCG 1) for UCG Subset Choice.
The rule (UCG 1) is applied to an instance (G, q) for UCG Subset Rejection. The top figure illustrates braching on an edge (u, v) in G where both endpoints u and v have positive rejection-value. The left figure illustrates branching on an edge (v, w), where only one endpoint, v, has positive rejection-value. In the first case two new instances (G_1, q_1) and (G_2, q_2) are created, while in the second case only one new instance (G_1, q_1) is created (in other words, in the latter case (USC 1) works as a reduction rule). Note that, in unit-weighted conflict graphs, vertices with non-positive rejection-value are singletons or pendant vertices (e.g. w, x, y, and z). This means that we cannot apply (UCG 1) if and only all vertices left in the graph are of this type. Note that including any of these vertices in a subset can never improve its rejection-value (see Observation 6.1).

UCG Subset Rejection if and only if (G_1, q_1) or (G_2, q_2) is a yes-instance for UCG Subset Rejection. (2) Let only one of v_1 and v_2 have positive rejection-value. W.l.o.g. let $\text{reject}_G(\{v_1\}) > 0$ and $\text{reject}_G(\{v_2\}) \leq 0$. Then application of rule (UCG 1) leads to the creation of only one instance (G_1, q_1) representing the assumption that $v_1 \in V'$. From Observation 6.2 we know that there exists a subset $V' \subseteq V$ with maximum rejection-value and $v_1 \in V'$ or $v_2 \in V'$. Further, from observation 6.1 we know that there exists a subset $V' \subseteq V$ with maximum rejection-value such that $v_2 \notin V'$. We conclude that (G, q) is a yes-instance for UCG Subset Rejection if and only if (G_1, q_1) or (G_2, q_2) is a yes-instance for UCG Subset Rejection. ■

With the following lemma we conclude an fpt-algorithm for q-UCG Subset Rejection that runs in time $O(2^q|V|)$.

Lemma 6.4. q-UCG Subset Rejection can be solved in time $O(2^q|V|)$.

Proof: We describe an fpt-algorithm for q-UCG Subset Rejection. The algorithm takes as input an instance (G, q) and creates a search tree T by recursively applying (UCG 1) to (G, q) until either an instance (G', q') is encountered with $q' \leq 0$ (in which case the algorithm returns the answer "yes") or (UCG 1) cannot be applied anymore. If the algorithm halts without returning the answer "yes" then we know that each leaf s_i in the search tree T is labeled by an instance (G_i, q_i), such that all vertices in G_i have non-positive rejection-value. Then, from Observation 6.1, we can conclude that (G, q) is a no-instance. We now prove that the algorithm halts in time $O(2^q|V|)$.

First, to apply (UCG 1) to an instance (G, q), $G = (V, E)$, we need to find a vertex $v \in V$ with $\text{reject}_G(\{v\}) \geq 1$. To find such a vertex we need to consider at most $|V|$ vertices. Further, whenever we consider a vertex that has non-positive rejection-value we spend no more than $O(1)$ time to compute its rejection-value. Hence, we can find a vertex with positive rejection-value (or know that none exists) in time $O(|V|)$. If we find a vertex v with $\text{reject}_G(\{v\}) \geq 1$, then we branch on any edge (v, w) incident to v. For each new search tree node s_i that we create we spend at most $O(|V|)$ time to label it by (G_i, q_i). Namely, we spend time $O(\text{deg}_G(v))$ to compute the value $\text{reject}_G(\{v\})$, and we need time $O(\text{deg}_G(v))$ to delete v and its adjacent edges from G. Thus each node in the search tree can be labeled in time $O(\text{deg}_G(v))$. Since $\text{deg}_G(v) \leq |V| - 1$, we conclude that $O(\text{deg}_G(v)) \leq O(|V|)$.

Second, observe that each application of (UCG 1) leads to the creation of at most 2 new branches in the search tree, and thus, fan(T) $\leq$ 2. Further, whenever (UCG 1) creates a node labeled by (G', q') for a parent labeled by (G, q) then $q' \leq q - 1$. Thus, we have depth(T) $\leq q$. We conclude that the size of the search tree is at most $O(2^q)$. Combined with the running time needed for applying (UCG 1) we can conclude the algorithm runs in time $O(2^q|V|)$. ■

Since, the parameterized problem q-UCG Subset Rejection is equivalent to q-Subset Choice, we also have:

Corollary 6.3. q-UCG Subset Choice can be solved in time $O(2^q|V|)$.
Since, $O(2^q|V|)$ is fpt-time for parameter q, Corollary 6.3 proves Theorem 6.3.

6.4.4. Improved Results for q-UCG Subset Choice

In the previous section we have shown how we can solve q-UCG Subset Choice in fpt-time $O(2^q|V|)$. The arguments we used to derive this result are intended to provide an easy to follow illustration. I remark that, with the use of different techniques and a better running-time analysis, it is possible to derive a much faster fpt-algorithm for q-UCG Subset Choice that runs in time $O(1.151^q + q|V|)$. This result follows from work by Ulrike Stege and myself on the parameterized problem p-Profit Cover (see e.g. Stege et al. 2002). Here I briefly sketch how our results obtained for p-Profit Cover translate to q-UCG Subset Choice.

Let $G = (V, E)$ be a unit-weighted conflict graph. Then for any subset $V' \subseteq V$, we have $\text{reject}_G(V') = -\left(\sum_{v \in V'} \text{w}_V(v) + \sum_{e \in E_G(V')} \text{w}_E(e)\right) = |R_G(V')| - |V'| = \text{profit}_{PC,G}(V')$. This means that ($G$, q) is a yes-instance UCG Subset Rejection if and only if (G, p), with $p = q$, is a yes-instance for Profit Cover. In other words, the parameterized problems q-UCG Subset Rejection and p-Profit Cover are equivalent. This means that any result for p-Profit Cover directly translates to q-UCG Subset Rejection, for $q = p$, and thus also to q-UCG Subset Choice.

In Stege et al. (2002), it is shown that p-Profit Cover has a problem kernel of size $|V| \leq 2p$ for connected input graphs, and a problem kernel of size $|V| \leq 3p - 3$ for disconnected input graphs. Hence we can conclude:

Corollary 6.4. q-UCG Subset Choice has a problem kernel of size $|V| \leq 2q$ for connected input graphs, and a problem kernel of size $|V| \leq 3q - 3$ for disconnected input graphs.

The kernelization procedure of Stege et al. (2002) runs in time $O(p|V|)$. From Corollaries 6.3 and 6.4 we conclude that q-UCG Subset Choice is solvable in time $O(2^q(3q - 3) + q|V|)$, which is $O(2^q q + q|V|)$. Since it is possible to re-kernelize the input after each application of branching rule (UCG 1), using an analysis technique by Neidermeier and Rossmanith (1999), we conclude the following improvement in the running time for q-UCG Subset Choice:

Corollary 6.5. q-UCG Subset Choice can be solved in time $O(2^q + q|V|)$.

Moreover, with the use of reduction rules, kernelization, a different branching rule, re-kernelization after each branching, and detailed case analyses, it is possible to conclude a bounded search tree for p-Profit Cover whose size is $O(1.151^p)$ (Stege et al., 2002).

Corollary 6.6. q-UCG Subset Choice can be solved in time $O(1.151^q + q|V|)$.

6.5. Generalizing q-UCG Subset Choice

Theorem 6.3 shows that if a decision-maker has a value-structure that can be represented by a unit-weighted conflict graph, and s/he aims to choose a subset with a value that is at least q more than value$_G(V)$, then the task is practically feasible for large $|V|$ as long as q is not too large. In this subsection we study to what extent this result generalizes to value-structures that form generalizations of the unit-weighted conflict graph. Specifically, we will consider Subset Choice on edge-weighted conflict graphs (ECG Subset Choice), on vertex-weighted conflict graphs (VCG Subset Choice), on conflict graphs (CG Subset Choice), and on conflict hypergraphs (CH Subset Choice). For problems Π and Π', let $\Pi' \subseteq \Pi$ denote that Π' is a special case of Π. Then we have UCG Subset Choice $\subseteq$ ECG Subset Choice $\subseteq$ CG Subset Choice $\subseteq$ CH Subset Choice; and also UCG Subset Choice $\subseteq$ VCG Subset Choice $\subseteq$ CG Subset Choice $\subseteq$ CH Subset Choice.

The investigation in Sections 6.5.1–6.5.4 takes the following form. Each subsection considers one of the aforementioned problems. For each considered problem Π we ask: Is q sufficient to capture the non-polynomial complexity inherent in Π? If the answer is “no,” we attempt to find a superset $\kappa \supseteq q$, such that κ-$\Pi \in$ FPT. In Section 6.6, we will review to what extent the analysis in this section has led to the identification of crucial sources of complexity as defined in Section 4.5.

6.5.1. q-ECG Subset Choice is in FPT

Here we show that q-ECG Subset Choice is in FPT. First note that Observation 6.1 also applies to ECG Subset Choice (viz., edge-weighted graphs are a special type of conflict hypergraphs). Further, Observation 6.2 for unit-weighted conflict graphs directly generalizes for edge-weighted conflict graphs, as shown in Observation 6.3.

Observation 6.3. Let $G = (V, E)$ and q form an instance for ECG Subset Rejection and let $(u, v) \in E$. Then (G, q) is a yes-instance for ECG Subset Rejection if and only if there exists $V' \subseteq V$ with $\text{reject}_G(V') \geq q$ and $u \in V'$ or $v \in V'$.

Proof: Analogous to the proof of Observation 6.2, with G being an edge-weighted conflict graphs instead of unit-weighted graph. ■

From Observations 6.1 and 6.3 we can conclude that the algorithm described for q-UCG Subset Choice also solves q-ECG Subset Choice in time $O(2^q|V|)$.

Corollary 6.7. q-ECG Subset Choice $\in$ FPT.

Corollary 6.7 shows that the presence of edge-weights in a conflict graph, in itself, does not add non-polynomial time complexity to subset choice on conflict-graphs over and above the non-polynomial time complexity already captured by q.

6.5.2. q-VCG Subset Choice is W[1]-hard

Here we show that, although q is sufficient for capturing the non-polynomial time complexity in ECG Subset Choice, the same is not true for VCG Subset Choice (unless FPT = W[1]).

Theorem 6.4. q-VCG Subset Choice $\notin$ FPT (unless FPT = W[1]).

To prove Theorem 6.4, we present a parametric reduction from k-Independent Set to q-VCG Subset Choice in Lemma 6.5 (see Figure 6.4 for an illustration). For simplicity, in this reduction we assume that the input graph for k-Independent Set is of minimum degree 1 (i.e., G contains no singletons). Since k-Independent Set (with or without singletons) is known to be W[1]-complete (Downey & Fellows, 1999) the reduction shows that q-VCG Subset Choice is W[1]-hard.

Lemma 6.5. Let (G, k), $G = (V, E)$, be an instance for k-Independent Set, where G contains no singletons. We build an instance (G^*, q), with $G^* = (V^*, E^*)$, for q-VCG Subset Choice as follows. G^* has the same edges and vertices as G but is a vertex-weighted conflict graph. Therefore let $V^* = V$ and $E^* = E$. We define the weights for G^* as follows: for every $v \in V^*$ let $w_V(v) = \deg_{G^*}(v)-1$ and for every $e \in E^*$ let $w_E(e) = -1$. Note that $\text{reject}_{G^*}(\{v\}) = 1$ for all $v \in V^*$. Furthermore let $q = k$. Then G has an independent set of size at least k if and only if there exists $V' \subseteq V^*$ with $\text{reject}_{G^*}(V') \geq k$.

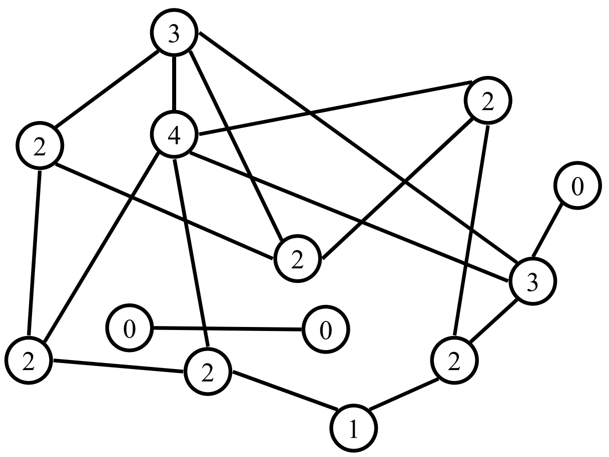


Figure 6.4. Illustration of the reduction in Lemma 6.5.
The reduction transforms G into G^*. The unweighted version of the figure represents graph G, and the weighted version of the figure represents the graph G^*. Only the vertex weights are shown for G^*; since G^* is a vertex-weighted conflict graph, all edge weights are set '-1.' Note that $w_V(v) = \deg_G(v) - 1$ for each vertex v in G^*. This property ensures that G has an independent set of size k if and only if there exists a subset $V' \subseteq V$, with $\text{reject}_{G^*}(V') \geq q = k$.

Proof of Lemma 6.5: ($\Rightarrow$) Let $V' \subseteq V$, $V' = \{v_1, v_2, ..., v_k\}$, be an independent set for G. This means that no two vertices $v_i, v_j \in \{v_1, v_2, ..., v_k\}$ share an edge in G and since $V = V^*$ also no two vertices $v_i, v_j \in \{v_1, v_2, ..., v_k\}$ share an edge in G^*. Thus $\text{reject}_{G^*}(V') = \text{reject}_{G^*}(\{v_1\}) + \text{reject}_{G^*}(\{v_2\}) + ... + \text{reject}_{G^*}(\{v_k\}) = k$.

($\Leftarrow$) Let $V' \subseteq V^*$ with $\text{reject}_{G^*}(V') \geq q$. We show G has an independent set of size at least q. We distinguish two cases: (1) If V' is an independent set for G^* then V' is an independent set for G. Assume $V' = \{v_1, v_2, ..., v_k\}$. Then $\text{reject}_{G^*}(V') = \text{reject}_{G^*}(\{v_1\}) + \text{reject}_{G^*}(\{v_2\}) + ... + \text{reject}_{G^*}(\{v_k\}) = q$, and thus $k = q$. (2) If V' is not an independent set for G^*, we transform V' into an independent set V'' for G^* with the algorithm described in the proof of Lemma 6.1 (page 123). Note that line 4 of the algorithm always results in the removal of at most $\deg_{G^*}(v) - 1$ edges from $R_{G^*}(V')$. Hence, in line 4, $\text{reject}_{G^*}(V''\backslash\{v\}) \geq \text{reject}_{G^*}(V'') - w_V(v) - (\deg_{G^*}(v) - 1) = \text{reject}_{G^*}(V'') - (\deg_{G^*}(v) - 1) + (\deg_{G^*}(v) - 1) = \text{reject}_{G^*}(V'')$, and thus in line 6, $\text{reject}_{G^*}(V'') \geq q$. Furthermore, when the algorithm halts V'' is an independent set for G^* and therefore for G. Thus case (1) applies to $V' = V''$. ■

I comment that the above reduction runs in polynomial-time, and thus Lemma 6.5 also constitutes a proof that Subset Choice is NP-hard (Theorem 6.1). Further, since VCG Subset Choice is a special case of CG Subset Choice, which in turn is a special case of CH Subset Choice, we also conclude:

Corollary 6.8. q-CG Subset Choice $\notin$ FPT (unless FPT = W[1]).

Corollary 6.9. q-CH Subset Choice $\notin$ FPT (unless FPT = W[1]).

6.5.3. $\{q, \Omega_V\}$-CG Subset Choice is in FPT

We consider another parameter for Subset Choice: The maximum vertex weight, denoted by Ω_V. Note that for every instance (H, q), $H = (V, E)$, for Subset Choice, there exist a positive integer value Ω_V such that for all $v \in V$, $w_V(v) \leq \Omega_V$. Thus, Ω_V is an implicit parameter for Subset Choice. In the following we will show that, although q-CG Subset Choice is W[1]-hard (Corollary 6.8), the parameter set $\{q, \Omega_V\}$ is sufficient for capturing the non-polynomial time complexity inherent in CG Subset Choice.

Theorem 6.5. $\{q, \Omega_V\}$-CG Subset Choice $\in$ FPT

The proof of Theorem 6.5 is organized as follows. First we observe that for every vertex in a conflict graph with positive rejection-value we can always either reject that vertex or reject at least one of its neighbors (Observation 6.4 below). Using Observation 6.4 and Observation 6.1 (see page 131) we define a branching rule that can be used to construct an fpt-algorithm for $\{q, \Delta\}$-CG Subset Choice (here Δ denotes the maximum vertex degree). Then we show that there exists a function $f(\Omega_V, q)$ such that $\Delta \leq f(\Omega_V, q)$. This allows us to conclude the existence of an fpt-algorithm for $\{q, \Omega_V\}$-CG Subset Choice.

Observation 6.4 shows that, for an instance (G, q) for CG Subset Choice and a vertex v with positive rejection-value, if a subset $V' \subseteq V$ has maximum rejection-value and $v \notin V'$ then v has at least one neighbor $u \in V$ with $u \in V'$.

Observation 6.4. Let G and q form an instance for CG Subset Choice and let $v \in V$ with $\text{reject}_G(v) > 0$. Then G and q form a yes-instance for CG Subset Choice if and only if there exists $V' \subseteq V$ with $\text{reject}_G(V') \geq q$, and at least one vertex $u \in N_G[v]$, with $u \in V'$.

Proof: ($\Rightarrow$) Let (G, q) be a yes-instance for CG Subset Choice and let $v \in V$ with $\text{reject}_G(v) > 0$. Then there exists $V^* \subseteq V$ with $\text{reject}_G(V^*) \geq q$. We show there exists $V' \subseteq V$ such that $\text{reject}_G(V') \geq \text{reject}_G(V^*) \geq q$ and at least one vertex $u \in N_G[v]$, with $u \in V'$. We distinguish two cases: (1) There exists a vertex $u \in N_G[v]$ with $u \in V^*$. Then $V' = V^*$ proves the claim. (2) There does not exist $u \in V^*$ with $u \in N_G[v]$. Then $v \notin V^*$, since $v \in N_G[v]$. Further, since $\text{reject}_G(\{v\}) \geq 1$ we know that $\text{reject}_G(V^* \cup \{v\}) \geq \text{reject}_G(V^*) + \text{reject}_G(\{v\}) \geq \text{reject}_G(V^*) + 1 > q$, and thus $V' = V^* \cup \{v\}$ proves the claim. ($\Leftarrow$) Let G and q form an instance for CG Subset Choice and let $V' \subseteq V$ be any subset with $\text{reject}_G(V') \geq q$. Then (G, q) is a yes-instance for CG Subset Choice. ■

From Observation 6.1 (page 131) we know we can exclude vertices with non-positive rejection-value from consideration for a solution subset for CG Subset Rejection. From Observation 6.4, we know that a subset with maximum rejection-value for a vertex-weighted conflict graph contains at least one vertex in $N_G[v]$ for each v in G with positive rejection-value. The rule (CG 1) uses these observations to branch on vertices with positive rejection-value in a vertex-weighted conflict graph (refer to Figure 6.5 for an illustration). Note how (CG 1) allows us to construct a bounded search tree for $\{q, \Delta\}$-CG Subset Choice (cf. branching rule (IS 1) for Independent Set discussed in Chapter 4, page 70).

(CG 1) The Positive Vertex-or-At-Least-One-Neighbor Branching Rule #1. Let s be a search tree node labeled by an instance (G, q), $G = (V, E)$ for CG Subset Choice and let $v \in V$, with $\text{reject}_G(\{v\}) > 0$ and $N_G(v) = \{v_1, ..., v_k\}$, $k \leq \Delta$. Then for each $v_i \in N_G[v]$ with $\text{reject}_G(\{v_i\}) > 0$ we create a child s_i of s and label it by (G_i, q_i), $G_i = (V \backslash \{v_i\}, E \backslash R_G(\{v_i\}))$, $q_i = q - \text{reject}_G(\{v_i\})$.

Proof: Let (G, q) be a yes-instance for CG Subset Rejection and let $v \in V$, with $\text{reject}_G(\{v\}) > 0$, $\text{N}_G(v) = \{v_1, ..., v_k\}$. Application of (CG 1) results in the creation of an instance (G_u, q_u) for each $u \in U$, with $U = \{v_i \in \text{N}_G[v] : \text{reject}_G(\{v_i\}) > 0\}$. Here, each instance (G_u, q_u) represents the assumption that $u \in V'$. From Observation 6.4 we know that there exists a subset $V' \subseteq V$ with maximum rejection-value and $V' \cap U \neq \emptyset$. Further, from observation 6.1 we know that there exists a subset $V' \subseteq V$ with maximum rejection-value and $v_i \notin V'$ for all $v_i \in \text{N}_G[v]$ with $\text{reject}_G(\{v_i\}) \leq 0$. We conclude that (G, q) is a yes-instance for CG Subset Rejection if and only if at least one of (G_u, q_u) is a yes-instance for CG Subset Rejection. ■

Application of (CG 1) to a search tree node s leads to the creation of at most $\text{deg}_G(v) + 1 \leq \Delta + 1$ children of s. Further for each newly created instance (G_i, q_i), we have $|G_i| \leq |G|$, $\Delta_i \leq \Delta$, $q_i \leq q - 1$. Thus we can use (CG 1) to build a search tree with $\text{fan}(T) \leq \Delta + 1$ and $\text{depth}(T) \leq q$.

Lemma 6.6. $\{\Delta, q\}$-CG Subset Rejection can be solved in time $O((\Delta + 1)^q|V|)$.

Proof: Analogous to the proof of Lemma 6.5, we define an fpt-algorithm for $\{\Delta, q\}$-CG Subset Rejection: The algorithm takes as input an instance (G, q) and recursively applies (CG 1) to some vertex v in G with $\text{reject}_G\{v\} > 0$ until either a "yes"-answer is returned or (CG 1) cannot be applied anymore. If the algorithm halts without returning the answer "yes" then we know that each leaf s_i in the search tree T is labeled by an instance (G_i, q_i), such that all vertices in G_i have non-positive rejection-value. Then, from Observation 6.1, we can conclude that (G, q) is a no-instance.

We next prove that the algorithm halts in time $O(2^q|V|)$. To find a vertex v to branch on (or know that none exists), and label a new node in the search tree, we need at most in time $O(|V|)$. Since each application of (CG 1) leads to the creation of at most $\Delta + 1$ new branches in the search tree, we know that $\text{fan}(T) \leq \Delta + 1$. Further, whenever (CG 1) creates a node labeled by (G', q'), for a parent labeled (G, q), then $q' \leq q - 1$, and thus $\text{depth}(T) \leq q$. We conclude that the size of the search tree is at most $O((\Delta + 1)^q)$. Combined with the time spent per node of the search tree we conclude that the algorithm runs in time $O((\Delta + 1)^q|V|)$. ■

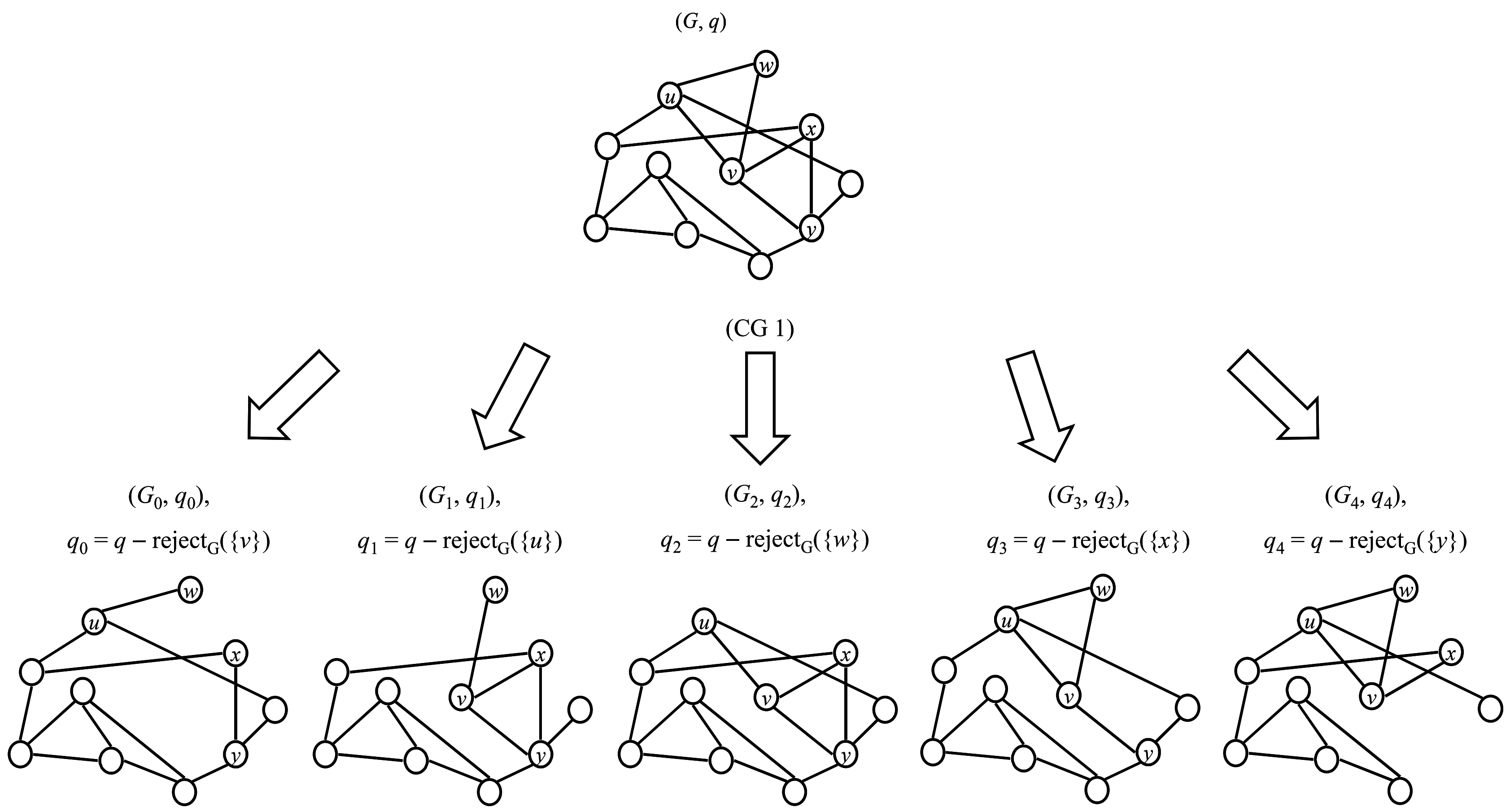

(G, q)
(CG 1)
(G_0, q_0), $q_0 = q - \text{reject}_G(\{v\})$
(G_1, q_1), $q_1 = q - \text{reject}_G(\{u\})$
(G_2, q_2), $q_2 = q - \text{reject}_G(\{w\})$
(G_3, q_3), $q_3 = q - \text{reject}_G(\{x\})$
(G_4, q_4), $q_4 = q - \text{reject}_G(\{y\})$
u
v
w
x
y

Figure 6.5. Illustration of branching rule (CG 1). See next page for description.

Figure 6.5. Illustration of branching rule (CG 1).
Here (CG 1) is applied to an instance (G, q) for CG Subset Rejection. The instances obtained after branching rule application on vertex v in G, are denoted (G_i, q_i) with $i = 0$, 1, 2, …, $\deg_G(v)$. For clarity, the vertex and edge weights are not depicted for; but the reader may assume that, in this example, each of the vertices u, v, w, x, and y, has positive rejection-value in G. The rule (CG 1) assumes that v or at least of its neighbors u, w, x, and y can be included in V'. In this example, $\deg_G(v) = 4$, thus (at most) five new instances are created by (CG 1).

The following lemma shows that for any instance (G, q) for CG Subset Rejection we can bound the vertex degree by a function $f(\Omega_V, q)$.

Lemma 6.7. Let (G, q) be an instance for CG Subset Rejection and let Ω_V be the maximum vertex weight in G. If there exists a vertex $v \in V$ with $\deg_G(v) \geq q + \Omega_V$ then (G, q) is a yes-instance.

Proof: We know for every $v \in V$, $\text{reject}_G(\{v\}) = -\left(w_V(v) + \sum_{e \in R_G(\{v\})} w_E(e)\right) \geq -\Omega_V +$ $\deg_G(v)$. Let $v \in V$ be a vertex with $\deg_G(v) \geq q + \Omega_V$. Then $\text{reject}_G(\{v\}) \geq q$ and thus (G, q) is a yes-instance for CG Subset Rejection. ■

From Lemma 6.7 we conclude a refinement of the search tree algorithm described above for Lemma 6.6: As soon as we encounter a node labeled by (G, q), such that $\Delta \geq q + \Omega_V$, we terminate the search and return the answer "yes." This way we ensure that for the resulting bounded search tree T, $\text{fan}(T) \leq \Delta + 1 \leq q + \Omega_V$. We conclude the following corollary.

Corollary 6.10. $\{q, \Omega_V\}$-CG Subset Choice can be solved in time $O((q + \Omega_V)^q|V|)$.

Since time $O((q + \Omega_V)^q|V|)$ is fpt-time for $\{q, \Omega_V\}$-CG Subset Choice, Corollary 6.10 proves Theorem 6.5. Note that, since VCG Subset Choice is a special case of CG Subset Choice, all results discussed above for CG Subset Choice also apply to VCG Subset Choice.

6.5.4. $\{q, \Omega_V, \varepsilon\}$-CH Subset Choice is in FPT

Here we consider Subset Choice on general conflict hypergraphs. Recall that a hypergaph is a generalization of a graph (Section 6.1, page 116). Like a conflict graph has positive weighted vertices and negative weighted edges, so a conflict hypergaph has positive weighted vertices and negative weighted hyperedges.

In this section we prove the following theorem.

Theorem 6.6. $\{q, \varepsilon, \Omega_V\}$-CH Subset Choice is in FPT

Recall that ε denotes the maximum span in a hypergraph. Since, for every hypergraph $H = (V, E)$, there exists a positive integer ε such that span(e) $\leq \varepsilon$ for every $e \in E$, the integer ε is an implicit parameter for CH Subset Choice.

The proof of Theorem 6.6 is organized as follows. First, we observe that Observation 6.4 for conflict graphs directly generalizes to conflict hypergraphs (Observation 6.5). Then we consider a new input parameter θ, denoting the maximum neighborhood of any vertex in H (i.e., θ is a positive integer, such that for every v in $|N_H(v)| \leq \theta$). Using Observation 6.1 and 6.4, we derive a branching rule (CH 1) that can be used to construct an fpt-algorithm for $\{q, \theta\}$-CH Subset Rejection. We will show that there exists a function $f(\varepsilon, \Delta)$ with $\theta \leq f(\varepsilon, \Delta)$. This allows us conclude an fpt-algorithm for $\{q, \varepsilon, \Delta\}$-CH Subset Rejection. Finally, we show that there exists a function $g(\Omega_V, q)$, with $\Delta \leq g(\Omega_V, q)$, and conclude an fpt-algorithm for $\{q, \varepsilon, \Omega_V\}$-CH Subset Choice.

Note that the proof of Observation 6.5 is the same as the proof of Observation 6.4, using as instance (H, q), such that H is a conflict hypergraph instead of conflict graph.

Observation 6.5. Let H and q form an instance for CH Subset Choice and let $v \in V$ with reject$_H(v) > 0$. Then H and q form a yes-instance for CH-Subset Choice if and only if there exists $V' \subseteq V$, with reject$_H(V') \geq q$, and at least one vertex $u \in N_H[v]$, with $u \in V'$.

Proof: ($\Rightarrow$) Let (H, q) be a yes-instance for CH Subset Choice and let $v \in V$ with reject$_H(v) > 0$. Then there exists $V^* \subseteq V$ with reject$_H(V^*) \geq q$. We show there exists $V' \subseteq V$ such that reject$_H(V') \geq$ reject$_H(V^*) \geq q$ and at least one vertex $u \in N_H[v]$, with $u \in V'$. We distinguish two cases: (1) There exists a vertex $u \in N_H[v]$ with $u \in V^*$. Then $V' = V^*$ proves the claim. (2) There does not exist $u \in V^*$ with $u \in N_H[v]$. Then $v \notin V^*$, since $v \in N_H[v]$. Further, since reject$_H(\{v\}) \geq 1$ we know that reject$_H(V^* \cup \{v\}) \geq$ reject$_H(V^*)$ + reject$_H(\{v\}) \geq$ reject$_H(V^*) + 1 > q$, and thus $V' = V^* \cup \{v\}$ proves the claim. ($\Leftarrow$) Let H

and q form an instance for CH Subset Choice and let $V' \subseteq V$ be any subset with $\text{reject}_H(V') \geq q$. Then (H, q) is a yes-instance for CH Subset Choice. ■

From Observations 6.1 and 6.5 we conclude a branching rule (CH 1) for CH Subset Choice that allows us to construct a bounded search tree T, with $\text{size}(T) \leq f(q, \theta)$. Note that (CH 1) is identical to (CG 1) with the exception that it takes as input a conflict hypergraph instead of a conflict graph.

(CH 1) The Positive Vertex-or-At-Least-One-Neighbor Branching Rule #2. Let s be a search tree node labeled by an instance (H, q), $H = (V, E)$ for CH Subset Rejection and let $v \in V$, with $\text{reject}_H(\{v\}) > 0$ and $N_H(v) = \{v_1, ..., v_k\}$, $k \leq \theta$. Then for each $v_i \in \{v, v_1, ..., v_k\}$ with $\text{reject}_H(\{v_i\}) > 0$ we create a child s_i of s and label it by (H_i, q_i), $H_i = (V\backslash\{v_i\}, E\backslash R_H(\{v_i\}))$, $q_i = q - \text{reject}_H(\{v_i\})$.

Proof: Analogous to the proof of (CG 1), using as instance (H, q), such that H is a conflict hypergraph instead of a conflict graph, and using Observation 6.5 instead of Observation 6.4. ■

We now show how (CH 1) can be used to define an fpt-algorithm for $\{q, \theta\}$-CH Subset Rejection.

Lemma 6.8. $\{q, \theta\}$-CH Subset Rejection can be solved in time $O((\theta +1)^q|V|^2)$.

Proof: Analogous to the proofs of Lemmas 6.5 and 6.6, we define an fpt-algorithm for $\{q, \theta\}$-CH Subset Rejection. The algorithm takes as input an instance (H, q) and recursively applies (CH 1) to a vertex v in H until either a "yes"-answer is returned or (CH 1) cannot be applied anymore (in which case, by Observation 6.1, we return the answer "no").

We can find a vertex v to branch on (or know that none exists), and label a new node in the search tree, in time $O(|V|^2)$ (the labeling can no longer be done in linear time, because a vertex in a hypergraph may have as many as $((|V| - 1)(|V| - 2))/2$ incident hyperedges). Since (CH 1) creates a bounded search tree T with $\text{fan}(T) \leq \theta + 1$ and $\text{depth}(T) \leq q$, we conclude that $\text{size}(T) \leq O((\theta +1)^q)$. In sum, we can decide $\{q, \theta\}$-CH CH Subset Rejection in time $O((\theta + 1)^q\,|V|^2)$. ■

Note that unlike $N_G(v)$, $N_H(v)$ may be larger than $\text{deg}_H(v)$, and thus θ is not bounded by a function $f(\Delta)$ in hypergraphs. Since $\text{span}_H(e) \leq \varepsilon$ for all $e \in E$, we do know that, for every $v \in V$, $|N_H(v)| \leq (\varepsilon - 1)\deg_H(v)$, and thus $\theta \leq (\varepsilon - 1)\Delta$. This observation allows us to conclude the following lemma.

Corollary 6.11. $\{q, \varepsilon, \Delta\}$-CH Subset Rejection can be solved in time $O(((\varepsilon - 1)\Delta + 1)^q |V|^2)$.

To show that Δ is bounded by some function $g(\Omega_V, q)$, we observe that Lemma 6.7 for conflict graphs generalizes directly to Lemma 6.9 for conflict hypergraphs.

Lemma 6.9. Let (H, q), with $H = (V, E)$, be an instance for CH Subset Rejection and let Ω_V be the maximum vertex weight in H. If there exists a vertex $v \in V$ with $\deg_H(v) \geq q + \Omega_V$, then (H, q) is a yes-instance.

Proof: We know for every $v \in V$, $\text{reject}_H(\{v\}) = -1 \cdot \left(w_V(v) + \sum_{e \in R_H(\{v\})} w_E(e) \right) \geq$ $-\Omega_V + \deg_H(v)$. Let $v \in V$ be a vertex with $\deg_H(v) \geq q + \Omega_V$. Then $\text{reject}_H(\{v\}) \geq q$ and thus (H, q) is a yes-instance for CH Subset Rejection. ■

Using Lemma 6.9 we can refine the algorithm in Lemma 6.8: We terminate the search as soon as we encounter an instance (G, q) with $\Delta \geq q + \Omega_V$ and return the answer "yes." This ensures that for the resulting search tree T, $\text{fan}(T) \leq q + \Omega_V - 1$, and we conclude the following corollary:

Corollary 6.12. $\{q, \varepsilon, \Omega_V\}$-CH Subset Choice can be solved in time $O(((\varepsilon - 1)(q + \Omega_V - 1) + 1)^q |V|^2)$.

Since time $O(((\varepsilon - 1)(q + \Omega_V - 1) + 1)^q |V|^2)$ is fpt-time for $\{q, \varepsilon, \Omega_V\}$-CH Subset Choice, Corollary 6.12 proves Theorem 6.6.

6.6. Crucial Sources of Complexity

In Sections 6.4 and 6.5 we have investigated the parameterized complexity of different parameterizations of Subset Choice on conflict hypergraphs. Many subset choice problems that arise in practice may be of this type. To illustrate, consider again the consumer that has to decide on a set of pizza toppings: Assume a person p likes pepperoni (which has, say, value $x > 0$ for p) and likes mushrooms (value $y > 0$). Let z denote the value for of having both pepperoni and mushrooms on a pizza. Even if p think that pepperoni and mushrooms taste well together, we may nevertheless have $z < x + y$

(cf. the notion ‘diminishing marginal utility’ in economic theory; e.g. Parkin & Bade, 1997). Loss in value due to combining alternatives may also happen when alternatives represent different activities or events that conflict in space and/or time (see e.g. footnote 11 on page 11) or when alternatives share some of their value-supporting features (cf. Figure 6.1).

In this section, we reconsider some of the results obtained in the previous sections, with the aim of identifying crucial sources of complexity in Subset Choice for value-structures that are conflict hypergraphs. We start by reconsidering the result that $\{q, \varepsilon, \Omega_V\}$-CH Subset Choice $\in$ FPT (Corollary 6.12). Is the parameter set $\{q, \varepsilon, \Omega_V\}$ a crucial source of complexity for CH Subset Choice? Recall from Section 4.5 (page 73) that we can conclude that $\{q, \varepsilon, \Omega_V\}$ is a crucial source of complexity for CH Subset Choice if and only if κ-CH Subset Choice $\notin$ FPT for every $\kappa \subset \{q, \varepsilon, \Omega_V\}$. In other words, we need to know if $\{q, \varepsilon, \Omega_V\}$ is a *minimal* parameter set for CH Subset Choice. What do we know about the parameterized complexity of κ-CH Subset Choice for different $\kappa \subset \{q, \varepsilon, \Omega_V\}$? To answer this question, in the following we first reconsider Theorem 6.2 (page 128) and then Theorem 6.4 (page 137).

Theorem 6.2 states that p-UCG Subset Choice is not in FPT (unless FPT = W[1]). We next show how this theorem implies the fixed-parameter intractability of κ-Subset Choice for the parameter set $\kappa = \{p, \varepsilon, \omega_V, \Omega_V, \omega_E, \Omega_E\}$. Here, p, ε and Ω_V are defined as before (i.e., the positive integer in the input, the maximum span, and the maximum vertex weight). The new parameters ω_V, ω_E, Ω_E are defined as follows: Let $H = (V, E)$ be a weighted hypergraph, Then $\omega_V \geq 0$ denotes the absolute size of the most negative vertex weight, $\omega_E \geq 1$ denotes the absolute size of the most negative hyperedge weight, and $\Omega_E \geq 1$ denotes the largest positive hyperedge weight. Note that for every instance $H = (V, E)$ for Subset Choice, there exist values ω_V, Ω_V, ω_E, and Ω_E such that for all $v \in V$, $-\omega_V \leq w_V(v) \leq \Omega_V$ and for all $e \in E$, $-\omega_E \leq w_E(e) \leq \Omega_E$. Hence, ω_V, Ω_V, ω_E, and Ω_E are all implicit parameters for Subset Choice.

For UCG Subset Choice the input hypergraph is a unit-weighted conflict graph. In other words, in UCG Subset Choice, the values ε, ω_V, Ω_V, ω_E, Ω_E are all constants;

specifically we have $\varepsilon = 2$, $\omega_V = 0$, $\Omega_V = 1$, $\omega_E = 1$, $\Omega_E = 1$. This means that we can conclude the following corollary from Theorem 6.2.

Corollary 6.13. $\{p, \varepsilon, \omega_V, \Omega_V, \omega_E, \Omega_E\}$-CH Subset Choice $\notin$ FPT (unless FPT = W[1]).

Proof: The proof is by contraction. Assume $\{p, \varepsilon, \omega_V, \Omega_V, \omega_E, \Omega_E\}$-CH Subset Choice $\in$ FPT and FPT $\neq$ W[1]. Then there exists an algorithm solving Subset Choice in time $O(f(p, \varepsilon, \omega_V, \Omega_V, \omega_E, \Omega_E)n^{\alpha})$. In UCG Subset Choice we have $\varepsilon = 2$, $\omega_V = 0$, $\Omega_V = 1$, $\omega_E = 1$, $\Omega_E = 1$, and thus we can solve UCG Subset Choice in time $O(f(p, \varepsilon, \omega_V, \Omega_V, \omega_E, \Omega_E)n^{\alpha}) = O(f(p, 2, 0, 1, 1, 1)n^{\alpha}) = O(f(p)n^{\alpha})$. This means that p- UCG Subset Choice $\in$ FPT. But then, from Theorem 6.2, we can conclude that FPT = W[1]. ■

Recall from Section 4.5 (page 73) that, for a problem Π and parameter set κ, if κ-$\Pi \notin$ FPT then κ'-$\Pi \notin$ FPT for all $\kappa' \subseteq \kappa$. Thus, from Corollary 6.13 we conclude that κ-Subset Choice $\notin$ FPT, for all $\kappa \subseteq \{p, \varepsilon, \omega_V, \Omega_V, \omega_E, \Omega_E\}$ (unless FPT = W[1]). In other words, we can conclude from Corollary 6.13 that κ-CH Subset Choice $\notin$ FPT, for $\kappa = \{\varepsilon\}$, $\kappa = \{\Omega_V\}$, and $\kappa = \{\varepsilon, \Omega_V\}$ (unless FPT = W[1]).

We now reconsider Theorem 6.4. This theorem states that q-VCG Subset Choice is not in FPT (unless FPT = W[1]). Since VCG Subset Choice is a special case of CH Subset Choice, such that $\varepsilon = 2$, $\omega_V = 0$, $\omega_E = 1$, $\Omega_E = 1$, we conclude the following corollary.

Corollary 6.14. $\{q, \varepsilon, \omega_V, \omega_E, \Omega_E\}$-UC Subset Choice $\notin$ FPT (unless FPT = W[1]).

Proof: Analogous to the proof of Corollary 6.13.

From Corollary 6.14, we conclude that κ-CH Subset Choice $\notin$ FPT, for $\kappa = \{q\}$, and $\kappa = \{q, \varepsilon\}$ (unless FPT = W[1]).

In sum, from the above analyses, we know that κ-CH Subset Choice is not in FPT (unless FPT = W[1]) for five of the six possible $\kappa \subset \{q, \varepsilon, \Omega_V\}$. The only proper subset that was not considered is $\kappa = \{q, \Omega_V\}$. It remains an open question whether or not $\{q, \Omega_V\}$-CH Subset Choice is in FPT. Hence, at present time, we do not know whether $\{q, \varepsilon, \Omega_V\}$ or $\{q, \Omega_V\}$ is a crucial source of complexity in CH Subset Choice.

Despite the open question posed above, I note that we did succeed in identifying crucial sources of complexity for some special cases of CH Subset Choice. Namely, from Theorem 6.3 (page 130) we know that q-ECG Subset Choice $\in$ FPT. Since $\{q\}$ contains only one element it is minimal, and thus we know that $\{q\}$ is a crucial source of complexity for ECG Subset Choice (and thus also for UCG Subset Choice). Further, from Corollary 6.8 we know q-CG Subset Choice $\notin$ FPT (unless FPT = W[1]), from Corollary 6.13 we know Ω_V-CG Subset Choice $\notin$ FPT (unless FPT = W[1]), and from Theorem 6.3 (page 130) we know $\{q, \Omega_V\}$-CG Subset Choice $\in$ FPT. We conclude that $\{q, \Omega_V\}$ is a crucial source of complexity in CG Subset Choice (and thus also for VCG Subset Choice).

6.7. Surplus Subset Choice

In Sections 6.4 – 6.6 we have considered special cases of Subset Choice with conflict hypergraphs as input. Here I briefly discuss the special cases of Subset Choice with surplus hypergraphs as input, called SH Subset Choice. Recall that in surplus hypergraphs all vertices have negative weights and all hyperedges have positive weights. Arguably, like CH Subset Choice, also SH Subset Choice often arises in practice. To illustrate, consider again the situation in which a consumer needs to decide on a set of computer parts. Since each part (e.g. computer, monitor, printer, scanner) costs money—and by itself is useless—we may assume that each part when evaluated alone has negative value. In combining parts, however, we may obtain a value surplus. For example, a computer alone, a monitor alone, and a printer alone may all have negative value, but the computer-monitor-printer combination may have positive value.

In the following we consider the special case of SH Subset Choice where the value-structure is modeled by a unit-weighted surplus graph (i.e., span $\varepsilon = 2$, all vertices have weight '-1,' and all edges have weight '$+1$'). We will refer to this special case as USG Subset Choice. In Section 6.3, we have seen that Subset Choice is NP-hard even on unit-weighted conflict graphs. Note that the only difference between unit-weighted conflict graphs and unit-weighted surplus graphs is a reversal in sign of vertex- and edge-weights. Interestingly, this reversal in sign leads to a different complexity classification.

Theorem 6.7. USG Subset Choice $\in$ P.

To prove Theorem 6.7 we prove Lemma 6.10 below. The lemma implies that we can solve USG Subset Choice by simply choosing all the vertices in every component of G that contains at least one cycle; we then check whether this set of vertices has at least value p in G. Since the components and cycles in a graph can be found in polynomial time (Brandstädt, Le, & Spinrad, 1999; Gross & Yellen, 1999), Theorem 6.7 follows.

Lemma 6.10. Let $G = (V, E)$ be an instance for USG Subset Choice. Let $C_i = (V_i, E_i)$, $1 \leq i \leq l$, be the components of G. Let U denote the set of components of G that contain at least one cycle. Then the set $V' = \{v \in V \mid v \in V_i \text{ and } C_i \in U, 1 \leq i \leq l\}$ has maximum value.

Proof: Let $V' = \{v \in V \mid v \in V_i \text{ and } C_i \in U, 1 \leq i \leq l\}$. We show in two steps that value$_G(V')$ is maximum. We show that (1) we cannot improve the value of any vertex set $V^* \subseteq V$ by including vertices from $V \backslash V'$ in V^*. We conclude that there is a vertex set with maximum value that does not contain any vertices from $V \backslash V'$. Then we show that (2) there does not exist a set $V^* \subset V'$ with value$_G(V^*) >$ value$_G(V')$.

(1) Let $V^* \subseteq V$. Consider the subgraph F of G induced by vertex set $V \backslash V'$. Then by the definition of V', $F = (V \backslash V', E_G(V \backslash V'))$ is a forest. In any forest the number of edges is smaller than the number of vertices (for a proof of this basic property see e.g. Gross & Yellen, 1999, see also page 90). Since every subgraph of a forest is also a forest, we know that for any subset $V'' \subseteq V \backslash V'$, $|E_G(V'')| < |V''|$. Therefore, for any subset $V'' \subseteq V \backslash V'$, value$_G(V'') = |E_G(V'')| - |V''| < 0$. Consider $V^* \cup V''$. Then value$_G(V^* \cup V'') <$ value$_G(V^*)$. In other words, we can never improve the value of a vertex set $V^* \subseteq V$ by including vertices in $V \backslash V'$.

(2) From the above we know that *if* the value of V' is *not* maximum, then there must exist a set $V^* \subset V'$ with value$_G(V^*) >$ value$_G(V')$. We show by contradiction that such a set cannot exist. Let $V^* \subset V'$ be a largest vertex set with value$_G(V^*) >$ value$_G(V')$. Then at least one of the following two situations is true. (a) There exists a vertex $v \in V' \backslash V^*$ such that v has at least one neighbor $u \in V^*$. But then value$_G(V^* \cup \{v\}) \geq$ value$_G(V^*) + |E_G(\{v, u\})| - |\{v\}|$ = value$_G(V^*) + |\{(v, u)\}| - |\{v\}|$ = value$_G(V^*) + (1 - 1)$ = value$_G(V^*)$, contradicting the claim that V^* is a largest vertex set with value$_G(V^*) >$ value$_G(V')$. (b) There exists a cycle $< v_1, v_2, ..., v_k, v_1>$ with $v_1, v_2, ..., v_k \in V' \backslash V^*$. But then

value$_G$($V^* \cup \{v_1, v_2, ..., v_k\}$) ≥ value$_G$($V^*$) + $|E_G(\{v_1, v_2, ..., v_k\})| - |\{v_1, v_2, ..., v_k\}|$ ≥ value$_G$(V^*) + $k - k$ ≥ value$_G$(V^*), again contradicting the claim that V^* is a largest vertex set with value$_G$(V^*) > value$_G$(V'). ■

Theorem 6.7 shows that Subset Choice is computationally easy for value-structures that can be modeled by unit-weighted surplus graphs. It remains an open question, however, whether Subset Choice for general surplus hypergraphs is in P.

6.8. Subset Choice when Size Matters

Throughout this chapter we have assumed no restrictions on the *size* of the to-be-chosen subset. Clearly, in many real-world settings size restrictions do apply (see also Figure 6.1); e.g., there may be an exact bound (as when you have exactly k job positions to fill), an upper-bound (as when you can afford at most k toppings on your pizza), or a lower-bound (as when you require at least k members in a committee) on the size of the to-be-chosen subset. In this section I offer a word of caution: We cannot assume that the results for subset choice without size restrictions automatically generalize to subset choice with size restrictions. To illustrate, we consider the problem Exact-bound Subset Choice:

Exact-bound Subset Choice

Input: A *weighted* hypergraph $H = (V, E)$, $\bigcup_{2 \le h \le |V|} V^h$, for every $v \in V$ a weight $w_V(v) \in \mathbb{Z}$, for every $e \in E$ a weight $w_E(e) \in \mathbb{Z}\backslash\{0\}$, and positive integers p and k.

Question: Does there exist a subset $V' \subseteq V$ such that value$_H$(V') ≥ p and $|V'| = k$?

Recall that USG Subset Choice *without* size restrictions is in P (Theorem 6.7, page 149). In contrast, we have the following theorem for Exact-bound Subset Choice on unit-weighted surplus graphs.

Theorem 6.8. USG Exact-bound Subset Choice is NP-hard.

To prove Theorem 6.8 we reduce from the NP-complete problem Clique (Garey & Johnson, 1979). For a graph $G = (V, E)$ and a subset $V' \subseteq V$, we say V' is a *clique* for G if for every two vertices $u, v \in V'$, $(u,v) \in E$.

Clique (*decision version*)

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Does there exist a clique $V' \subseteq V$ for G with $|V'| \ge k$?

For a *graph* $G = (V, E)$ and a positive integer k, we make two observations. First, if V' is a clique for G then any subset $V'' \subseteq V'$ is also a clique for G. We conclude Observation 6.6.

Observation 6.6. If G and k form a yes-instance for Clique then there exists $V' \subseteq V$ such that V' is a clique and $|V'| = k$.

Further, for any set $V' \subseteq V$, with $|V'| = k$, the number of possible pairs of vertices is given by $\frac{k(k-1)}{2}$. We conclude Observation 6.7.

Observation 6.7. Let $V' \subseteq V$ with $|V'| = k$. Then V' is a clique if and only if $|\mathrm{E}_G(V')| = \frac{k(k-1)}{2}$.

Using Observations 6.6 and 6.7, we now prove Theorem 6.8.

Proof of Theorem 6.8: Let graph $G = (V, E)$ and positive integer k constitute an instance for Clique. From G and k we build an instance for USG Exact-bound Subset Choice consisting of G^*, p and k as follows: let $G^* = (V^*, E^*)$ be a weighted graph such that $V^* = V$ and $E^* = E$. For every $v \in V^*$ let $\mathrm{w_V}(v) = -1$ and for every $e \in E^*$ let $\mathrm{w_E}(e) = +1$ (i.e., G^* is a unit-weighted surplus graph). Further, let $p = \frac{k(k-1)}{2} - k$. This transformation can be done in time $O(|V|^2)$. It remains to be shown G and k form a yes-instance for Clique if and only if G^*, p and k form a yes-instance for Exact-bound Subset Choice. Because Clique is NP-hard even for $k \geq 5$, in the following we can assume that $k \geq 5$.

($\Rightarrow$) Let G and k form a yes-instance of Clique. Then, from Observation 6.6, we know there exists $V' \subseteq V$, such that V' is a clique and $|V'| = k$. Since $\mathrm{value}_{G^*}(V') = \sum_{v \in V'} \mathrm{w_V}(v) + \sum_{e \in \mathrm{E}_{G^*}(V')} \mathrm{w_E}(e) = |\mathrm{E}_{G^*}(V')| - |V'| = \frac{k(k-1)}{2} - k = p$, we conclude that G^*, p and k form a yes-instance for USG Exact Bound Subset Choice.

($\Leftarrow$) Let G^*, p and k form a yes-instance for USG Exact Bound Subset Choice. Then there exists $V' \subseteq V = V^*$ with $|V'| = k$ and $\mathrm{value}_{G^*}(V') = |\mathrm{E}_{G^*}(V')| - |V'| \geq p =$

$\frac{k(k-1)}{2} - k$. But that means that $|E_G(V')| = |E_{G^*}(V')| = \frac{k(k-1)}{2}$ and thus, from Observation 6.7, we conclude that *V'* is a clique for *G* of size *k*. ■

Note that the polynomial-time reduction in the proof above also happens to be a parametric reduction from *k*-Clique to $\{k, p\}$-USG Exact Bound Subset Choice, with $p = \frac{k(k-1)}{2} - k$. Since *k*-Clique is known be W[1]-complete (Downey & Fellows, 1999), the reduction establishes that $\{k, p\}$-USG Exact Bound Subset Choice is W[1]-hard.

Corollary 6.15. $\{k, p\}$-USG Exact Bound Subset Choice $\notin$ FPT (unless FPT = W[1]).

Theorem 6.7 and 6.8 illustrate that classical complexity results do not automatically generalize to subset choice under subset-size restrictions. The same holds for parameterized complexity results. To know which results do generalize, and which do not, a case-by-case analysis will need to be performed.

6.9. Conclusion

In this chapter I have illustrated techniques for complexity analysis by considering the problem Subset Choice, a generalization of a model proposed by Fishburn and LaValle, (1996; see also van Rooij et al., 2003). Section 6.3 presented multiple polynomial-time reductions to prove that Subset Choice is NP-hard. Sections 6.4 and 6.5 presented parameterized complexity analyses of Subset Choice on conflict hypergraphs. The results obtained in these sections were then used, in Section 6.6, to illustrate the notion of ‘crucial source of complexity.’ Section 6.7 contrasted UCG Subset Choice with USG Subset Choice and showed that, while the former is NP-hard, the latter is in P. Finally, Section 6.8 considered Subset Choice with subset-size restrictions, and illustrated that results obtained for subset choice problems *without* subset-size restrictions do not necessarily generalizes to subset choice problem *with* subset-size restrictions.

Besides illustrating techniques, the analyses have lead to some interesting observations about Subset Choice. Among other things, we have found that Subset Choice is computationally easy (1) on unit-weighted conflict graphs if q is not too large, (2) on conflict graphs if q and Ω_V are not too large, (3) on conflict hypergraphs if q, ε, and Ω_V are not too large, and (4) on unit-weighted surplus graphs. In contrast, Subset Choice on unit-weighted surplus graphs is computationally hard if there is an exact bound on the size of the chosen subset. Further, we have found that the parameter sets $\{p, \varepsilon, \omega_V, \Omega_V, \omega_E, \Omega_E\}$ and $\{q, \varepsilon, \omega_V, \omega_E, \Omega_E\}$ do not constitute crucial sources of complexity for Subset Choice—not on general hypergraphs, and not on conflict hypergraphs.

Many open questions remain. As noted in Section 6.5, it remains to be shown whether $\{q, \Omega_V\}$ or $\{q, \varepsilon, \Omega_V\}$ is a crucial source of complexity in Subset Choice on conflict hypergaphs. In other words, does the span of a value-structure (i.e., the degree of interaction between choice alternatives) add non-polynomial complexity over and above q and Ω_V? Further, we do not know yet whether Subset Choice on surplus hypergraphs is in P. Also the effect of subset-size restriction on the computational complexity of Subset Choice remains to be explored.

In this chapter I have chosen to distinguish between conflict hypergaphs and surplus hypergaphs. These choices were motivated by (1) the belief that there exist real-world value-structures that conform to one of these two types of hypergraphs, and (2) the search for useful classical and/or fixed-parameter tractability results for special cases of Subset Choice. Conflict hypergraphs and surplus hypergraphs are, of course, just two examples of value-structures that may arise in practice. Future research may aim to study also other types of value-structures that arise in real-world situations (e.g, value-structures that can be modeled by 'intersection hypergraphs" as in Figure 6.2).

Chapter 7. Cue Ordering and Visual Matching

Chapters 5 and 6 presented detailed complexity analyses of the problems Coherence and Subset Choice. Both Coherence and Subset Choice are (modeled as) hypergraph problems. Complexity analysis is of course not restricted to that particular class of problems—it extends to all combinatorial problems (see e.g. Downey & Fellows, 1999). To illustrate, this chapter discusses two very different problems; the first is a *permutation problem* and the second is a *number problem*. The permutation problem is called Min-Incomp-Lex and was formulated by Martignon and Schmitt (1999; Schmitt, 2003) in the domain of binary-cue prediction. The number problem is called Bottom-up Visual Matching and was formulated by Tsotsos (1989, 1990) in the domain of visual search. We consider each problem in turn and sketch how the parameterized complexity techniques introduced in Chapter 4 also apply to these types of problems. The chapter closes with a combined discussion.

7.1. Min-Incomp-Lex

Here we consider the problem Min-Incomp-Lex, as formulated by Martignon and Schmitt (1999; Schmitt, 2003). Section 7.1.1, explains the application of this problem in the domain of binary-cue prediction (see also Martignon & Hoffrage, 2002). Section 7.1.2 presents notation, terminology and the exact problem definition of Min-Incomp-Lex. Finally, in Section 7.1.3, we discuss the result by Martignon and Schmitt (1999; Schmitt, 2003) that Min-Incomp-Lex is NP-hard, and sketch some preliminary fpt-results.

7.1.1. Motivation: Binary-Cue Prediction

Martignon and Schmitt (1999; see also Gigerenzer & Goldstein, 1996; Martignon & Hoffrage, 2002; Todd & Gigerenzer, 2000) studied the cognitive task of predicting the ordering of two objects (is $a > b$? or is $b > a$?) based on information about relevant features of the objects. As an example, consider the following scenario. You are asked which of two cities (the *objects*), say, Amsterdam and The Hague, has the larger population (the *ordering*). You are further given the following information (the *features*): Amsterdam has an international airport while The Hague does not, both cities have a street-car system, Amsterdam is the capital of the Netherlands, Dutch government is stationed in The Hague. Note that each feature takes on a *binary* value (e.g., a city either is a capital city or it is not; a city either has an airport or it does not). Further, the features serve as (fallible) *cues* to the relationship between the two cities with respect to their size (e.g., capital cities tend to be larger than other cities, government cities are typically larger than other cities).

If the presence (absence) of a feature serves as a cue that the object is likely "large," then we code the presence (absence) of that feature as '1,' and code the absence (presence) of that feature as '0.' For example, in Figure 7.1, we are given a set of objects $A = (a_1, a_2, ..., a_6)$ and set of features $F = \{f_1, f_2, \ldots, f_{10}\}$. Each object $a_i \in A$ has a value (either '1' or '0') for each of the 10 features, $f_1, f_2, \ldots, f_{10}$. Further, for simplicity, objects are labeled according to size, with $a_1 > a_2 > ... > a_6$. Note that, in the task of binary-cue prediction this ordering is unknown, and the task is to *predict* the order for any given pair based on the feature values. Since the feature values are *fallible* cues the prediction may be incorrect.

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
a_1	1	0	1	1	0	1	1	1	1	1
a_2	1	1	1	1	0	1	1	0	0	1
a_3	0	1	1	0	0	0	1	1	0	0
a_4	1	0	0	1	0	1	0	1	0	1
a_5	0	0	0	0	0	1	0	0	0	0
a_6	0	0	0	0	0	1	0	0	0	0

Figure 7.1. Illustration of a binary-cue prediction task.
A set of objects $A = (a_1, a_2, ..., a_6)$ and set of features $F = \{f_1, f_2, \ldots, f_{10}\}$. The binary-value (0 or 1) of object a_i on feature f_j is listed in cell (i,j) of the matrix. Objects are labeled according to size, i.e., $a_1 > a_2 > ... > a_6$. The ordering of objects is unknown to the person performing the binary-cue prediction task.

Say you are given the pair of objects $(a_1, a_3) \in A \times A$ from Figure 7.1. How would you use the information about the features of a_1 and a_3 to predict which is the larger of the two? There are several reasonable strategies one may take. One strategy described by

Martignon and Schmitt (1999) is the *lexicographic strategy* (see also Gingerenzer & Goldstein, 1996; Todd & Gigerenzer, 2000; Martignon & Hoffrage, 2002; cf. Payne et al. 1993). This strategy works as follows: For a given pair of objects we compare their features values, considering features in a *pre-specified* order, and as soon as a feature is encountered for which the two objects take a different value, we predict that the object with value '1' on that feature is the larger of the two.

To illustrate, consider again Figure 7.1. We define a fixed permutation $\pi(F)$ of n features, $\pi(F) = <f_{\pi_1}, f_{\pi_2}, \ldots, f_{\pi_n}>$. For example, for $F = \{f_1, f_2, \ldots, f_{10}\}$ in Figure 7.1 let us set $\pi(F) = <f_8, f_5, f_2, f_4, f_1, f_7, f_6, f_{10}, f_9, f_3>$. This permutation is then used to compare every possible pair in $A \times A$ as follows. For each pair we consider the features in the order specified by $\pi(F)$; as soon as we encounter a feature that takes a different value for the two objects, search is terminated and we return the object with the value '1' on that feature (the returned object is predicted to be the larger of the two). For example, let $(a_1, a_3) \in A \times A$ be the pair that we are comparing at this moment. We start by considering the first feature $f_{\pi_1} = f_8$ in our predetermined feature permutation $\pi(F)$. We note that a_1 and a_3 both take the same value on this feature (both have value '1'), so the feature does not serve as a cue to the order of these two objects (we say f_8 does not *distinguish* between a_1 and a_3). We continue by considering the second feature in the permutation, $f_{\pi_2} = f_5$. Because f_5 also does not distinguish between a_1 and a_3 we continue to consider the third feature in the permutation, $f_{\pi_3} = f_2$, and find that a_1 and a_3 have a different value on this feature (one has value '1' and the other value '0'). Because a_3 is the one with value '1,' we predict a_3 is the larger object.

In our example, using the feature permutation $\pi(F) = <f_8, f_5, f_2, \ldots, f_9, f_5>$, we predicted that a_3 is larger than a_1. This is an incorrect prediction, since $a_1 > a_3$. We could, of course, change the feature permutation so that pair (a_1, a_3) would be compared correctly (e.g., we could set $\pi(F) = <f_8, f_5, f_1, \ldots, f_9, f_5>$ instead), but this may cause other pairs to be compared incorrectly (e.g., pair (a_3, a_4) is compared incorrectly by the new permutation, while it was compared correctly by the original permutation).

Clearly, when using the Lexicographic strategy as described here, the total number of incorrect paired-comparisons one will make for a given set of objects A (i.e., the total number summed over all possible $m(m-1)/2$ pairs for $|A| = m$ objects) depends on how the permutation $\pi(F)$ is defined. Martignon and Schmitt (1999) studied the task of determining an *optimal* feature permutation—i.e., one that leads to a minimum number of total incorrect comparisons. They called this task Min-Incomp-Lex.[65] The next section defines this problem more formally.

7.1.2. Notation, Terminology and Problem Definition

We start by defining notation and terminology. Let $A = \{a_1, a_2, \ldots, a_m\}$ denote a set of *objects* and let $F = \{f_1, f_2, \ldots, f_n\}$ denote a set of *features*, with $|A| = m$ and $|F| = n$. For each $a \in A$ there is an associated value $b_i(a) \in \{0, 1\}$ denoting the *binary-value* that object a takes on feature f_i, $i = 1, 2, \ldots, n$. The vector $[b_1(a_i)\ b_2(a_i)\ \ldots\ b_n(a_i)]$ we call the *feature vector* of a_i and the vector $[b_j(a_1)\ b_j(a_2)\ \ldots\ b_j(a_m)]$ we call the *object vector* of feature f_j. For example, in Figure 7.1 feature vectors appear in the rows, and object vectors appear in the columns (e.g. object a_2 has feature vector [1 1 1 1 0 1 1 0 0 1] and feature f_4 has object vector [1 1 0 1 0 0]).

There is a complete ordering $S(A)$ on the elements in A: i.e., for any two objects $a_i, a_j \in A$, $i \neq j$, we have either $a_i > a_j$ or $a_i < a_j$ in $S(A)$. Unless otherwise noted, objects in A are labeled in order: i.e., $a_1 > a_2 > a_3 > \ldots > a_m$.[66] We say a feature f_k *distinguishes* a pair of objects $(a_i, a_j) \in A \times A$, if $b_k(a_i) \neq b_k(a_j)$. Further, if $b_k(a_i) > b_k(a_j)$ then f_k is said to *predict* that $a_i > a_j$, and if $i < j$ we say the prediction is *correct* (by this we mean that the prediction matches the order of a_i, a_j in S(A). A permutation of features $F = \{f_1, f_2, \ldots, f_n\}$ is denoted by $\pi(F) = \langle f_{\pi 1}, f_{\pi 2}, \ldots, f_{\pi n}\rangle$.

We next define the Lexicographical (LEX) algorithm. LEX is an algorithm that takes as input a set of objects $A = \{a_1, a_2, \ldots, a_m\}$, a complete ordering $S(A) = (a_1 > a_2 > a_3 > \ldots > a_m)$ of objects in A, a set of features $F = \{f_1, f_2, \ldots, f_n\}$, a set of pairs of distinct

[65] Min-Incomp-Lex stands for *min*imum number of *in*correct *comp*arisons under the *lex*icographic strategy.
[66] The only exception to this rule appears in the proof of Theorem 7.1, where the labeling of objects in A is driven by the reduction.

objects $P \subseteq A \times A$, and a permutation $\pi(F) = \langle f_{\pi 1}, f_{\pi 2}, \ldots, f_{\pi_n} \rangle$. LEX considers each feature $f \in F$ in the order specified by the permutation $\pi(F)$. For each such feature $f \in F$, LEX tests for each pair $(a_i, a_j) \in P$, whether f distinguishes between a_i and a_j. If it does, then LEX predicts that the object with value '1' on feature f is the larger of the two, and the pair is removed from P (for any given pair we never make a prediction more than once). Meanwhile, LEX keeps track of the number of incorrect predictions it makes (see the variable 'error' in the algorithm).

Lexicographical (LEX) algorithm

Input: A set of objects $A = \{a_1, a_2, \ldots, a_m\}$, a complete ordering $S(A) = (a_1 > a_2 > a_3 > \ldots > a_m)$ of objects in A, a set of features $F = \{f_1, f_2, \ldots, f_n\}$, a set of pairs of objects $P \subseteq A \times A$, and a permutation $\pi(F) = \langle f_{\pi 1}, f_{\pi 2}, \ldots, f_{\pi_n} \rangle$.

Output: A prediction (either $a_i > a_j$ or $a_i < a_j$) for each pair $(a_i, a_j) \in P$ that is distinguished by at least one feature in F, and the value *error*.

```
1. $error := 0$
2. **while** $(P \neq \emptyset)$ **do**
    3. pick a pair $(a_i, a_j) \in P$
    4. $x := 1$
    5. **while** $(x \leq n)$ **do**
        6. **if** $b_{\pi x}(a_i) > b_{\pi x}(a_j)$ **then**
            7. predict and output $a_i > a_j$
            8. **if** $i > j$
                9. $error := error + 1$
            10. **end if**
            11. $x := n$
        12. **end if**
        13. **if** $b_{\pi x}(a_i) < b_{\pi x}(a_j)$ **then**
            14. predict and output $a_i < a_j$
            15. **if** $i < j$
                16. $error := error + 1$
            17. **end if**
            18. $x := n$
        19. **end if**
        20. $x = x + 1$
    21. **end while**
    22. $P := P\backslash\{(a_i, a_j)\}$
23. **end while**
24. **return** *error*
```

Note that for some pairs in P the algorithm LEX may not output a prediction. This means that we cannot distinguish these pairs based on the features in F. Note that such undistinguished pairs do not contribute to the error count. Further, note that LEX runs in time $O(nm^2)$: There are in the worst case $m(m - 1)/2$ pairs of objects to compare and n features to consider per pair. We define the function $\text{Error}_{\text{LEX}}(A, \text{S}(A), F, P, \pi(F)) = error$, where *error* is the value returned in line 24 of LEX when run on input $(A, \text{S}(A), F, P, \pi(F))$. Note that $0 \leq \text{Error}_{\text{LEX}}(A, \text{S}(A), F, P, \pi(F)) \leq |P|$. Using this definition of the function $\text{Error}_{\text{LEX}}(A, \text{S}(A), F, P, \pi(F))$ we define the problem Min-Incomp-Lex as follows:

Min-Incomp-Lex (*optimization version*)

Input: A set of objects $A = \{a_1, a_2, \ldots, a_m\}$ and a set of features $F = \{f_1, f_2, \ldots, f_n\}$. Each $a \in A$ has an associated value $b_i(a) \in \{0, 1\}$, $i = 1, 2, \ldots, n$, denoting the value that a takes on feature $f_i \in F$. A complete ordering $S(A)$, with $a_1 > a_2 > \ldots > a_m$.

Output: A permutation $\pi(F) = \langle f_{\pi_1}, f_{\pi_2}, \ldots, f_{\pi_n} \rangle$, such that $\text{Error}_{\text{LEX}}(A, \text{S}(A), F, P, \pi(F))$ is minimized. Here $P \subseteq A \times A$ denotes the set of all pairs of distinct objects in A.[67]

In the analyses we again work with the problem's decision version:

Min-Incomp-Lex (*decision version*)

Input: A set of objects $A = \{a_1, a_2, \ldots, a_m\}$ and a set of features $F = \{f_1, f_2, \ldots, f_n\}$. Each $a \in A$ has an associated value $b_i(a) \in \{0, 1\}$, $i = 1, 2, \ldots, n$, denoting the

[67] Two objects $a_i, a_j \in A$ are said to be *distinct* if $i \neq j$.

value that a takes on feature $f_i \in F$. A complete ordering $S(A)$, with $a_1 > a_2 > \ldots > a_m$. An integer $k \geq 0$.

Question: Does there exist a permutation $\pi(F) = \langle f_{\pi 1}, f_{\pi 2}, \ldots, f_{\pi n} \rangle$, such that $\text{Error}_{\text{LEX}}(A, \text{S}(A), F, P, \pi(F)) \leq k$? Here $P \subseteq A \times A$ denotes the set of all pairs of distinct objects in A.

In the remainder of this section an instance for Min-Incomp-Lex will be denoted by a 4-tuple $(A, F, S(A), k)$.

7.1.3. Classical and Parameterized Complexity

The problem Min-Incomp-Lex has been shown to be NP-hard (Martignon & Schmitt, 1999; Schmitt, 2003).

Theorem 7.1. Min-Incomp-Lex is NP-hard (Martignon & Schmitt, 1999).

The proof of Theorem 7.1 involves a reduction from Vertex Cover. I sketch the reduction below. For the details of the proof the reader is referred to (Schmitt, 2003).

Sketch of proof for Theorem 7.1. Given an instance (G, k), $G = (V, E)$, for Vertex Cover we create an instance $(A, F, S(A), k)$ for Min-Incomp-Lex as follows: Let $F = \{f_1, f_2, \ldots, f_n, f_{n+1}\}$ be a feature set with $|F| = |V| + 1 = n + 1$. We define one special object $a_0 \in A$ with $b_x(a_0) = 1$, for $x = 1, 2, \ldots., n$ and $b_{n+1}(a_0) = 0$. Then, for every vertex $v_i \in V$, we define $a_i \in A$ with $b_x(a_i) = 0$ for $x = i$, and $b_x(a_i) = 1$ for all $x \neq i$, and let $a_i > a_0$ in $\text{S}(A)$; and for every edge $(v_i, v_j) \in E$, we define $a_{i,j} \in A$ with $b_x(a_{i,j}) = 0$ for $x = i$, and $x = j$, and $b_x(a_{ij}) = 1$ for all $x \neq i, j$, and let $a_{i,j} < a_0$ in $\text{S}(A)$. Note that so far we have defined only a *partial* order on the elements of A. We complete the order as follows: For all pairs (a_g, a_h) for which the order is not yet defined, if $g < h$ then let $a_g > a_h$; if $g > h$ then let $a_g < a_h$. On this transformation (G, k) is a yes-instance for Vertex Cover if and only if $(A, F, S(A), k)$ is a yes-instance for Min-Incomp-Lex. ■

Martignon and Schmitt (1999, p. 574; see also Martignon & Hoffrage, 2002, p. 39) contend that, with Theorem 7.1, it is proven that there are no essentially simpler strategies for solving Min-Incomp-Lex than by searching through all n! possible permutations of n features.[68] Since, $\text{Error}_{\text{LEX}}(A, \text{S}(A), F, P, \pi(F))$ can be computed in time $O(nm^2)$ per permutation $\pi(F)$, the proposed exhaustive search would run in time $O(n!\ nm^2)$. In the following we qualify the claim by Martignon and Schmitt.

Although Martignon and Schmitt are correct in saying that Theorem 7.1 establishes that no polynomial-time algorithm can solve Min-Incomp-Lex (unless P = NP), there may still exist algorithms with running times that are polynomial in $n = |F|$ (albeit non-polynomial in other aspects of the input). In other words, there may exist an fpt-algorithm that runs in time $O(g(\kappa)n^{\alpha})$, where α is a constant, κ is some input parameter of Min-Incomp-Lex, and function g depends only on κ (not on n). A first obvious input parameter for Min-Incomp-Lex is the size of the set of objects, $|A| = m$. Next we show there exists an fpt-algorithm for m-Min-Incomp-Lex that runs in time $O(g(m)\ n^{\alpha})$, with $\text{g}(m) = 2^m!m^2$ and $\alpha = 2$.

Consider again Figure 7.1, and observe that, for example, the object vectors for f_1 and f_4 are the same; i.e., both are [1 1 0 1 0 0]. This means that for each pair $(a_i, a_j) \in P$, both features f_1 and f_4 make the same prediction (either both predict $a_i > a_j$ or both predict $a_i < a_j$), or both make no prediction. Since in the computation of the function $\text{Error}_{\text{LEX}}(A, \text{S}(A), F, P, \pi(F))$ we never consider a pair in P more than once, only one of f_1 and f_4 can contribute to the error count. This means that deleting one of them from F does not affect the answer to the problem Min-Comp-Lex. This observation leads to the following reduction rule:

(MIL 1) Feature Duplicates Rule. Let $(A, F, S(A), k)$ be an instance for Min-Incomp-Lex. Further, let $f_x, f_y \in F$ be two features such that for every pair $(a_i, a_j) \in P$ features f_x and f_y make the same prediction or both features make no prediction. Then let $(A, F^*, S(A), k)$ with $F^* = F\backslash\{f_y\}$, be the new instance for Min-Incomp-Lex.

We call an instance $(A, F, S(A), k)$ for Min-Incomp-Lex *reduced* if (MIL 1) does not apply to $(A, F, S(A), k)$. How many features can a reduced instance have? It can have at most $|F| \leq 2^m$ features, since with m objects we can have at most 2^m different object

[68] I note that, assuming P $\neq$ NP, an NP-hardness proof is sufficient to conclude at least exponential time complexity of a problem (i.e., a complexity on the order of α^n for constant α, or worse), but not a complexity on the order of $n!$.

vectors. We reduce an instance in time $O(n^2m)$,[69] and we can solve Min-Incomp-Lex in time $O(|F|!\ n^2m)$—simply by checking all possible permutations of F—we conclude that Min-Incomp-Lex is solvable in time $O(2^m!nm^2 + n^2m)$, which is $O(2^m!\ n^2m^2)$. Since $O(2^m!\ n^2m^2)$ is fpt-time for m-Min-Incomp-Lex, we conclude that m-Min-Incomp-Lex $\in$ FPT.

Now, a critical reader may contest that in many applications for the Min-Incomp-Lex problem $|A|$ is larger than $|F|$. In such applications $2^m! >> n!$, making the running time $O(2^m!\ n^2m^2)$ worse than the original $O(n!\ nm^2)$. In this case, I have two comments: (1) Since the size of parameters depends on the particulars of the application, knowing the size of parameters goes beyond complexity analysis *per se*. (2) Admittedly $2^m!$ is a horrible function and a running time $O(2^m!\ n^2m^2)$ is unfeasible even for m as small as 5. The point of the illustration, however, was merely to show that it is possible to have a running-time that is polynomial in n. Furthermore, it may very well be possible to bound $|F|$ by a much slower growing function of $|A|$ than the one I have presented here. I leave this for future research to determine.

A second point of criticism might be that, although we have shown that it is possible to have a running time for Min-Incomp-Lex that is polynomial in n, the strategy used is still not essentially different than checking all possible permutations on $|F|$ features. That is, although we introduced a reduction rule (MIL 1) that potentially shrinks the size of F, as well as bounds $|F|$ in a function of $|A|$, if (MIL 1) does not apply (anymore) we still proceeded with an exhaustive search on all $|F|!$ possible feature permutations. In other words, the main strategy for solving Min-Incomp-Lex has remained basically the same; only our analysis changed. Next we derive a more strategic algorithm. To do so we first define an annotated version of Min-Incomp-Lex as follows:

Annotated Min-Incomp-Lex

Input: A set of objects $A = \{a_1, a_2, \ldots, a_m\}$ and a set of features $F = \{f_1, f_2, \ldots, f_n\}$. Here F partitions into $F_1 = \{f_1, f_2, \ldots, f_{n_1}\}$ and $F_2 = \{f_{n_1+1}, f_{n_1+2}, \ldots, f_n\}$. Each $a \in A$ has an associated value $b_i(a) \in \{0, 1\}$, $i = 1, 2, \ldots, n$, denoting the value that a takes on feature $f_i \in F$. A complete ordering $S(A)$, with $a_1 > a_2 > \ldots > a_m$, and a permutation of the features in F_1, $\pi(F_1)$. An integer $k \geq 0$.

[69] To compare two object vectors we need to make at most $m = |A|$ comparisons; and there are at most $n(n - 1)/ 2 = |F|(|F| - 1)/2$ pairs of features to check.

Question: Does there exist a permutation of the features in F_2, $\pi(F_2)$, such that $\text{Error}_{\text{LEX}}(A, S(A), F, P, \pi(F_2)) \leq k$? Here $P \subseteq A \times A$ denotes the set of pairs of distinct objects in A that are not distinguished by any feature in F_1.

We denote an instance of Annotated Min-Incomp-Lex by a 5-tuple $(A, (F_1 \cup F_2)^F, S(A), \pi(F_1), k)$. Note that Min-Incomp-Lex is a special case of Annotated Min-Incomp-Lex with $F_1 = \varnothing$. Annotated Min-Incomp-Lex can be thought of as the problem that arises when we are in the process of solving Min-Incomp-Lex: The features for the first n_1 positions of the permutation $\pi(F)$ for Min-Incomp-Lex have already been determined (these are the features in F_1, and their order is given by $\pi(F_1)$). The remaining $n - n_1$ positions need to be filled with the features in F_2.

We observe that the previously described reduction rule (MIL 1) for Min-Incomp-Lex directly generalizes to the reduction rule (AMIL 1) for Annotated Min-Incomp-Lex.

(AMIL 1) Feature Duplicates Rule. Let $(A, (F_1 \cup F_2)^F, S(A), \pi(F_1), k)$ be an instance for Annotated Min-Incomp-Lex. Further, let $f_x, f_y \in F_2$ be two features such that for every pair $(a_i, a_j) \in P$, feature f_x and f_y make the same prediction or both features make no prediction. Then let $(A, (F_1 \cup F_2{}^*)^{F^*}, S(A), \pi(F_1), k)$, with $F_2{}^* = F_2 \backslash \{f_y\}$, be the new instance for Annotated Min-Incomp-Lex.

We next derive a second reduction rule for Annotated Min-Incomp-Lex. Consider again Figure 7.1, and assume we have decided on the first two positions of our permutation, given by $\pi(F_1)$, with $F_1 = \{f_2, f_3\}$. Observe that f_2 and f_3 together distinguish the pairs (a_1, a_2), (a_1, a_3), (a_1, a_4), (a_2, a_4), (a_3, a_4), (a_1, a_5), (a_2, a_5), (a_3, a_5), (a_1, a_6), (a_2, a_6), and (a_3, a_6).[70] Recall that in the problem Annotated Min-Incomp-Lex P is the set of all pairs of distinct objects that are not distinguished by any feature in F_1. Thus, in this example, $P = \{(a_2, a_3), (a_4, a_5), (a_4, a_6), (a_5, a_6)\}$. Now inspect f_1. Even though f_1 by itself makes an incorrect prediction for pair (a_3, a_4), placing f_1 in the 3rd position of our permutation (i.e., *after* f_2 and f_3) cannot lead to an incorrect prediction anymore. Namely, $(a_3, a_4) \notin P$, and for all other pairs in P the feature f_1 by itself makes either no prediction or the correct prediction. In general, if there exists a feature $f \in F_2$ that for all pairs in P

[70] Note that which pairs are distinguished by $\{f_2, f_3\}$ does not depend on $\pi(\{f_2, f_3\})$.

either makes no prediction or makes the correct prediction, then we can safely place f in the $(n_1 + 1)$th position of our permutation. This leads to the following reduction rule:

(AMIL 2) Never-Wrong Feature Rule. Let $(A, (F_1 \cup F_2)^F, S(A), \pi(F_1), k)$, with $S(A) = (a_1 > a_2 > \ldots > a_n)$, be an instance for Annotated Min-Incomp-Lex. If there exist a feature $f_x \in F_2$, such that for every pair $(a_i, a_j) \in P$, $i < j$, we have $b_x(a_i) \geq b_x(a_j)$ then let $F_2^* = F_2 \backslash \{f_x\}$, let $F_1^* = F_1 \cup \{f_x\}$, and let $\pi(F_1^*) = < \pi(F_1), f_x >$. Finally, let $(A, (F_1^* \cup F_2^*)^{F^*}, S(A), \pi(F^*_1), k)$ be the new instance for Annotated Min-Incomp-Lex.

When solving Annotated Min-Incomp-Lex for an instance $(A, (F_1 \cup F_2)^F, S(A), \pi(F_1), k)$ we apply reduction rules (AMIL 1) and (AMIL 2) for as long as possible. If the rules do not apply anymore we call an instance $(A, (F_1 \cup F_2)^F, S(A), \pi(F_1), k)$ *reduced**. Now observe that reduced* instances for Annotated Min-Incomp-Lex have a useful property: Every feature in F_2 makes an incorrect prediction for at least one pair in P (otherwise (AMIL 2) would apply). We can use this observation to build a search tree whose depth is bounded by a function of k. We do so using branching rule (AMIL 3). This branching rule takes as input a reduced* instance $(A, (F_1 \cup F_2)^F, S(A), \pi(F_1), k)$. For each $f \in F_2$ the rule (AMIL 3) creates a new node in the search tree representing the possibility that in an optimal permutation for $(A, (F_1 \cup F_2)^F, S(A), \pi(F_1), k)$, feature f is in position $n_1 + 1$. At every node we update k to reflect the number of incorrect predictions made by feature f for pairs in P.

(AMIL 3) Feature In-or-Out Rule. Let search tree node s be labeled by an instance $(A, (F_1 \cup F_2)^F, S(A), \pi(F_1), k)$ for Annotated Min-Incomp-Lex. Then, for every $f \in F_2$ create a child s_f of node s and label it by $(A, (F^*_1 \cup F^*_2)^{F^*}, S(A), \pi(F^*_1), k^*)$. Here $F_2^* = F_2 \backslash \{f\}$, $F_1^* = F_1 \cup \{f\}$, $\pi(F_1^*) = < \pi(F_1), f >$, and $k^* = k - \text{Error}_{\text{LEX}}(A, \text{S}(A), \{f\}, P, \pi(\{f\}))$.

We now solve Min-Incomp-Lex using the following algorithm: The algorithm takes as input an instance for $(A, (F_1 \cup F_2)^F, S(A), \pi(F_1), k)$ for Annotated Min-Incomp-Lex. We initialize $F_1 = \emptyset$, and start by applying reduction rules (AMIL 1) and (AMIL 2) until no longer possible. Then we recursively apply branching rule (AMIL 3) (while always reducing* before applying (AMIL 3) again), until either $P = \emptyset$ (in which case we return the answer "yes"), or $F_2 = \varnothing$ (in which case we also return the answer "yes"),[71] or $k < 0$. If the algorithm halts without returning the answer "yes" then we return the answer "no."

Note that on each application of (AMIL 3) $k^* \leq k - 1$ and $|F_2^*| \leq |F_2| - 1$, with $|F_2| \leq |F| = n$. Thus, the ith application of (AMIL 3) leads to the creation of at most $n - (i + 1)$ new search tree nodes, and we never apply (AMIL 3) more than k times. This means that the size of the search tree is bounded by $n(n-1)(n-2) \ldots (n-(k-1))$, which is $O\left(\frac{n!}{(n-k)!}\right)$. Together with the time spent at each node of the search tree, to label the node and reduce* an instance before branching, we conclude a total running time of $O\left(\frac{n!}{(n-k)!} n^2 m^2\right)$.

The described algorithm is an fpt-algorithm for $\{n, k)$-Min-Incomp-Lex. Recall, however, that the goal was not to show that $\{n, k\}$-Min-Incomp-Lex $\in$ FPT. This we already knew. Namely, as remarked at the beginning of this subsection (page 162), the problem Min-Incomp-Lex is solvable in time $O(n!\, nm^2)$, and thus $\{n\}$-Min-Incomp-Lex $\in$ FPT. Since, $\{n, k\} \supseteq \{n\}$, it follows that also $\{n, k\}$-Min-Incomp-Lex $\in$ FPT (see Section 4.5, page 73). The goal here was to describe a strategy that is better than the naïve exhaustive search proposed by Martignon and Schmitt (1999). The algorithm that I have sketched here is not only more strategic, but also its running time $O\left(\frac{n!}{(n-k)!} n^2 m^2\right)$ is better than $O(n!\, nm^2)$ for $k < n$ (albeit still impractical for moderately sized n and k).

[71] Reminder: The pairs in P that cannot be distinguished by any feature in F_2 are ignored in the error count.

7.2. Bottom-up Visual Matching

Here we consider the problem Bottom-up Visual Matching, formulated by Tsotsos (1989, 1990, 1991). Section 7.2.1, explains the application of this problem in the domain of visual search. Section 7.2.2 presents the exact problem definition of Bottom-up Visual Matching. Finally, in Section 7.2.3, we discuss the result by Tsotsos (1989, 1990) that Bottom-up Visual Matching is NP-hard, and we show how a known result for this problem can be interpreted as an fpt-result.

7.2.1. Motivation: Visual Search

Tsotsos (1990) studied the complexity of visual search tasks. In a visual search task one is presented with a visual display containing a set of stimuli (e.g. a collection of green circles). The goal is to decide whether or not a target stimulus is present among them, where a target is defined as a stimulus that is different in some respect from all other stimuli in the display (e.g. a green square, or a red circle). In other words, the target is the “odd-man-out.” Figure 7.2 presents an illustration.

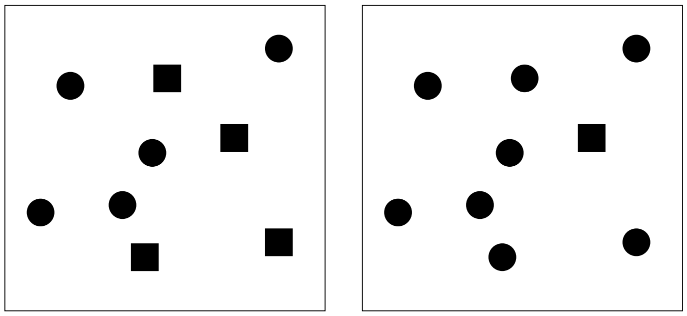


Figure 7.2. Illustration of a visual search task.
The target is the “odd man out;” i.e., a visual object that is different from all other visual objects in the display. Note that in the display on the left no such target is present, while in the display on the right there is.

Tsotsos argues that, to perform the visual search task, the perceptual system not only has to search through a set of visual objects and potential targets, but it also has to determine for each visual object and each potential target whether or not they *match*. The latter subtask of visual search, Tsotsos calls *visual matching*. Since visual input is noisy, visual matching is not a trivial task. Consider, for example, Figure 7.3. There, on the left, a visual object is displayed (called the *test image*) and a hypothesized target is presented on the right (called the *target image*). Both test image I and target image T are sets of pixels.[72] A *pixel p* is a 3-tuple $p = (x, y, b)$, where x, y specify the location of the pixel in a Euclidean coordinate system, and b is a positive integer representing the brightness level of pixel p. The images I and T use the same coordinate system and the origins coincide for I and T. Thus, for each pixel $p_i \in I$, $p_i = (x, y, b_i)$, there is a corresponding $p_t \in T$, with $p_t = (x, y, b_t)$.

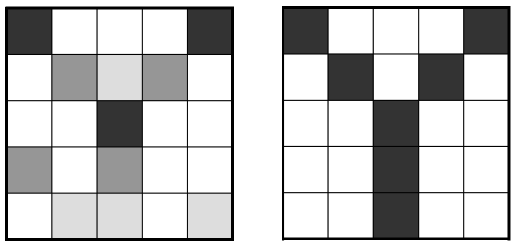


Figure 7.3. Illustration of a visual matching task.
A noisy test image I (left) and a target image T (right). Here, pixels take on four different brightness levels: (1) white, (2) light gray, (3) dark gray, and (4) black.

Do the two images in Figure 7.3 match sufficiently to warrant the response "yes, the target is present"? To determine this, Tsotsos proposes, the visual system computes for each pixel $p_i = (x, y, b_i) \in I$, (1) the *difference* in brightness of p_i and $p_t = (x, y, b_t)$, defined as $\text{diff}(p_i) = |b_i - b_t|$; and (2) the *correlation* in brightness for p_i and $p_t = (x, y, b_t)$, defined as $\text{corr}(p_i) = b_ib_t$. Figure 7.4 illustrates the computation of these functions for the test and target image in Figure 7.3.

[72] My characterization of the matching task here is a simplification of the one by Tsotsos (1989, 1990). The three main simplifications are as follows: (1) Here we assume that each pixel has only one task relevant feature (brightness), while Tsotsos' model allows pixels to have multiple features (e.g. color, depth, motion). (2) Here we assume that brightness values are represented by non-negative integers, while Tsotsos allows them to take on any non-negative value of fixed precision. (3) Here we assume $|I| = |T|$, while Tsotsos allows I to be larger than T or vice versa. Note that the restrictions that apply here are for simplicity of exposition only and do not alter the nature of the task in any fundamental way (see also Tsotsos, 1991).

b_i

4	1	1	1	4
1	3	2	3	1
1	1	4	1	1
3	1	3	1	1
1	2	2	1	2

b_t

4	1	1	1	4
1	4	1	4	1
1	1	4	1	1
1	1	4	1	1
1	1	4	1	1

diff(p_i)

0	0	0	0	0
0	1	1	1	0
0	0	0	0	0
2	0	1	0	0
0	1	2	0	1

corr(p_i)

16	1	1	1	16
1	12	2	12	1
1	1	16	1	1
3	1	12	1	1
1	2	8	1	2

Figure 7.4. Illustration of Top-down or Bottom-up Visual Matching
The two matrices at the top give a numeric representation of the brightness values in the image *I* (top-left) and the target *T* (top-right) from Figure 7.3. The matrices at the bottom show for each of the 25 pixels in *I*, the absolute difference of the brightness value of that pixel and its corresponding pixel in *T* (bottom-left), and the product of the brightness value of that pixel and its corresponding pixel in *T* (bottom-right).

Then, one possible way of determining whether or not a satisfactory match exists is as follows:

(1) Compute the sum of all difference values, $\sum_{p \in I} \text{diff}(p)$, and the sum of all correlation values, $\sum_{p \in I} \text{corr}(p)$.

(2) Set two criteria θ and ϕ (Here θ represents that largest satisfactory total difference between test and target, and ϕ represents the smallest satisfactory total correlation between test and target image).

(3) If $\sum_{p \in I} \text{diff}(p) \leq \theta$ and $\sum_{p \in I} \text{corr}(p) \geq \phi$, then we return the answer "yes" (i.e., we conclude a satisfactory match between test and target) and "no" otherwise.

Because in step (1) the boundary of the test/target image is used to guide the computation of the match, Tsotsos calls this approach Top-down Visual Matching. Top-down Visual Matching is computationally easy and solvable in time $O(|I|)$ (or in time $O(|T|)$, since $|I| = |T|$). In contrast, the problem Bottom-up Visual Matching is computationally much harder. Here the task is to decide whether there exist any arbitrary[73] subset of pixels $I' \subseteq I$, such that $\sum_{p \in I'} \text{diff}(p) \leq \theta$ and $\sum_{p \in I'} \text{corr}(p) \geq \phi$. In other words, in Bottom-up Visual Matching, unlike in Top-Down Visual Matching, the boundaries of the image and/or the target are not used to guide the search for a match. Clearly, we can solve Bottom-up Visual Matching in time $O(2^{|I|})$.[74] But the question is: Can we solve Bottom-up Visual Matching more efficiently? Section 7.2.3 discusses Tsotsos' answer to this question, and Kube's (1990, 1991) critique of it.

7.2.2. Notation, Terminology and Problem Definition

Let I denote an image, and let T denote a target. Both I and T are sets of pixels, with $|I| = |T|$. A *pixel p* is a 3-tuple $p = (x, y, b)$. Here coordinates (x, y) represent the location of p in a Euclidean coordinate system that is the same for both I and T; and b is a non-negative integer representing the brightness level of p. For each pixel $p_i \in I$, $p_i = (x, y, b_i)$, there is a corresponding $p_t \in T$, with $p_t = (x, y, b_t)$. For each such pair of pixels, we define the function diff(p_i) = $|b_i - b_t|$ and the function corr(p_i) = $b_i b_t$. Then, the problem Bottom-up Matching is defined as follows:

Bottom-up Visual Matching

Input: An image I and a target T. Each pixel $p_i = (x, y, b_i)$, $p_i \in I$, with $p_t = (x, y, b_t)$, $p_t \in T$, has an associated value diff(p_i) = $|b_i - b_t|$ and an associated value corr(p_i) = $b_i b_t$. Two positive integers θ and ϕ.

Question: Does there exist a subset of pixels $I' \subseteq I$ such that $\sum_{p \in I'} \text{diff}(p) \leq \theta$ and $\sum_{p \in I'} \text{corr}(p) \geq \phi$?

Note that, for each $p \in I$, the values diff(p) and corr(p) are part of the input, and need not be computed anymore. This assumption is made to simplify the discussion and is adopted from Tsotsos (1989, 1990).

---

[73] Tsotsos explicitly allows I' to be a subset of pixels that are not spatially contiguous (see e.g. Tsotsos, 1990, p. 429).

[74] Compute the functions $\sum_{p \in I'} \text{diff}(p)$ and $\sum_{p \in I'} \text{corr}(p)$ for all of the $2^{|I|}$ possible subsets $I' \subseteq I$ and compare them to ϕ and θ respectively.

7.2.3. Classical and Parameterized Complexity

The problem Bottom-Up Visual Matching has been shown to be NP-hard (Tsotsos, 1989).

Theorem 7.2. (Tsotsos, 1989) Bottom-Up Visual Matching is NP-hard

The proof by Tsotsos (1989) involves a reduction from the known NP-hard problem Knapsack. The decision version of this problem is as follows.

Knapsack

Input: A finite set U. Each $u \in U$ has a size $s(u) \in Z^+$ and a value $v(u) \in Z^+$. Positive integers B and K.

Question: Does there exist a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$?

The problems Bottom-up Visual Matching and Knapsack are clearly very related. The only difference between the two problems is the following. In Knapsack, for each $u \in U$, s(u) and v(u) can take arbitrary integer values, while in Bottom-up Visual Matching, for any $p_i \in I$, the value that corr(p_i) can take is constrained by diff(p_i), and vice versa. Consider, for example, corr(p_i) =12. Then we can have at most 3 different values for diff(p_i). We can have diff(p_i) = 1 (if $b_\text{i} = 3$, $b_t = 4$, or vice versa), or diff(p_i) = 4 (if $b_\text{i} = 2$, $b_t = 6$, or vice versa), or diff(p_i) = 11 (if $b_\text{i} = 1$, $b_t = 12$, or vice versa). This means that we cannot have arbitrary integer values for diff(p_i) go with arbitrary integer values for corr(p_i), as in Knapsack. As a consequence of this difference, Bottom-up Visual Matching is a special case of Knapsack, and thus the NP-hardness of Knapsack does not directly imply that Bottom-up Visual Matching is NP-hard. Nevertheless, Tsotsos has shown that there exists a polynomial-time reduction from Knapsack to Bottom-up Visual Matching, hence proving Theorem 7.2. Because the proof is quite elaborate I will not repeat it here. The interested reader is referred to Tsotsos (1989) for the proof.

In his 1990 paper, Tsotsos wrote that Theorem 7.2 implies that the complexity of Bottom-up Visual Matching is inherently exponential in the number of pixels in the image, i.e., $O(2^{|I|})$. In a commentary, Kube (1990; see also Kube, 1991) replied that Tsotsos claim is incorrect because Bottom-up Visual Matching is not NP-hard “in the strong sense." To explain we consider the strategy for solving Bottom-up Visual Matching described by Kube (1991).

The strategy used by Kube (1991) is generally known as *dynamic programming* (e.g. Goodrich & Tamassia, 2002). The basic idea of dynamic programming is that the optimal solution to a problem contains an optimal sub-solution. Using the known optimality of the sub-solutions within sub-solutions the algorithm recursively builds an optimal solution. I sketch the algorithm by Kube (1991) below:

Kube's Algorithm

Input: An instance (I, T), with $I = \{p_1, p_2, \ldots, p_n\}$, for Bottom-up Visual Matching

Output: "yes" if (I, T) is a yes-instance for Bottom-up Visual Matching, and "no" otherwise.

[Description of algorithm:] Let BEST(i, j) be a function that outputs the maximum value of corr(I') over all subsets $I' \subseteq I$ with $I' = \{p_1, p_2, \ldots., p_k\}$, $k \leq i$, and diff$(I') \leq j$. Note that, if we know the value of BEST$(|I|, \theta)$, then we have solved Bottom-up Visual Matching. Namely, if BEST$(|I|, \theta) \geq \phi$ then we know that the answer is "yes" (there exists a subset $I' \subseteq I$ with corr$(I') \geq \phi$ and diff$(I') \leq \theta$), and if BEST$(|I|, \theta) \geq \phi$ the answer is "no." We compute BEST$(|I|, \theta)$ as follows:

(1) We define BEST$(i, j) = -\infty$ for all $i \leq 0$ and $j \leq 0$.

(2) For $j = 1, 2, \ldots, \theta$ we do the following: If diff$(p_1) \leq j$ then we define BEST$(1, j)$ = corr(p_1), else we define BEST$(1, j) = 0$.

(3) We set $i = |I|$ and $j = \theta$.

(4) We recursively apply the rule BEST(i, j) = MAX(BEST$(i-1, j)$, corr(p_i) + BEST$(i-1, j -$diff(p_i))).[75]

[75] To see that the rule is valid consider the following: Let I' be a subset of pixels such that corr(I') is maximum and diff$(I') \leq j$. In other words, BEST(i, j) = corr(I'). We distinguish two cases:

(1) Let $p_i \notin I'$. Then corr(I') = BEST(i, j) = BEST$(i-1, j)$, since BEST$(i-1, j)$ outputs the maximum value of corr(I^*) over all subsets $I^* \subseteq I$, with $I = \{p_1, p_2, \ldots., p_{i-1}\}$, and diff$(I') \leq j$.

If the computed value BEST($|I|$, θ) is such that BEST($|I|$, θ) $\geq \phi$, then we output "yes," else we output "no." [End of description]

In the recursion sketched above we compute BEST(i, j) only for values $0 < i \leq |I|$ and $0 \leq j \leq \theta$. Hence Kube's algorithm computes BEST($|I|$, θ) in time $O(|I|\ \theta)$. This function is not exponential in $|I|$, so indeed the statement by Tsotsos that the time-complexity of Bottom-up Visual Matching is inherently exponential in $|I|$ is hereby refuted.[76] Kube (1991) goes on to argue that Bottom-up Visual Matching is only hard if brightness values are very large. Let λ denote the largest brightness value (i.e., for all $p_i \in I$, we have $b_i \leq \lambda$). Then $\sum_{p\in I} \text{diff}(p) \leq \lambda|I|$. Now observe that, if $\sum_{p\in I} \text{diff}(p) \leq \lambda|I| \leq \theta$, then the problem Bottom-up Visual Matching is trivial.[77] Thus we conclude that for any non-trivial instance we have $\theta \leq \lambda|I|$, and thus Bottom-up Visual Matching is solvable in time $O(|I|^2\lambda)$. In sum, if λ or θ is small, then Bottom-up Visual Matching is practically feasible.

Although running times like $O(|I|\theta)$ and $O(|I|^2\lambda)$ may look like polynomial running times, it is important to realize that they are *not*.[78] Recall from Chapter 2 (page 28) that the complexity of an algorithm is measured in terms of the length of the input when encoded in a *reasonable* way. Since θ and λ are numbers, a reasonable encoding of them would be in n-ary with $n \geq 2$. Then the size of the encoding of θ is $O(\log_n\theta)$, and $O(|I|\theta)$ is not bounded by any polynomial of $O(|I|\ \log_n\theta)$, $n \geq 2$. Similarly, the size of the encoding of λ is $O(\log_n\lambda)$, and $O(|I|^2\lambda)$ is not bounded by any polynomial of $O(|I|^2 \log_n\lambda)$, $n \geq 2$ (see also Garey & Johnson, 1979).

---

(2) Let $p_i \in I'$. Then corr(I') = BEST(i, j) = corr(p_i) + BEST($i-1$, $\theta -$diff(p_i)), since BEST($i-1$, $\theta -$diff(p_i)) outputs the maximum value of corr(I^*) over all subsets $I^* \subseteq I = \{p_1, p_2, \ldots, p_{i-1}\}$ with diff(I^*) $\leq \theta -$diff(p_i).

Since corr(I') is *maximum* we conclude that BEST(i, j) = MAX(BEST($i-1, j$), corr(p_i) + BEST($i-1$, $\theta -$diff(p_i))).

[76] In his response to Kube (1991), Tsotsos (1991, p. 770) wrote that he "thought this was a good way of making the point for a noncomputational audience," even though it "sweeps much under the rug." Although I understand Tsotsos' motivation, in the present context I will have to go with Kube's criticism.

[77] Then we set $I' = I$, and if $\sum_{p\in I'} \text{corr}(p) \geq \phi$ we return "yes," and otherwise "no."

[78] Note that if they were, then Kube's algorithm, together with Theorem 7.2, would have proven P = NP.

However, if we were to place some bound on θ or λ, then $O(|I|\theta)$ and $O(|I|^2\lambda)$ would be polynomial-time functions. Importantly, this would *even* be the case if the bound were a polynomial function of $|I|$. For this reason, algorithms with this type of running time are called *pseudo-polynomial time* algorithms (Garey & Johnson, 1979). NP-hard problems that are not solvable by any pseudo-polynomial time algorithm (unless P = NP) are called *strong* NP-hard.[79]

I would like to bring to the reader's attention that a pseudo-polynomial time algorithm is a special case of an fpt-algorithm, viz. one in which the function $f(\kappa)$ is a polynomial function the parameter κ (although κ itself can be a non-polynomial in the size of the input). In other words, the pseudo-polynomial time algorithm described by Kube shows that θ-Bottom-up Visual Matching $\in$ FPT and λ-Bottom-up Visual Matching $\in$ FPT. Also note that Kube's arguments discussed above, mirror the FPT-Cognition thesis introduced in Chapter 3 (page 46): NP-hard problems may still be tractable, as long as their inherent exponential-time complexity can be captured by input parameters that are small.

7.3. Conclusion

In this chapter we have considered two problems, Min-Incomp-Lex (Martignon & Schmitt, 1999) and Bottom-up Visual Matching (Tsotsos, 1990). Both problems are very different from the problems discussed in Chapters 4–6, and they are very different from each other. I have illustrated how the techniques for complexity analysis used in previous chapters also apply to these problems. For example, I have presented a reduction rule to show that $|A|$-Min-Incomp-Lex is in FPT, and I showed how to construct a branching algorithm solving Min-Incomp-Lex in time $O\left(\frac{n!}{(n-k)!}n^2m^2\right)$. For the problem Bottom-up Visual Matching we have discussed Kube's (1991) pseudo-polynomial time algorithm, and I have explained that this algorithm is an fpt-algorithm.

The sole purpose of this chapter was to show how the techniques, discussed in detail in previous chapters, generalize to problems of very different character. As a result

[79] All NP-hard problems considered in this work (except for the number problems Knapsack and Bottom-up Visual Matching) are strong NP-hard.

the discussion has been relatively brief and superficial. Many parameters and questions remain unexplored. For example, we have seen that n-Min-Incomp-Lex $\in$ FPT and $\{n, k\}$-Min-Incomp-Lex $\in$ FPT. It would be very interesting to know if also k-Min-Incomp-Lex is in FPT. Further, Kube's pseudo-polynomial time algorithm does not use the fact that in Bottom-up Visual Matching there are strong restrictions on the types of values that diff(.) and corr(.) can take (see page 171). As a consequence, the algorithm also solves Knapsack in the same time. Is it possible to use the restrictions on Bottom-Up Visual Matching to derive faster fpt-algorithms for this problem than for Knapsack?

Chapter 8. Synthesis and Potential Objections

This chapter serves to synthesize the arguments and ideas pursued throughout this work. I start by recapitulating the main goal of this research and summarize how I have set out to attain that goal. As I anticipate that this research may give rise to objections by cognitive psychologists, I subsequently discuss a set of potential objections. For each objection I identify the theoretical perspective it reflects and I give a brief response.

8.1. Synthesis

The main aim of this research has been to make the theory of computational complexity tangible for the cognitive psychologist, such that s/he can study the *a priori* feasibility of computational level theories. Towards this end, I have presented a set of chapters—each chapter contributing to the main goal in its own way.

Chapter 1 explained that, at the computational level, cognitive theories *are* mathematical functions. Once it is recognized that cognitive systems (are thought to) 'compute' functions, the question "which functions can be computed by cognitive systems?" becomes a natural one to ask. The answer to this question can then serve as a guiding principle in the development of cognitive theories.

In the above, the meaning of 'compute' is key. Many cognitive psychologists have an intuitive idea of computation and computability, but often they lack a formal understanding. This can lead to all kinds of preconceptions about what types of mechanisms are, and are not, computational (see also Section 8.2). Chapter 2 was intended to remedy this problem, by presenting a brief but accessible introduction to the theory of computation. Furthermore, the exposition in this chapter led to the formulation of the Tractable Cognition thesis: Cognitive functions are among the *computationally tractable* functions.

Chapter 3 discussed how many cognitive psychologists subscribe to the P-Cognition thesis as a formalization of the Tractable Cognition thesis. This choice is motivated by the exclusive use of classical complexity theory in present-day cognitive psychology. Based on parameterized complexity theory, and its accompanying notion of fixed-parameter tractability, I have proposed the FPT-Cognition thesis as an alternative formalization of the Tractable Cognition thesis. I have explained that, despite its apparent plausibility, the P-Cognition thesis is overly restrictive and at risk of excluding veridical cognitive theories from empirical investigation. Unlike the P-Cognition thesis, the FPT-Cognition thesis recognizes that different aspects of a problem's input may contribute to its complexity in qualitatively different ways. As such, the FPT-Cognition thesis encourages the cognitive psychologist to engage in active investigation of problem parameters and their contribution to a problem's complexity. Chapters 5−7 illustrated possible forms that such an investigation may take.

What good is a tool if you do not know how to use it? To enable the reader to learn how to perform basic complexity analysis, I have tried to make the text as much self-contained as possible. Chapters 2 and 3 explained and illustrated basic notions and techniques in classical complexity theory. Furthermore, Chapter 4 presented a whole toolbox of basic techniques for parameterized complexity analysis. Chapters 2−4 have the added advantage that they can make the formal literature in computer science more accessible to the reader; hence facilitating searches for different tools in that literature.

Chapters 5−7 demonstrated the power and generality of the presented toolbox, by illustrating its use in analyses for four different cognitive tasks: Coherence, Subset Choice, Min-Incomp-Lex, and Bottom-up Visual Matching. Notably, in each case the analysis led to fundamental insights into the nature of the task. I believe that, after having read and understood these analyses, the reader is more likely to agree that complexity theory has a role to play in cognitive theory. Also, after having read these analyses, the reader will be in a much better position to appreciate the difference between the P-Cognition thesis and the FPT-Cognition thesis.

8.2. Potential Objections

It is my experience that the views and ideas expressed in this work give rise to questions and/or criticisms by cognitive psychologists. This section discusses a set of objections that I have encountered in discussions with colleagues. Each objection can be seen as arising from a particular theoretical perspective. Here I distinguish between three perspectives: The perspective of (A) a researcher who subscribes to the P-Cognition thesis, but who questions the FPT-Cognition thesis; (B) a researcher who subscribes to the computational approach to cognition, but who questions the Tractable Cognition thesis; and (C) a researcher who does not subscribe to the computational approach to cognition. Because an argument is best understood if one knows who is making it, I will indicate for each objection the perspective from which (I believe) it arises.

8.2.1. The Empiricist Argument.

(Perspective B or C) *Cognitive theories should be evaluated on how well they explain empirical data, not on* a priori *plausibility.*

The Tractable Cognition thesis is in no sense intended to replace empirical evaluation of cognitive theories. It is well understood that, in the end, all cognitive theories must stand the test of empirical scrutiny. What the Tractable Cognition thesis offers is a way of evaluating the *a priori* feasibility of cognitive theories on theoretical grounds. This way, the thesis helps constrain the vast space of possible computational level theories for any given cognitive task. Further, the Tractable Cognition thesis is useful for evaluating (aspects of) cognitive theories that cannot (yet) be evaluated (solely) on empirical grounds. This seems particularly helpful since many cognitive theories are about unobservable, or only indirectly observable, cognitive processes.

Of course, if one is a non-computationalist cognitive scientist then one may not recognize the tractability constraint on cognitive theories. In that case see my response to the *Cognition-is-not-Computation Argument* below.

8.2.2. The Cognition-is-not-Computation Argument

(Perspective C) *Computational complexity theory has nothing to offer cognitive science, since cognition is not computation.*

What is meant with this objection depends crucially on the meaning of the phrase "cognition is not computation." I believe the phrase is associated with a multitude of meanings. In my reactions below, I distinguish between four possible versions of the argument.

Version 1: *Complexity analysis does not apply to cognition because cognition is not symbolic computation.*

In cognitive science, the terms computation and computationalism have become associated with the symbolic tradition (also referred to as *good-old-fashioned-artificial-*

intelligence or GOFAI), and sometimes even with particular models in this tradition (e.g. Anderson, 1987; Fodor, 1987; Pylyshyn, 1984, 1991; Newell & Simon, 1988a, 1988b). As explained in Chapters 1 and 2, to recognize the role of complexity theory in cognitive science all that is required is a commitment to computationalism in some form or another—not to any particular form of computationalism (cf. Chalmer's, 1994, *minimal computationalism*). Many theories that are considered 'non-computational' may still fall under the heading of computationalism in this broad sense. For example, despite their presumed 'non-computational' status (e.g. Port & van Gelder, 1995; Thelen & Smith, 1994; van Gelder, 1995, 1998, 1999), dynamical systems models can be subjected to computational complexity analysis (unless Version 2 applies).

> Version 2: *Complexity theory does not apply to cognitive systems, because cognitive systems do not compute functions.*

The present work is based on the idea that cognitive systems are to be understood in terms of the input-output mappings that they realize; i.e., in terms of the functions that they compute (see Chapter 1). It may be, however, that the purpose of some (or all) cognitive systems is not to compute any functions at all. Instead, for example, their purpose may be to cycle through a set of states indefinitely, without ever halting and producing an output (cf. Levesque, 1988, p. 385). In such cases we can reasonably say that the system is computing (in the sense that each state leads to a different state in a deterministic way), but the system cannot be said to be computing a *function* (since it never produces an output). As should be clear, such cognitive systems are precluded from the type of analyses presented here (see also Version 4 of the *Cognition-is-not-Computation Argument*).

> Version 3: *Cognitive functions need not be computationally tractable, because cognitive systems realize their input-output mappings via non-computational means.*

Some non-computationalists do not question that the purpose of cognitive systems is to realize input-output mappings (i.e., they do not subscribe to Version 2), but they propose that cognitive systems realize such mappings in 'non-computational' ways (e.g. Horgan & Tienson, 1996; Krueger & Tsav, 1990; but see also my comment to Version 1). On this view, cognitive functions need not be computationally tractable. This may be so. But the *explanatory* value of theories that propose that a (potentially intractable) cognitive function is realized in some non-computational way is questionable (see also Cherniak, 1986; Levesque, 1988). As Tsotsos puts it:

> "Experimental scientists attempt to explain their data, not just describe it (…) There is no appeal to non-determinism or to oracles that guess the right answer or to undefined, unjustified, or "undreamed of" mechanisms that solve difficult components. Can you imagine theories that do have these characteristics passing a peer-review procedure?" (Tsotsos, 1990, p. 466).

> Version 4: *Complexity theory does not apply to cognition, because computation is an altogether wrong way of thinking about cognition.*

Finally, Version 4 of the *Cognition-is-not-Computation Argument* represents the non-computationalist that is not persuaded by any of my reactions to Versions 1–3. In my opinion, even this researcher should appreciate the contribution that the Tractable Cognition thesis makes to cognitive science. Namely, a non-computationalist can still recognize that tractability is a constraint on *computational* theories of cognition. Then the Tractable Cognition thesis offers the non-computationalist a way of evaluating the success of his/her competition. If, in the long run, human cognition systematically defies *tractable* computational description then this can be taken as empirical support for the idea that computation is the wrong way of thinking about cognition.

8.2.3. The Super-Human Argument

> (Perspective B or C) *Humans are found to perform computationally intractable tasks. This goes to show that tractability is not a constraint on human computation.*

Some tasks that are performed effortlessly by humans are presently being modeled by computational intractable functions (e.g. Haselager, 1997; Oaksford & Chater, 1993, 1998). Some researchers interpret this as evidence that people perform computationally intractable tasks (e.g. Siegel, 1990). In my view the argument is flawed. There are two possibilities: either one is a computationalist (Perspective B) or one is not (Perspective C). If one is, then one should concede that either the tasks are incorrectly modeled or that the wrong criterion for tractability has been adopted. If one is not, then one does not recognize that the models truly capture the nature of the tasks in the first place, and thus their classification as 'intractable' is irrelevant the conceptualization of the tasks. (Unless Version 3 of the *Cognition-is-not-Computation Argument* applies. In that case see my response on page 179).

8.2.4. The Heuristics Argument

(Perspective B) *Humans may approach intractable cognitive tasks by using heuristics/approximation algorithms. Then intractability is not an issue.*

This argument we already encountered in Chapter 3 (page 40), and my reaction still stands: Since heuristics do not solve the problem they are used for, they are unsatisfactory algorithmic level descriptions. If one wishes to maintain that a 'heuristic' *M* is a satisfactory algorithmic description of a cognitive system, then one has to concede that the task being solved at the computational level is the task solved by *M*, not some different task. The story is different for approximation algorithms, since in those cases there is a provable and lawful relationship between the behavior of the algorithm and the problem Π that it approximates. However, if indeed an approximate solution serves just as well, then the computational level theory should incorporate this aspect of the task, making the computational level theory Π an approximation problem Π', and making the approximation algorithm for Π an exact algorithm for Π'.

One might counter that my argument makes sense for descriptive but not for *normative* cognitive theories. Indeed normative theories just need to be sound—not necessarily tractable. However, when evaluating how cognitive systems fare in comparison to normative models it may not make sense to compare them to standards that are physically unrealizable in the first place. For example, Oaksford and Chater (1998, p. 113), argue that expecting humans to behave rationally on intractable tasks is like expecting them to be able to "breath under water, even though [they] do not possess gills." (cf. Cherniak, 1986; Frixione, 2001; Gigerenzer & Goldstein, 1996; Oaksford & Chater, 1993; Simon, 1990; Todd & Gigerenzer, 2000).

8.2.5. The Average-case Argument

(Perspective B) *A task that is classified as intractable on a worst-case analysis may still be tractable in practice. An average-case measure of complexity should be used instead.*

In this work we adopted a worst-case measure of complexity. The use of this measure is validated by the fact that the worst-case happens in practice (if it would not, then we would be dealing with a restricted version of the problem, which would have a different worst-case). Furthermore, for the purpose of determining the complexity of a task, worst-case and average-case analysis often leads to the same conclusion (see also Totsos, 1990). Consider, for example, the task of finding a particular number, s, in a list of n numbers. Assume that s occurs exactly once in any given list and that numbers in a list appear in any arbitrary order. Then, in the worst-case we consider n numbers in the list before we find s. If we perform this task many times, each time for a new list, then on average we consider $\frac{n}{2}$ numbers before we find s. Note that $\frac{n}{2}$ is on the same order of magnitude as n—thus both are *polynomial*. Similarly, to find a particular subset among all possible subsets on n elements, we need to consider 2^n subsets in the worst case, and 2^{n-1} subsets on average—both are *exponential*.

This is not to say that worst-case analysis and average-case analysis cannot lead to different conclusions about complexity. This may happen, for example, if certain inputs are much more likely to occur than others. Not only is it often difficult (or even impossible) to know the probability distribution on inputs, but the introduction of probability distributions makes complexity analysis much more difficult. In those cases, worst-case analysis of the problem with restricted inputs (excluding inputs with very low probability of occurrence) may provide a reasonable alternative to average-case analysis.

8.2.6. The Parallelism Argument

(Perspective B) *Cognitive computation is (to a large extent) parallel, not serial. A task that is intractable for a serial machine need not be intractable for a parallel machine.*

We have worked with the serial Turing machine model here. Nonetheless, the arguments for a Tractable Cognition thesis can be extended to include parallel computation (see also

Frixione, 2001). From a complexity perspective the difference between serial and parallel computation may be quite insubstantial, depending on the particular parallel machine model used. To illustrate, let us first consider a parallel machine *M* with *S* processing channels, such that *M* computes a given serial computation by performing *S* steps in parallel (i.e. simultaneously). Further, let Π be a function with time-complexity $O(f(n))$. Then *M* computes Π at best in time $O(\frac{f(n)}{S})$.[80] Note that, if *S* is a constant, then the speed-up due to parallelization is by a constant factor only (see also Chapter 3, page 32), and $O(\frac{f(n)}{S}) = O(f(n))$. The speed-up factor *S* can of course be taken into account in the analysis—there is nothing inherent in complexity theory that prevents one from doing so. Importantly though, if $f(n)$ is a *non-polynomial* function then the speed-up due to *S* becomes negligibly small very fast as *n* grows (see also Table 2.1 on page 32). The same is true if *S* is bounded by some polynomial function of *n*. Of course, if *S* grows *non-polynomially* as a function of *n*, then $O(\frac{f(n)}{S})$ may be a polynomial running-time. In that case indeed time would no longer be a limiting factor, but the space required for implementing the astronomical number of processors would be (cf. Frixione, 2001).

In other models of parallel computation the speed-up due to *S* need not be constant, but may grow with *n*.[81] Importantly, though, the Invariance thesis, as discussed in Section 2.4.4 on page 28, includes both serial and parallel computation: It is widely believed that for any reasonable parallel machine the speed-up due to *S* will be by at most a *polynomial amount* (Frixione, 2001; Tsotsos, 1990; Parberry, 1994). In other words, if the Invariance thesis is true, then parallel machines cannot compute functions outside P in polynomial-time nor compute parameterized functions outside FPT in fpt-time.

---

80 Here we are assuming that Π is parallelizable in this way. This may not be possible for all functions.

81 This is the case, for example, in the parallel random access machine (P-RAM) model, where *M* is assumed to have available *S* processors that can all communicate to each other in constant time (e.g. Gibbons & Rytter, 1988; cf. the extensions of the Turing machine concept discussed on page 17).

8.2.7. The Non-Determinism Argument

(Perspective B or C) C*ognitive systems are not deterministic machines, and thus functions that are intractable for deterministic machines may still be tractable for cognitive systems.*

The approach we have taken here assumes a deterministic worldview, and not, for example, a probabilistic worldview. In Chapter 3 we have seen how some functions that cannot be computed in polynomial-time by any deterministic Turing machine (unless P = NP), can be computed in polynomial-time by a non-deterministic Turing machine. Because a reader may mistakenly assume that non-deterministic computation—as defined in computer science—is the same as "non-deterministic" computation in the sense of *stochastic* computation (see e.g. Martignon & Hoffrage, 2002), I will clarify the important difference below.

Let M be a non-deterministic Turing machine. Recall from Chapter 2, that M is said to "compute" a function Π if, for every possible input i, there exists at least one possible sequence of state transitions in M that leads to output $o = \Pi(i)$. The number of possible different outputs that M can generate for any fixed i is ignored in this definition, and may be arbitrarily large. In other words, M can be seen as operating like an oracle that always "guesses" the right output for any given input. We now define a stochastic interpretation of M, called M', as follows: The transition relation for M' is the same as for M, but in M' each possible transition has an associated probability that it occurs. Now it becomes clear that, if M has an arbitrarily large number of possible outputs, then the probability that M' "guesses" the right output for any given function may be arbitrarily small. In other words, if M "computes" a function $\Pi(i)$ for a given i, its stochastic version M' may "compute" $\Pi(i)$ with only arbitrarily small probability.

This is not to say that stochastic computational models of cognitive systems are impractical or useless. On the contrary, they may very well provide better models of certain cognitive systems than deterministic models. But one should not be fooled into thinking that stochastic machines have the same abilities as non-deterministic machines. Even if it turns out that stochastic machines have different abilities than deterministic machines, this does not obviate complexity analyses in cognitive psychology. Tractability is as much a requirement on stochastic computation, as it is on deterministic computation.

8.2.8. The Small-Inputs Argument

(Perspective B) *For some cognitive tasks, the size of the input is small in practice. In those cases intractability is not an issue.*

Correct. This reasoning is also naturally captured by the FPT-Cognition thesis. As explained in Chapter 2 (page 47) and Chapter 4 (page 64), all problems are in FPT when parameterized by their input size. Thus, if the input size is small then, on the FPT-Cognition thesis, the problem is classified as *tractable*. It should be noted that also no one who subscribes to the P-Cognition thesis would claim that tractability is an issue if input size is small. The problem is, of course, that for many cognitive tasks the size of the input as a whole is *not* small (or at least not small *enough*). It is then that the P-Cognition thesis and the FPT-Cognition thesis diverge.

8.2.9. The Nothing-New Argument

(Perspective A) *The requirement that a problem be in FPT for some "small" input parameters is not essentially different from the requirement that a special case of the problem is in P.*

The claim reflects a misunderstanding about the relationship between P and FPT. Parameterization is not the same as problem restriction; it just determines how we analyze the complexity of a problem. If a problem Π, with input parameter κ, can be solved by an algorithm that runs in time $O(f(\kappa)n^{\alpha})$, where α is a constant and $f(\kappa)$ is a function depending only on κ, *not* on $|i|$, then we say that the parameterization κ-Π is in FPT. It is important to realize that in this analysis the value of κ is *not* fixed[82]—it remains a variable just like $|i|$. It is true that if we were to fix κ then $O(f(\kappa)n^{\alpha})$ would be $O(n^{\alpha})$, and thus polynomial time, but the same is true also for some fixed-parameter intractable problems; e.g., a running time $O(n^{\kappa})$ is not fpt-time for parameter κ but is polynomial-time if κ is a constant. In this respect, the requirement that a problem be in FPT for some parameter κ is more stringent than the requirement that a problem is in P for constant κ. However, the requirement that the problem be in FPT for parameter κ is more lenient than the requirement that the problem be in P.

[82] In this respect, the name 'fixed-parameter tractable' may be somewhat misleading.

8.2.10. The P-is-not-strict-enough Argument

(Perspective A) *Polynomial-time computability is already a too liberal constraint on cognitive functions. Thus the FPT-cognition thesis only makes matters worse.*

Some researchers subscribe to the idea that most, if not all, (higher-order) cognitive functions are of extremely low complexity (e.g. Gigerenzer & Goldstein, 1996; Martignon & Hoffrage, 1999, 2002; Martignon & Schmitt, 1999; Todd & Gigerenzer, 2000). For example, cognitive tasks "requiring more than a quadratically growing number of computation steps already appear to be excessively complicated" to Martignon and Schmitt (1999, p. 566). To such researchers, the proposal that cognitive systems perform non-polynomial fpt-time computations may seem outrageous.

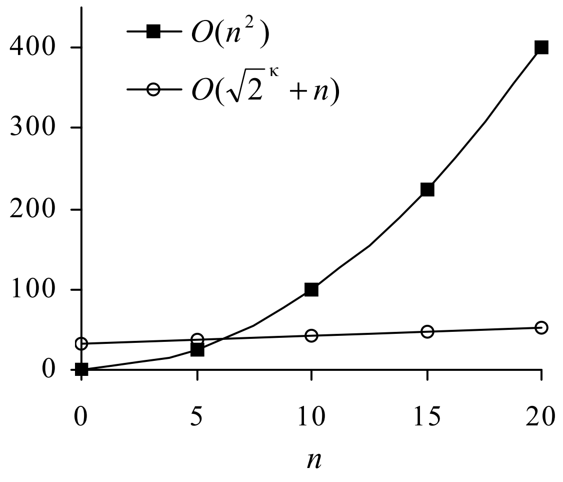


Figure 8.1. Polynomial time versus fpt-time.
The figure compares polynomial time $O(n^2)$ and fpt-time $O(\sqrt{2}^{\kappa}+n)$ for parameter κ. In this example κ = 10. Increasing the value of κ will cause the curve for the function $O(\sqrt{2}^{\kappa}+n)$ to translate upwards.

First, I comment that quadratic running times are associated with even relatively simple (heuristic) strategies (see e.g. the Greedy Vertex Cover algorithm in Chapter 2; and the LEX algorithm in Chapter 7). Second, and more importantly, a non-polynomial time algorithm need not be slower than a polynomial time algorithm—all depends on the size of the parameters. Figure 8.1 illustrates that, for example, an algorithm that runs in non-polynomial fpt-time $O(\sqrt{2}^{\kappa} + n)$ can be much faster than an algorithm that runs in polynomial time $O(n^2)$, provided only that the parameter κ is small.

Chapter 9. Summary and Conclusions

This research contributes to cognitive psychological theory on two different levels: At the metatheoretical level this dissertation explicates and clarifies existing arguments, as well as adds new arguments, for the role of complexity theory in cognitive psychology. Second, at the level of specific cognitive theories, this research has led to (1) a set of new complexity results and (2) the identification of some open problems that can guide and motivate new research on these theories. Sections 9.1 and 9.2 review these contributions respectively. I close, in Section 9.3, with ideas about future directions.

9.1. Metatheoretical Contributions

The main contributions of this research are to be found at the metatheoretical level: By adopting, adapting and synthesizing both new and existing ideas this research adds to the continuing discussion among cognitive scientists about the nature of cognition—how best to characterize it and how best to study it. To explicate this contribution I present here a list of what are the core metatheoretical contributions of this research:

- I have explicated and clarified existing arguments for the thesis that cognitive functions are constrained by tractability requirements: the Tractable Cognition thesis (Chapters 1−3 and 8).
- The Tractable Cognition thesis is traditionally seen as equivalent to the P-Cognition thesis. I have reformulated the Tractable Cognition thesis as an informal thesis (Chapter 2), and I have explicated that the P-Cognition thesis is but one possible formalization of the Tractable Cognition thesis (Chapter 3).
- Classical complexity theory has influenced cognitive psychology for some time now, but parameterized complexity theory has so far remained unknown in cognitive psychology. I have shown how parameterized complexity theory naturally extends the application of complexity theory in cognitive psychology (Chapters 3, 5−8).
- I have explained how parameterized complexity theory, and its accompanying notion of fixed-parameter tractability, challenges the P-Cognition thesis (Chapters 3, 5−8).

- I have proposed and defended the FPT-Cognition thesis as an alternative formalization of the Tractable Cognition thesis (Chapter 3).
- By introducing techniques from the theory of parameterized complexity (Chapter 4) and illustrating their use in the analysis of several existing cognitive theories (Chapters 5–7), I have illustrated how the FPT-Cognition thesis may be put into practice.
- I have reviewed a set of potential objections of the Tractable Cognition thesis, the FPT-Cognition thesis and the use of complexity theory in cognitive psychology in general (Chapters 8). For each objection I have presented a brief reply to either rebut or qualify the objection.

9.2. Theory Specific Contributions

In Chapters 5–7, I have presented detailed analyses of four cognitive tasks and their variants: Coherence, Subset Choice, Min-Incomp-Lex, and Bottom-up Visual Matching. The aim of these analyses was to illustrate how to perform complexity analyses of cognitive theories. As a consequence, we have also obtained new insights into the complexity of these specific tasks, demonstrated by the list of new complexity results summarized in Table 8.1. Furthermore, the investigations naturally lead to the identification of open problems that may guide and motivate further research on these tasks. In the following I discuss 16 open problems. For each open problem I give a brief motivation, always assuming P ≠ NP and FPT ≠ W[1].

Is Single-Element Coherence easier than Coherence?

Thagard (2000) proposed that the problem Coherence can be used to model, for example, jury decision-making. In Section 5.4.3, we have considered two possible ways in which Coherence may model jury decision-making (cf. Thagard, 1989, 2000). This led to the formulation of two new problems: Single-Element Discriminating Coherence and Single-Element Foundational Coherence. We have found that—unlike Coherence—Single-Element Discriminating Coherence and Single-Element Foundational Coherence are in P for inputs with $D = \varnothing$. The classical complexity of these two problems on general inputs remains unknown however.

Table 9.1. Overview of complexity results

Coherence and Related Problems	
Coherence on networks with $C^+ = \varnothing$ and $w(e) = 1$ for all $e \in E$ is NP-hard	Corollary 5.3, page 84
Coherence on consistent networks is in P	Theorem 5.1, page 86
Coherence on trees is in P	Lemma 5.4, page 89
Discriminating Coherence is NP-hard	Corollary 5.6, page 91
Foundational Coherence is NP-hard	Corollary 5.7, page 91
Single-Element Discriminating Coherence with $D = \varnothing$ is in P	Corollary 5.8, page 93
Single-Element Foundational Coherence with $D = \varnothing$ is in P	Corollary 5.9, page 93
c-Double-Constraint Coherence $\in$ FPT	Theorem 5.2, page 103
c-Coherence $\in$ FPT	Corollary 5.10, page 103
Pos-Annotated Coherence $\in$ P	Lemma 5.11, page 113
$\|P^-\|$-Annotated Coherence $\in$ FPT	Theorem 5.3, page 114
$\|C^-\|$-Annotated Coherence $\in$ FPT	Corollary 5.12, page 114
$\|C^-\|$-Coherence $\in$ FPT	Corollary 5.13, page 114
$\|P^-\|$- Coherence $\in$ FPT	Corollary 5.14, page 114
Subset Choice and Related Problems	
Subset Choice is NP-hard	Theorem 6.1, page 122
UCG Subset Choice is NP-hard	Corollary 6.1, page 124
p-UCG Subset Choice is W[1]-hard	Theorem 6.2, page 128
q-UCG Subset Choice $\in$ FPT	Theorem 6.3, page 130
q-ECG Subset Choice $\in$ FPT	Corollary 6.7, page 137
q-VCG Subset Choice is W[1]-hard	Lemma 6.5, page 138
$\{q, \Omega_V\}$-CG Subset Choice $\in$ FPT	Theorem 6.5, page 139
$\{\Delta, q\}$-CG Subset Choice $\in$ FPT	Lemma 6.6, page 141
$\{q, \varepsilon, \Omega_V\}$-CH Subset Choice $\in$ FPT	Theorem 6.6, page 144
$\{q, \theta\}$-CH Subset Choice $\in$ FPT	Lemma 6.8, page 145
$\{q, \varepsilon, \Delta\}$-CH Subset Choice $\in$ FPT	Corollary 6.11, page 146
USG Subset Choice $\in$ P	Theorem 6.7, page 149
USG Exact-bound Subset Choice is NP-hard	Theorem 6.8, page 151
$\{p, k\}$-USG Exact-bound Subset Choice is W[1]-hard	Corollary 6.15, page 153
Min-Incomp-Lex	
$\|A\|$-Min-Incomp-Lex $\in$ FPT	Sketched on page 163
$\|F\|$-Min-Incomp-Lex $\in$ FPT	Sketched on page 166
Bottom-up Visual Matching	
θ-Bottom-up Visual Matching $\in$ FPT	Sketched on page 174
λ-Bottom-up Visual Matching $\in$ FPT	Sketched on page 174

Open Problem 1. Is Single-Element Discriminating Coherence in P?

Open Problem 2. Is Single-Element Foundational Coherence in P?

Can we solve *c*-Coherence faster?

We have shown that the NP-hard problem Coherence is in FPT for parameter c. Section 5.5 described a non-constructive algorithm for c-Coherence that runs in time $O(2^c + |P|)$ and Section 5.6 described a constructive algorithm that runs in time $O(2.52^c + |P|^2)$. It is of interest to know if these running times can be improved.

Open problem 3. Do there exist faster non-constructive and/or constructive fpt-algorithms for c-Coherence?

Is ι-Coherence in FPT?

We have shown that the natural parameterization of Coherence, c-Coherence, is in FPT. But what about the relational parameterization ι-Coherence, with $\iota = \sum_{(p,q) \in C} w(p,q) - c$?

Open problem 4. Is ι-Coherence in FPT?

Note that if ι-Coherence $\in$ FPT then that means that Coherence is computationally tractable both for small *and* large c (since large c correspond to small ι).

Can we solve $|P^-|$-Coherence and $|C^-|$-Coherence faster?

In Section 5.7 we have shown that $|P^-|$-Coherence is in FPT with a running time of $O(2^{|P^-|} |P|^3)$, and $|C^-|$-Coherence is in FPT with a running time $O(2^{|C^-|} |P|^3)$. It is of interest to study if these running times can be improved.

Open problem 5. Do faster fpt-algorithms for $|P^-|$-Coherence exist?

Open problem 6. Do faster fpt-algorithms for $|C^-|$-Coherence exist?

Is $\{q, \varepsilon, \Omega_V\}$ a crucial source of complexity in Subset Choice?

In Section 6.6 we have shown that κ-CH Subset Choice $\notin$ FPT (unless FPT = W[1]), for $\kappa = \{q\}$, $\kappa = \{\varepsilon\}$, $\kappa = \{\Omega_V\}$, $\kappa = \{q, \varepsilon\}$, and $\kappa = \{\varepsilon, \Omega_V\}$. For the parameter set $\kappa = \{q, \Omega_V\}$ it remains unknown whether or not κ-CH Subset Choice is in FPT. Hence, we have the following open problem:

Open problem 7. Is $\{q, \varepsilon, \Omega_V\}$ or $\{q, \Omega_V\}$ a crucial source of complexity in CH Subset Choice?

Is Subset Choice computationally easy on surplus (hyper)graphs?

We have shown that Subset Choice on unit-weighted surplus graphs is in P, but the classical complexity of Subset Choice on general surplus graphs and hypergraphs remains unknown:

Open problem 8. Is SG Subset Choice in P?

Open problem 9. Is SH Subset Choice in P?

Which results generalize to Subset Choice with size restrictions?

In Chapter 6 we studied mostly Subset Choice problems *without* restrictions on the *size* of the chosen subset. In many practical settings subset size restrictions may apply, and thus it is of interest to also study the complexity of Subset Choice problems with various types of size restrictions (e.g. upper bound, lower bound or exact bound). As an example, we have considered the problem Exact-bound Subset Choice. Interestingly, we found that although Subset Choice is in P on unit-weighted surplus graphs, Exact-bound Subset Choice is NP-hard on unit-weighted surplus graphs. This shows that complexity results obtained for Subset Choice problems without subset size restrictions may no longer hold when subset size restrictions are imposed.

Open problem 10. Which of the complexity results obtained for Subset Choice, and its special cases, generalize to Exact-bound Subset Choice?

Open problem 11. Which of the complexity results obtained for Subset Choice, and its special cases, generalize to Lower-bound Subset Choice?

Open problem 12. Which of the complexity results obtained for Subset Choice, and its special cases, generalize to Upper-bound Subset Choice?

Can we solve *m*-Min-Incomp-Lex faster?

In Section 7.1.3 we have shown that m-Min-Incomp-Lex is in FPT with a running time of $O(2^m!\ n^2m^2)$. This running time is impractical even for relatively small m.

Open problem 13. Do faster fpt-algorithms for m-Min-Incomp-Lex exist?

Can we solve {*n*, *k*}-Min-Incomp-Lex faster?

In Section 7.1.3 we presented two reduction rules and one branching rule for $\{n, k\}$-Annotated Min-Incomp-Lex. Using these rules we derived an fpt-algorithm for $\{n, k\}$-

Min-Incomp-Lex that runs in time $O\left(\frac{n!}{(n-k)!}n^2m^2\right)$. Whenever $k < n$, this running time improves upon the running time $O(n!\ nm^2)$ for a naïve exhaustive search.

Open problem 14. Is it possible to derive more reduction rules and branching rules for $\{n, k\}$-Annotated Min-Incomp-Lex so as to improve the running time for $\{n, k\}$-Min-Incomp-Lex even more?

Is *k*-Min-Incomp-Lex in FPT?

We know that $\{n, k\}$-Min-Incomp-Lex $\in$ FPT and n-Min-Incomp-Lex $\in$ FPT. However, the parameterized complexity of k-Min-Incomp-Lex remains unknown:

Open problem 15. Is k-Min-Incomp-Lex in FPT?

Can we solve Bottom-up Visual Matching faster?

In Section 7.2.3 we have discussed how Kube's (1991) pseudo-polynomial time algorithm solves the problem Bottom-up Visual Matching in time $O(|I|\theta)$ and $O(|I|^2\lambda)$. Kube's algorithm can also be used to solve the more general problem Knapsack in the same time. We have seen that in Bottom-up Visual Matching there is a strong relationship between the corr(p) and the diff(p) for any pixel $p \in I$, while in Knapsack for an element $u \in U$, size s(u) and value v(u) are unrelated.

Open problem 16. Is it possible to exploit the relationship between corr(p) and diff(p) to build faster and/or different fpt-algorithms for Bottom-up Visual Matching than for Knapsack?

9.3. Musings on the Future

Computer scientists and AI researchers routinely analyze the complexity of their algorithms. As Tsotsos (1989, p. 1571) writes, "this is simply good computer science." Hopefully, one day, cognitive scientists and psychologists will consider it *simply good cognitive science* to analyze the complexity of their cognitive theories.

This work embodies one of several attempts to establish complexity theory as a standard analytic tool in cognitive psychology (see e.g. Frixione, 2001; Levesque, 1988; Tsotsos, 1990, for other attempts). Of course, large-scale utilization of this tool not only requires recognition of its value, but also the means to use it. For this reason, cognitive science could greatly benefit from a database of complexity results on existing cognitive theories (cf. Downey & Fellows, 1999; Garey & Johnson, 1979; see also Appendix B). As demonstrated here, once complexity results are known for many different problems, proving results for new problems becomes much easier.

I have argued that cognitive theories are constrained by tractability requirements. This does not mean, however, that we should be out to simply classify cognitive theories as either tractable or intractable. On the contrary, upon finding that a given cognitive theory is intractable, we should investigate all kinds of different versions and aspects of that theory in search of tractable special cases and/or parameterizations. This search can provide invaluable insights into the nature of a task, far beyond what a 'tractable *versus* intractable' classification has to offer (cf. Downey & Fellows, 1999; Garey & Johnson, 1979; Nebel, 1996; Wareham, 1996). Also, using complexity analysis in this way, we can discover ways in which cognitive systems can cope with complexity in different types of problem environments (i.e. for different parameter ranges)—potentially leading to psychological hypotheses that can be empirically tested. As always, the proof of the pudding will be in the eating, but I suspect that in the end the contributions of complexity theory to cognitive science will far surpass what we can envision at this point.

References

Alber, J., Fan, H., Fellows, M. R., Fernau, H., Niedermeier, R., Rosamond, F. & Stege, U. (2001). Refined search tree technique for dominating set on planar graphs. *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science, LNCS 2136* (pp. 111-122). Berlin: Springer-Verlag.

Anderson, J. R. (1987). Methodologies for studying human knowledge. *Behavioral and Brain Sciences, 10*, 467-505.

Anderson, J. R. (1990). *The adaptive character of thought.* Hillsdale, NJ: Lawrence Erlbaum Publishers.

Balasubramanian, R., Fellows, M. R., & Raman, V. (1998). An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters, 65,* 163-168.

Barton, G. E., Berwick, R. C., & Ristad, E. S. (1987). *Computational complexity and natural language.* Cambridge, MA: MIT Press.

Bechtel, W. (1988). *Philosophy of mind: An overview for cognitive science*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Block, N. (1980). Introduction: What is functionalism? In N. Block (Ed), *Readings in philosophy of psychology (171-184).* Cambridge, MA: Harvard University Press.

Bossert, W. (1989). On The extensions of preferences over a set to the power set: An Axiomatic characterization of a quasi ordering. *Journal of Economic Theory, 49*, 84-92.

Brandstädt, A., Le, V. B., & Spinrad, J. P. (1999). *Graph classes: A survey*. Philadelphia: SIAM.

Chalmers, D. J. (1994). A Computational Foundation for the Study of Cognition. *Technical report in Philosophy-Neuroscience-Psychology*. Washington University.

Chen, J., Kanj, I. A., & Jai, W. (2001). Vertex Cover: Further observations and further improvements. *Journal of Algorithms, 41*(2), 280-301.

Cherniak, C. (1986). *Minimal rationality.* Cambridge, MA: MIT Press.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics, 58*(2), 345-363.

Cleland, C. E. (1993). Is the Church-Turing thesis true? *Minds and Machines, 3*, 283-312.

Cleland, C. E. (1995). Effective procedures and computable functions. *Minds and Machines, 5*, 9-23.

Cook, S. (1971). The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 151-158.

Cooper, G. F. (1990). The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence, 42,* 393-405.

Copeland, B. J. (2002). Accelerating Turing machines. *Minds and Machines, 12*, 281-301.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L.(1990). *Introduction to Algorithms*. Cambridge, MA: MIT Press and McGraw-Hill.

Deutsch, D. (1985). Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A, 400*, 97-117.

Downey, R. G. & Fellows, M. R. (1999). *Parameterized complexity.* New York: Springer-Verlag.

Downey, R. G., Fellows, M. R., & Stege, U. (1999a). Parameterized complexity: A framework for systematically confronting computational intractability. *Contemporary trends in discrete mathematics: From DIMACS and DIMATIA to the future, 49*, 49-99.

Downey, R. G., Fellows, M. R., & Stege, U. (1999b). Computational tractability: The view from Mars. *Bulletin of the European Association for Theoretical Computer Science, 69*, 73-97.

Dunn, J. C. (2003). The elusive dissociation. *Cortex, 39*(1), 177-179.

Dunn, J. C. & Kirsner, K. (2003). What can we infer from double dissociations? *Cortex, 39*(1), 1-7.

Eliasmith, C. (2000). Is the brain analog or digital? The solution and its consequences for cognitive science. *Cognitive Science Quarterly, 1*(2), 147-170.

Eliasmith, C. (2001). Attractive and in-discrete: A critique of two putative virtues of the dynamicist theory of mind. *Minds and Machines, 11,* 417-426.

Eliasmith, C. & Thagard, P. (1997). Waves, particles, and explanatory coherence. *British Journal for the Philosophy of Science, 48,* 1-19.

Eysenck, M. W. & Keane, M. T. (1994). *Cognitive psychology: A student's handbook.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Falmagne, J.-Cl. & Regenwetter, M. (1996). A random utility model for approval voting. *Journal of Mathematical Psychology, 40*, 152-159.

Farquhar, P. H. & Rao, V. R. (1976). A balance model for evaluating subsets of multiattributed items. *Management Science, 22*(5), 528-539.

Fellows, M. R. (2002). Parameterized complexity: The main ideas and connections to practical computing. In R. Fleischer, B. Moret, & E. Meineche Schmidt (Eds.), *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software, LNCS 2547* (pp. 51-77). Berlin: Springer-Verlag.

Fellows, M. R. & McCartin, C. (1999). Personal communication.

Fellows, M. R., McCartin, C., Rosamond, F., & Stege, U. (2000). Coordinatized kernels and catalytic reductions: An improved FPT algorithm for Max Leaf Spanning Tree and other problems. In S. Kapoor & S. Prasad (Eds.), *Foundations of Software Technology and Theoretical Computer Science, LNCS 1974* (pp. 240-251). Berlin: Springer-Verlag.

Fishburn, P. C. & LaValle, I. H. (1993). Subset preferences in linear and nonlinear utility theory. *Journal of Mathematical Psychology, 37,* 611-623.

Fishburn, P. C. & LaValle, I. H. (1996). Binary interactions and subset choice. *European Journal of Operational Research, 92,* 182-192.

Fodor, J. A. (1987). Psychosemantics: The problem of meaning in the philosophy of mind. Cambridge, MA: MIT Press.

Foulds, L.R. (1992). *Graph theory applications*. New York: Springer-Verlag.

Frixione, M. (2001). Tractable competence. *Minds and Machines, 11*, 379-397.

Gandy, R. (1988). The confluence of ideas in 1936. In R. Herken (Ed.), *The universal Turing machine: A half-century survey* (pp. 55-111). New York: Oxford University Press.

Garey, M. R. & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. New York: Freeman.

Gibbons, A. & Rytter, W. (1988). *Efficient parallel algorithms.* Cambridge, UK: Cambridge University Press.

Gigerenzer, G. & Goldstein, D. G. (1996). Reasoning the fast and frugal way: Models of bounded rationality. P*sychological Review, 103*(4), 650-669.

Glenberg, A. M. (1997). What memory is for. *Behavioral and Brain Sciences, 20*,1-55.

Goodrich, M. T. & Tamassia, R. (2002). *Algorithm design: Foundations, analysis, and internet examples.* New York: John Wiley & Sons, Inc.

Gottlob, G., Scarcello, F., & Sideri, M. (2002). Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence, 138*(1-2), 55-86.

Gould, R. (1988). *Graph theory*. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Graham, S. M., Joshi, A., & Pizlo, Z. (2000). The traveling salesman problem: A hierarchical model. *Memory & Cognition, 28*(7), 1191-1204.

Green, D. M. & Swets, J. A. (1966). *Signal detection theory and psychophysics.* New York: John Wiley & Sons, Inc.

Gross, J. & Yellen, J. (1999). *Graph theory and its applications.* New York: CRC Press.

Haselager, W. F. G. (1997). *Cognitive science and folk psychology: The right frame of mind*. London: Sage.

Haselager, W. F. G., Bongers, R. M. & van Rooij, I. (forthcoming). Cognitive science, representations and dynamical systems theory. In W. Tschacher and J-P. Dauwalder (Eds.), *Dynamical Systems Approaches to Embodied Cognition.*

Haselager, W. F. G., de Groot, A. D., & van Rappard, J. F. H. (2003). Representationalism vs. anti-representationalism: A debate for the sake of appearance. *Philosophical Psychology, 16*(1), 5-23.

Haugeland, J. (1991). Representational genera. In W. Ramsey, S. P. Stich, and D. E. Rumelhart (Eds.), *Philosophy and connectionist theory* (pp.61-89). Hillsdale, NJ: Lawrence Erlbaum Associates.

Haynes, T. W., Hedetniemi, S. T., & Slater, P. J. (1998). *Fundamentals of domination in graphs*. New York: Marcel Dekker, Inc.

Herken, R. (Ed.) (1988). *The universal Turing machine: A half-century survey.* New York: Oxford University Press.

Hopcroft, J., Motwani, R., & Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation.* Reading, MA: Addison-Wesley.

Horgan, T. & Tienson, J. (1996). *Connectionism and the philosophy of psychology.* Cambridge, MA: MIT Press.

Horsten, L. & Roelants, H. (1995). The Church-Turing thesis and effective mundane procedures. *Minds and Machines, 5*, 1-8.

Humphreys, M. S., Wiles, J., & Dennis, S. (1994). Toward a theory of human memory: Data structures and access processes. *Behavioral and Brain Sciences, 17*, 655-692.

Inhelder, B. & Piaget, J. (1958). *The growth of logical thinking from childhood to adolescence.* New York: Basic Books.

Israel, D. (2002). Reflections on Gödel's and Gandy's reflections on Turing's thesis. *Minds and Machines, 12,* 181-201.

Jagota, A. (1997). Optimization by a Hopfield-style network. In D. S. Levine & W. R. Elsberry (Eds.), *Optimality in biological and artificial networks?* (pp. 203-226). Hillsdale, NJ: Lawrence Erlbaum Publishers.

Judd, J. S. (1990). *Neural network design and the complexity of learning.* Cambridge, MA: MIT Press.

Kadlec, H., & Townsend, J. T. (1992). Signal detection analyses of dimensional interactions. In F. G. Ashby (Ed.), *Multidimensional models of perception and cognition* (pp. 181-227). Hillsdale, NJ: Lawrence Erlbaum Associates.

Kadlec, H. & van Rooij, I. (2003). Beyond existence: Inferences about mental processes from reversed associations. *Cortex, 39*(1), 183-187.

Kannai, Y. & Peleg, B. (1984). A note on the extension of an order to the power set. *Journal of Economical Theory, 32*, 172-175.

Karp, R. M. (1972). Reducibility among combinatorial problems. In R. Miller and J. Thatcher (Eds.), *Complexity of computer computations* (pp.85-104). New York: Plenum Press.

Khot, S. & Raman, V. (2000). Parameterized complexity of finding subgraphs with hereditary properties. In D.Z. Du, P. Eades, V. Estivill-Castro, X. Lin, & A. Sharma (Eds.), *Computing and Combinatorics, LNCS 1858* (pp.137-147). New York: Springer-Verlag,.

Kleene, S. C. (1936). General recursive functions of natural numbers. *Mathematische Annelen, 112,* 727-742.

Kleene, S. C. (1988). Turing's analysis of computability, and major applications of it. In R. Herken (Ed.). *The universal Turing machine: A half-century survey* (pp. 17-54). New York: Oxford University Press.

Kolb, B. & Whishaw, I. Q. (1996). *Fundamentals of human neuropsychology.* New York: W.H. Freeman.

Krueger, L. E. & Tsav, C.-Y. (1990). Analyzing vision at the complexity level: Misplaced complexity? *Behavioral and Brain Sciences, 13*(3), 449-450.

Kube, P. R. (1990). Complexity is complicated. *Behavioral and Brain Science, 13*(3), 450-451.

Kube, P. R. (1991). Unbounded visual search is not *both* biologically plausible *and* NP-complete. *Behavioral and Brain Sciences, 14*(4), 768-773.

Levesque, H. J. (1988). Logic and the complexity of reasoning. *Journal of Philosophical Logic, 17,* 355-389.

Lewis, H. R. & Papadimitrou, C. H. (1998). *Elements of the theory of computation.* Upper Saddle River, NJ: Prentice-Hall.

Li, M. & Vitányi, P. (1997). *An introduction to Kolmogorov complexity and its applications.* New York: Springer-Verlag

Luce, R. D. & Raiffa, H. (1957). *Games and decisions: Introduction and critical survey*. New York: John Wiley & Sons, Inc.

Luce, R. D., Raiffa, H. (1990) Utility theory. In P. K. Moser (Ed.), *Rationality in action: Contemporary approaches* (pp. 19-40). New York: Cambridge University Press.

MacGregor, J. N. & Ormerod, T. C. (1996). Human performance of the traveling salesman problem. *Perception & Psychophysics, 58*(4), 527-539.

MacGregor, J. N., Ormerod, T. C., & Chronicle, E. P. (1999). Spatial and contextual factors in human performance on the traveling salesperson problem. *Perception, 28*, 1417-1427.

MacGregor, J. N., Ormerod, T. C., & Chronicle, E. P. (2000). A model of human performance on the traveling salesperson problem. *Memory & Cognition, 28*(7), 1183-1190.

Margolis, H. (1987). *Patterns, thinking, and cognition: A theory of judgment*. Chicago: The University of Chicago Press.

Marr, D. (1977). Artificial intelligence - A personal view. *Artificial Intelligence, 9,* 37-48.

Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information.* San Francisco: W.H. Freeman and Company.

Martignon, L. & Hoffrage, U. (1999). Why does one-reason decision making work? A case study in ecological rationality. In G. Gigerenzer, P. M. Todd, & the ABC Research Group (Eds.). *Simple heuristics that make us smart* (pp. 119-140). New York: Oxford University Press.

Martignon, L. & Hoffrage, U. (2002). Fast, frugal, and fit: Simple heuristics for paired comparison. *Theory and Decision, 52,* 29–71.

Martignon, L. & Schmitt, M. (1999). Simplicity and robustness of fast and frugal heuristics. *Minds and Machines, 9*, 565-593.

Massaro, D. W. & Cowan, N. (1993). Information processing models: Microscopes of the mind. *Annual Review of Psychology, 44*, 383-425.

McClamrock, R. (1991). Marr's three levels: A re-evaluation. *Minds and Machines, 1*(2), 185-196.

Milllgram, E. (2000). Coherence: The price of the ticket. *Journal of Philosophy, 97*(2), 82-93.

Nebel, B. (1996). Artificial intelligence: A computational perspective. In G. Brewka (Ed.), *Principles of knowledge representation* (pp.237-266). Stanford, CA: CSLI Publications

Newell, A. (1982). The knowledge level. *Artificial Intelligence, 18,* 87-127.

Newell, A. & Simon, H. A. (1988a). The theory of human problem solving. In A. M. Collins & E. E. Smith (Eds.), *Readings in cognitive science: A perspective from psychology and artificial intelligence* (pp. 33-51). San Mateo: Morgan Kaufmann, Inc.

Newell, A. & Simon, H. A. (1988b). GPS, a program that simulates human thought. In A. M. Collins & E. E. Smith (Eds.), *Readings in cognitive science: A perspective from psychology and artificial intelligence* (pp. 453-460). San Mateo: Morgan Kaufmann, Inc.

Niedermeier, R. (2002). Invitation to fixed-parameter algorithms. *Habilitationsschrift, University of Tübingen.*

Niedermeier, R. & Rossmanith, P. (1999). Upper bounds for Vertex cover further improved. In C. Meinel and S. Tison (Eds.), *16th Annual Symposium on Theoretical Aspects of Computer Science LNCS 1563* (pp. 561-570). Berlin: Springer-Verlag.

Niedermeier, R. & Rossmanith, P. (2000). A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters, 73*(3-4), 125-129.

Oaksford, M. & Chater, N. (1993). Reasoning theories and bounded rationality. In K. I. Manktelow & D.E. Over (Eds.), *Rationality: Psychological and philosophical perspectives* (pp. 31-60). London: Routledge.

Oaksford, M. & Chater, N. (1998). *Rationality in an uncertain world: Essays on the cognitive science of human reasoning.* Hove, UK: Psychology Press.

O'Laughlin, C., & Thagard, P. (2000). Autism and coherence: A computational model. *Mind and Language, 15*, 375-392.

Papadimitriou, C. H. & Steiglitz, K. (1988). *Combinatorial optimization: Algorithms and complexity.* New York: Dover Publications.

Parberry, I. (1994). *Circuit complexity and neural networks*. Cambridge, MA: MIT Press.

Parberry, I. (1997). Knowledge, understanding, and computational complexity. In D. S. Levine & W. R. Elsberry (Eds.), *Optimality in biological and artificial networks?* (pp. 125-144). Hillsdale, NJ: Lawrence Erlbaum Publishers.

Parkin, M. & Bade, R. (1997). *Microeconomics*. Don Mills, Ont: Addison-Wesley Publishers Limited.

Payne, J. W., Bettman, J. R., & Johnson, E. J. (1993). *The adaptive decision maker*. New York: Cambridge University Press.

Port, R. F. & van Gelder, T. (Eds.) (1995). *Mind as motion: Explorations in the dynamics of cognition.* Cambridge, MA: MIT Press.

Post, E. L. (1936). Finite combinatory processes-formulation I. *Journal Symbolic Logic, 1,* 103-105.

Prieto, E. & Sloper, C. (forthcoming). Either/Or: Using Vertex Cover Structure in designing FPT-algorithms -- the case of *k*-Internal Spanning Tree. *Proceedings of Workshop on Algorithms and Data Structures.*

Putnam, H. (1975). *Mind, language and reality.* Cambridge, MA: Cambridge University Press.

Putnam, H. (1988). *Representation and reality.* Cambridge, MA: MIT Press.

Putnam, H. (1994). *Words and life*. Cambridge, MA: Harvard University Press.

Pylyshyn, Z. W. (1984). *Computation and Cognition: Towards a Foundation for Cognitive Science*. Cambridge, MA: MIT Press.

Pylyshyn, Z. (1991). The role of Cognitive Architectures in the Theory of Cognition. In K. VanLehn (Ed.), *Architectures for Intelligence* (pp. 189-223). Hillsdale, NJ: Lawrence Erlbaum Associates.

Regenwetter, M., Marley, A. A. J., & Joe, H. (1998). Random utility threshold models of subset choice. *Australian Journal of Psychology, 50*(3), 175-185.

Ristad, E. S. (1993). *The language complexity game*. Cambridge, MA: MIT Press.

Ristad, E. S. (1995). Computational complexity of syntactic agreement and lexical ambiguity. *Journal of Mathematical Psychology, 39*(3), 275-284.

Ristad, E. S. & Berwick, R.C. (1989). Computational consequences of agreement and ambiguity in natural language. *Journal of Mathematical Psychology, 33*(4), 379-396.

Rosch, E. (1973). Natural categories. *Cognitive Psychology, 4,* 328-350.

Rumelhart, D. E., McClelland, J. L., & the PDP Research Group (1986). *Parallel distributed processing. Explorations in the microstructure of cognition. Volume 1: Foundations.* Cambridge, MA: MIT Press.

Schmitt, M. (2003). Personal communication.

Schoch, D. (2000). A fuzzy measure for explanatory coherence. *Synthese, 122*, 291-311.

Searle, J. R. (1980). Minds, Brains and Programs. *Behavioral and Brain Sciences, 3*, 417-424.

Siegel, R. M. (1990). Is it really that complex? After all, there are no green elephants. *Behavioral and Brain Science, 13*(3), 453.

Siegelmann, H. & Sontag, E. (1994). Analog computation via neural networks. *Theoretical Computer Science, 131*, 331-360.

Simon, H. A. (1957). *Models of man: Social and rational.* New York: John Wiley & Sons, Inc.

Simon, H. A. (1988). Rationality as process and as product of thought. In D. E. Bell, H. Raiffa, & A. Tversky (Eds.), *Decision making: Descriptive, normative, and prescriptive interactions* (pp. 58-77). Cambridge, UK: Cambridge University Press.

Simon, H. A. (1990). Invariants of human behavior. *Annual Review of Psychology, 41*(1), 1-19.

Stege, U. (2000). *Resolving conflicts from problems in computational biology*. Ph.D. thesis, No. 13364, ETH Zürich.

Stege, U. & van Rooij, I. (2003). On practical fixed-parameter-tractable algorithms for dominating set. *Manuscript under revision*.

Stege, U. & van Rooij, I. (2001). Profit sets: Graph problems in human decision making. *Unpublished manuscript.*

Stege, U., van Rooij, I., Hertel. A, & Hertel P. (2002). An $O(pn + 1.151^p)$ algorithm for p-Profit Cover and its practical implications for Vertex Cover. In P. Bose and P. Morin (Eds.), *13th International Symposium on Algorithms and Computation, LNCS 2518* (pp. 249-261). Berlin: Springer-Verlag.

Steinhart, E. (2002). Logically possible machines. *Minds and Machines, 12,* 259-280.

Stillings, N. A., Feinstein, M. H., Garfield, J. L., Rissland, E. L., Rosenbaum, D. A., Weisler, S. E., & Baker-Ward, L. (1987). *Cognitive science: An Introduction*. Cambridge, MA: MIT Press.

Thagard, P. (1989). Explanatory coherence. *Behavioral and Brain Sciences, 12*, 435-502.

Thagard, P. (1993). Computational tractability and conceptual coherence: Why do computer scientists believe that P≠NP? *Canadian Journal of Philosophy, 23*(3), 349-364.

Thagard, P. (2000). *Coherence in thought and action.* Cambridge, MA: MIT Press.

Thagard, P., Eliasmith, C., Rusnock, P., & Shelley, C. P. (2002). Knowledge and coherence. In R. Elio (Ed.), *Common sense, reasoning, and rationality*, pp. 104-131. New York: Oxford University Press.

Thagard, P. & Kunda, Z. (1998). Making sense of people: Coherence mechanisms. In S. J. Read & L. C. Miller (Eds.), *Connectionist models of social reasoning and social behavior* (pp. 3-26). Mahwah, NJ: Lawrence Erlbaum Associates, Inc.

Thagard, P. & Shelley, C. P. (1997). Abductive reasoning: Logic, visual thinking, and coherence. In M.-L. D. Chiara, K. Doets, D. Mundici, & J. van Benthem (Eds.), *Logic and scientific methods* (pp. 413-427). Dordrecht: Kluwer.

Thagard, P. & Verbeurgt, K. (1998). Coherence as constraint satisfaction. *Cognitive Science, 22*(1), 1-24.

Thelen, E. & Smith, L. B. (1994). *A dynamic systems approach to the development of cognition and action.* Cambridge, MA: MIT Press.

Todd, P. M. & Gigerenzer, G. (2000). Précis of simple heuristics that make us smart. *Behavioral and Brain Sciences, 23*, 727-780.

Tsotsos, J. K. (1988). A 'complexity level' analysis of immediate vision. *International Journal of Computer Vision, Marr Prize Special Issue, 2*(1), 303-320.

Tsotsos, J. K. (1989). The complexity of perceptual search tasks. In N. S. Sridharan (Ed.), *Proceedings International Joint Conference on Artificial Intelligence*, San Francisco: Morgan Kaufmann Publishers.

Tsotsos, J. K. (1990). Analyzing vision at the complexity level. *Behavioral and Brain Sciences, 13*(3), 423-469.

Tsotsos, J. K. (1991). Is complexity theory appropriate for analyzing biological systems. *Behavioral and Brain Sciences, 14*(4), 770-773.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society,42,* 230-265.

van Gelder, T. (1995). What might cognition be if not computation? *The Journal of Philosophy, 7*, 345-381.

van Gelder, T. (1998). The dynamical hypothesis in cognitive science. *Behavioral and Brain Sciences, 21*, 615-665.

van Gelder, T. J. (1999). Defending the dynamical hypothesis. In W. Tschacher & J.-P. Dauwalder (Eds.), *Dynamics, Synergetics, Autonomous Agents: Nonlinear Systems Approaches to Cognitive Psychology and Cognitive Science* (pp. 13-28). Singapore: World Scientific.

Van Orden, G. C. & Kloos, H. (2003). The module mistake. *Cortex, 39*(1), 164-166.

van Rooij, I., Bongers, R. M., & Haselager, W. F. G. (2000). The dynamics of simple prediction: Judging reachability. In L. R. Gleitman & A. K. Joshi (Eds.), *Proceedings of the Twenty Second Annual Conference of the Cognitive Science Society* (pp. 535-540). Mahwah, NJ: Lawrence Erlbaum Associates, Publishers.

van Rooij, I., Bongers, R. M., & Haselager, W. F. G. (2002). A non-representational approach to imagined action. *Cognitive Science, 26*(3), 345-375.

van Rooij, I., Schactman, A., Kadlec, H., & Stege, U. (2003). Children's Performance on the Euclidean Traveling Salesperson Problem. *Manuscript under revision.*

van Rooij, I., Stege, U., & Kadlec, H. (2003). Sources of complexity in subset choice. *Manuscript under review.*

van Rooij, I., Stege, U., & Schactman, A. (2003). Convex hull and tour crossings in the Euclidean Traveling Salesperson problem: Implications for human performance studies. *Memory & Cognition, 31*(2), 215-220.

Vickers, D., Butavicius, M., Lee, M., & Medvedev, A. (2001). Human performance on visually presented traveling salesman problems. *Psychological Research, 65*, 34-45.

Wareham, H. T. (1996). The role of parameterized computational complexity theory in cognitive modeling. *AAAI-96 Workshop Working Notes: Computational Cognitive Modeling: Source of the Power.* [On-line]. Available: http://web.cs.mun.ca/~harold/papers.html.

Wareham, H. T. (1998). *Systematic parameterized complexity analysis in computational phonology.* Ph.D. thesis. University of Victoria.

Wells, A. J. (1996). Situated action, symbol systems and universal computation. *Minds and Machines, 6*, 33-46.

Wells, A. J. (1998). Turing's analysis of computation and theories of cognitive architecture. *Cognitive Science, 22*(3), 269-294.

Wiedemann, U. (1999). *Michael Williams' concepts of systematical and relational coherence.* [On-line]. Available: http://www.pyrrhon.de/cohere/williams.htm.

Appendix A: Notation and Terminology for Graphs

Many problems considered in this research are (or can be formulated as) graph problems. This appendix presents a basic introduction to graphs and defines the graph theoretic notation and terminology used throughout the manuscript. For more information on graphs and graph theory see e.g. Foulds (1992), Gould (1988), and Gross & Yellen (1990).

A graph $G = (V, E)$ is a pair of sets, V and E. The set V is called the *vertex set* of G and its elements are called *vertices*. The set E is called the *edge set* and is a subset of the Cartesian product $V \times V$ (we also write $E \subseteq V^2$, where V^2 denotes the 2-fold product of V). The elements of E are called *edges.* A graph can be thought of as a network consisting of nodes (vertices) and lines connecting some pairs of nodes (edges). Figure A1 gives an illustration of a graph $G = (V, E)$. The nodes in the figure represent the vertices in V and the lines connecting nodes represent edges in E. In other words, the graph in Figure A1 has vertex set $V = \{a, b, c, \ldots, z\}$ and edge set $E \subseteq V \times V$, with $E = \{(a, b), (a, d), (a, e), (b, i), \ldots, (z, y)\}$.

The number of elements in a set S is called the *size* of S and is denoted by $|S|$. Note that, for the graph in Figure A1, $|V| = 26$ and $|E| = 31$. The size of the set V is often denoted by the letter $n = |V|$, and the size of the set E is often denoted by the letter $m = |E|$.

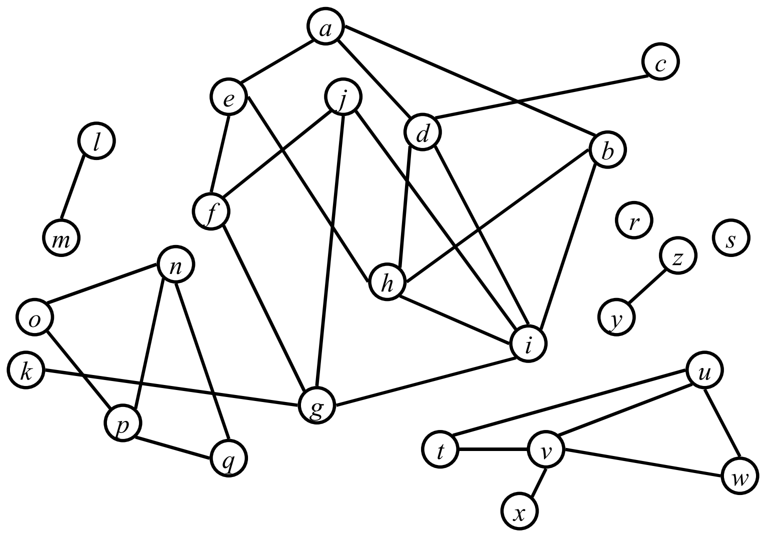


Figure A1. An illustration of a graph.

Note that in the graph in Figure A1 there are no connections from any vertex to itself (called a *self-loop*) and each pair of vertices is connected by at most one edge (i.e., there are no *multi-edges*). Graphs without self-loops and without multi-edges are called *simple* graphs. All graphs considered in this manuscript are simple unless otherwise noted. Further, note that there is no particular order associated with the vertices in an edge, i.e. $(u, v) = (v, u)$. Graphs of this type are called *undirected.* All graphs considered in this manuscript are undirected.

We define additional terminology for graphs. Let $G = (V, E)$ be a graph. We say a vertex $v_1 \in V$ is *incident* to edge (v_2, v_3) and, conversely, (v_2, v_3) is incident to v_1, if $(v_2, v_3) \in E$ and $v_1 = v_2$ or $v_1 = v_3$. If an edge $(v_1, v_2) \in E$ we also call v_1 and v_2 the *endpoints* of the edge (v_1, v_2) and we say that v_1 and v_2 are *neighbors*. For example, in Figure A1, edge (e, f) is incident to vertex e, and e is an endpoint of (e, f). Further, e and f are neighbors, while, for example, m and f are not.

The set of vertices that are neighbors of vertex v in graph G is called the (*open*) *neighborhood* of v, denoted by $\mathrm{N}_G(v) = \{u \in V : u \text{ is a neighbor of } v\}$. Note that v itself is not a member of $\mathrm{N}_G(v)$. We call $\mathrm{N}_G[v] = \mathrm{N}_G(v) \cup \{v\}$ the *closed neighborhood* of v. For example, in Figure A1, vertex d has neighbors a, c, i, and e. Thus we have $\mathrm{N}_G(d) = \{a, c, i, e\}$ and $\mathrm{N}_G[d] = \{a, c, i, e, d\}$. The number of edges incident to a vertex v is called the *degree* of v and is denoted by $\deg_G(v)$. For example, in Figure A1, vertex d had degree $\deg_G(d) = 4$. Note that, for any vertex v in a graph G, we have $\deg_G(v) = |\mathrm{N}_G(v)|$. If a vertex has $\deg_G(v) = k$ we also say that v is a *k-degree* vertex. We call degree-0 vertices *singletons*, and we call degree-1 vertices *pendant* vertices. For example, in Figure A1, vertices r and s are singletons, and vertices c, k, l, m, x, y, and z are all pendant vertices.

Let $G = (V, E)$ be a graph and let $V' \subseteq V$ be a subset of vertices. Then $\mathrm{N}_G(V') = \{v \in V : \text{there is a neighbor } u \text{ of } v \text{ with } u \in V'\}$ is called the (open) *neighborhood* of V' and $\mathrm{N}_G[V'] = \{v \in V : v \in V' \text{ or there is a neighbor } u \text{ of } v \text{ with } u \in V'\}$, is called the *closed neighborhood* of set V'. In other words, $\mathrm{N}_G[V'] = \mathrm{N}_G(V') \cup V'$. For example, in Figure A1, $\mathrm{N}_G(\{b, h, i\}) = \{a, e, d, j, g\}$ and $\mathrm{N}_G[\{b, h, i\}] = \{a, e, d, j, g, b, h, i\}$. Further, $\mathrm{E}_G(V') = \{(u, v) \in E : u \in V' \text{ and } v \in V')$ denotes the set of edges that have both endpoints in V', and $\mathrm{R}_G(V') = \{(u, v) \in E : u \in V' \text{ or } v \in V')$ denotes the set of edges that have at least one of their endpoints in V'. For example, in Figure A1, we have $\mathrm{E}_G(\{b, d, h, i, l, m\}) = \{(b, h), (d, i), (h, i), (d, h), (l, m)\}$ and $\mathrm{R}_G(\{e, f, l\}) = \{(e, f), (e, h), (e, a), (f, j), (f, g), (l, m)\}$.

The *set difference* of two sets S_1 and S_2 is denoted by $S_1 \backslash S_2 = \{v \in S_1 : v \notin S_2\}$. Whenever we delete a vertex from a graph we also have to remove its incident edges (otherwise the remaining vertex and edge sets would not form a graph anymore). Hence, the set $\mathrm{R}_G(V')$ is the set of edges that gets removed from G when we delete the vertices in V' from V. That is, if we remove V' from V the resulting graph $G^* = (V^*, E^*)$ will have vertex set $V^* = V \backslash V'$ and edge set $E^* = E \backslash \mathrm{R}_G(V')$.

Let $G = (V, E)$ be a graph. Then a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ is called a *subgraph* of G. We also call G a *supergraph* of G'. Let $V' \subseteq V$ be a subset of vertices in G. Then we call the subgraph $G' = (V', E')$ with $V' = V$ and $E' = \mathrm{E}_G(V')$ the subgraph of G *induced by* V'. For example, for the graph $G = (V, E)$ in Figure A1, $G' = (V', E')$, with $V' = \{a, b, d, h, i\}$ and $E' = \{(a, b), (b, h), (a, d), (h, i)\}$, is a subgraph of G, and $G'' = (V'', E'')$ with $V'' = \{a, b, d, h, i\}$ and $E'' = \mathrm{E}_G(\{a, b, d, h, i\}) = \{(a, b), (b, h), (a, d), (d, i), (h, i), (d, h)\}$ is the subgraph of G induced by $\{a, b, d, h, i\}$.

A sequence $< v_1, v_2, ..., v_k >$ of pairwise distinct vertices with $(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k) \in E$ is called a *path* in G. If $v_1 = v_k$ and $k \geq 3$ then $< v_1, v_2, ..., v_k >$ is called a *cycle* in G. For example, in Figure A1, $<k, g, i, b, h>$ is a path, while $<g, i, b, h, i>$ is not. Further, in the graph in Figure A1, $<a, b, i, h, e, a>$ is a cycle. We may also denote a path and a cycle by its edges instead of its vertices, as follows: $<(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k)>$. For example, we may denote the path $<k, g, i, b, h>$ in Figure A1 by $<(k, g), (g, i), (i, b), (b, h)>$ instead. The *length* of a path is the number of edges visited when traversing the path. Thus, path $<k, g, i, b, h>$ in Figure A1 has length 4.

A graph is *connected* if for every pair of vertices $u, v \in V$ there is a path in G from u to v. Note that the graph in Figure A1 is not connected (e.g., there is no path from t to g). However the graph in Figure A2 is connected.

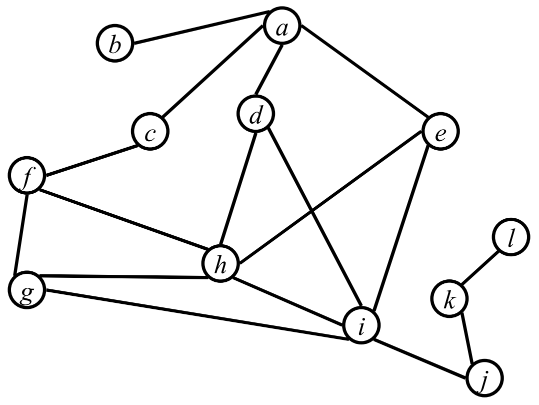


Figure A2. An illustration of a connected graph.

Let *G'* = (*V'*, *E'*) be a subgraph of *G*. We say *G'* is a *component* of *G* if (1) *G'* is connected, and (2) there does not exist a subgraph *G** of *G*, *G** ≠ *G'*, such that *G** is connected and *G** is a supergaph of *G'*. For example, the graph in Figure A1 consist of 7 components: viz., the components induced by vertex sets {*l*, *m*}, {*o*, *n*, *p*, *q*}, {*e*, *a*, *f*, *j*, *d*, *c*, *b*, *h*, *i*}, {*t*, *u*, *v*, *x*, *w*}, {*y*, *z*}, {*r*}, and {*s*}. Note that, for example, the subgraph of *G* induced by {*t*, *u*, *v*, *x*} is *not* a component of *G*. To see why this is so, consider the subgraph *G'* = (*V'*, *E'*) of *G* induced by {*t*, *u*, *v*, *x*} and the subgraph *G** = (*V**, *E**) of *G* induced by {*t*, *u*, *v*, *x*, *w*}. Note that *G** is connected and *G** is a supergraph of *G'*. We conclude that *G'* is not a component of *G*. Also note that since the graph in Figure A2 is connected it has only one component.

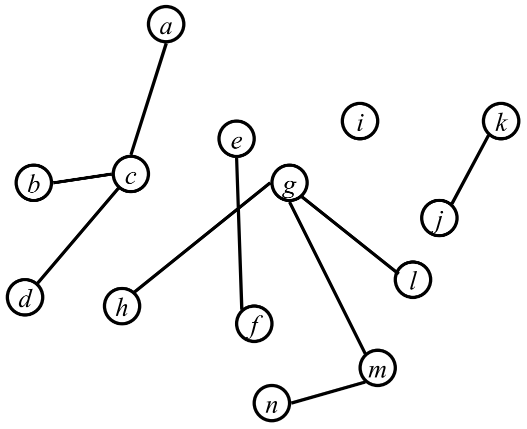


Figure A3. An illustration of a forest.

A graph without any cycles is called a *forest* and a connected forest is called a *tree*. Figure A3 shows a graph that is a forest and Figure A4 shows a graph that is a tree. A *rooted tree* is a tree with a designated vertex called the *root*. Let $T = (V_T, E_T)$ be a rooted tree, with root $r \in V_T$. A pendant vertex in a tree is called a *leaf*. For two vertices $u, v \in V_T$, with $(u,v) \in E_T$, we say u is *parent* of v, and v is *child* of u, if $< r, ..., u, v>$ is a path in T. The *depth* of T is the length of the longest path from root r to a leaf in T. For example, for the tree $T = (V_T, E_T)$ in Figure A4 we may call vertex a the root of T. Then vertices b, d, h, f, i, k are leaves in T. For example, e is a child of a, and e is parent of g. Further, T in Figure A4 has depth 4.

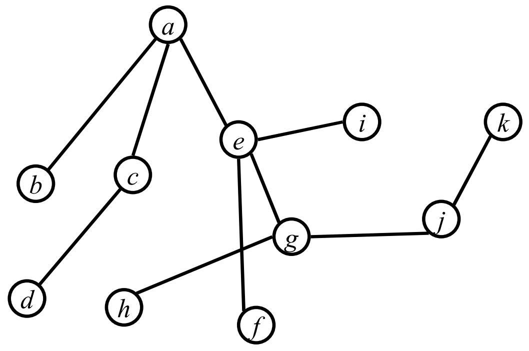


Figure A4. An illustration of a tree.

In the text also generalizations of the graph concept are used. Specifically, we discuss weighted graphs, multigraphs, and hypergraphs. A *weighted* graph $G = (V, E)$ is a graph in which each vertex $v \in V$ has an associated *vertex-weight* $w_V(v)$ and each edge $e \in E$ has an associated *edge-weight* $w_E(e)$. A *multigraph* $M = (V, E)$ is a generalization of a graph in which more than one edge may connect any two vertices in G (i.e., the edge set E is a multiset). A *hypergraph* $H = (V, E)$ is a generalization of a graph in which a *hyperegde* $(v_1, v_2, \ldots, v_h) \in E$, $h \geq 2$, can be incident to more than two vertices. Specific properties of these types of graphs will be detailed in the text where they are first used.

Appendix B: A Compendium of Problems

This appendix presents an overview of decision problems that appear in the text. Problems are listed in alphabetical order. For each problem a page number is given, indicating where the problem first appears in the text. Where possible also a literature reference is given. Each problem entry lists basic complexity results and/or other relevant information. For some problems, special cases appear as a part of the entry of the general problem.

Annotated Coherence [page 107]
Input: A network $N = (P, C)$, with $(P' \cup A' \cup R')^P$ and $(C^+ \cup C^-)^C$. For each $(p, q) \in C$ there is an associated positive integer weight $w(p, q)$. A positive integer c.
Question: Does there exist a partition of P into A and R such that $A' \subseteq A$, $R' \subseteq R$, and $\text{Coh}_N(A, R) \geq c$?

We have shown Annotated Coherence to be in FPT for parameter $|P^-|$, denoting the number of elements in P that are incident to a negative edge, (Theorem 5.3, page 114) and for parameter $|C^-|$ (denoting the number of negative constraints in the network; Corollary 5.12, page 114). (see Coherence; Foundational Coherence)

Annotated Min-Incomp-Lex [page 163]
Input: A set of objects $A = \{a_1, a_2, \ldots, a_m\}$ and a set of features $F = \{f_1, f_2, \ldots, f_n\}$. Here F partitions into $F_1 = \{f_1, f_2, \ldots, f_{n_1}\}$ and $F_2 = \{f_{n_1+1}, f_{n_1+2}, \ldots, f_n\}$. Each $a \in A$ has an associated value $b_i(a) \in \{0, 1\}$, $i = 1, 2, \ldots, n$, denoting the value that a takes on feature $f_i \in F$. A complete ordering $S(A)$, with $a_1 > a_2 > \ldots > a_m$, and a permutation of the features in F_1, $\pi(F_1)$. An integer $k \geq 0$.
Question: Does there exist a permutation of the features in F_2, $\pi(F_2)$, such that $\text{Error}_{\text{LEX}}(A, \text{S}(A), F, P, \pi(F_2)) \leq k$? Here $P \subseteq A \times A$ denotes the set of pairs of distinct objects in A that are not distinguished by any feature in F_1.

Annotated Min-Incomp-Lex is a generalization of the problem Min-Incomp-Lex. We have shown that Annotated Min-Incomp-Lex is solvable in time $O\left(\frac{n!}{(n-k)!}n^2m^2\right)$, for $|A| = m$ and $|F| = n$ (page 166).

Bottom-up Visual Matching [page 170] (e.g. Tsotsos, 1990)
Input: An image I and a target T. Each pixel $p_i = (x, y, b_i)$, $p_i \in I$, with $p_t = (x, y, b_t)$, $p_t \in T$, has an associated value diff(p_i) = $|b_i - b_t|$ and an associated value corr(p_i) = b_ib_t. Two positive integers θ and ϕ.
Question: Does there exist a subset of pixels $I' \subseteq I$ such that $\sum_{p\in I'} \text{diff}(p) \le \theta$ and $\sum_{p\in I'} \text{corr}(p) \ge \phi$?

Bottom-up Visual Matching is known to be NP-hard by reduction from Knapsack (Tsotsos, 1989). The problem is solvable in pseudo-polynomial time $O(\theta|I|)$ (Kube, 1991) and in pseudo-polynomial time $O(\lambda|I|^2)$ (page 173) (see also Knapsack, Top-down Visual Matching).

Clique [page 151] (e.g. Garey & Johnson, 1979)
Input: A graph $G = (V, E)$ and a positive integer k.
Question: Does there exist a clique $V' \subseteq V$ for G with $|V'| \ge k$? (Here a vertex set V' is called a clique if for every two vertices $u, v \in V'$, $(u, v) \in E$.)

Clique is known to be NP-complete (Garey & Johnson, 1979), and W[1]-complete for parameter k (Downey & Fellows, 1999). A graph $G = (V, E)$ has a clique of size k if and only if its complement $G' = (V', E')$, $V' = V$ and $E' = V^2\backslash E$, has an independent set of size k (see Independent Set).

Coherence [page 80] (e.g. Thagard & Verbeurgt, 1998)
Input: A network $N = (P, C)$, with $(C^+ \cup C^-)^C$. For each $(p, q) \in C$ there is an associated positive integer weight w(p, q). A positive integer c.
Question: Does there exist a partition of P into A and R such that $\text{Coh}_N(A, R) \ge c$? Here $\text{Coh}_N(A, R) = \sum_{(p,q)\in S_N(A,R)} w(p,q)$, with $S_N(A, R) = \{(p, q) \in C^+ : (p \in A$ and $q \in$

A) or ($p \in R$ and $q \in R$)} $\cup$ {$(p, q) \in C^-$: ($p \in A$ and $q \in R$) or ($p \in R$ and $q \in A$)}.

Coherence is known to be NP-complete, by reduction from Max-Cut (Thagard & Verbeurgt, 1998; see also Lemma 5.1, page 84). It follows that Coherence is NP-hard even for inputs without positive constraints (see also Corollary 5.2, page 84). Further, Coherence remains NP-complete if all constraints are negative and all weights are '1' (Corollary 5.3, page 84). (see also Annotated Coherence, Discriminating Coherence, Foundational Coherence)

Conflict Graph (CG) Subset Choice [page 136] (e.g. van Rooij et al., 2003)

Input: An conflict graph $G = (V, E)$. For $v \in V$, $w_V(v) \in \mathbb{Z}$ and for $e \in E$, $w_E(e) \in \mathbb{Z}^-$. A positive integer p.

Question: Does there exist a subset $V' \subseteq V$ such that $\text{value}_G(V') \geq p$?

We have shown that CG Subset Choice is in FPT for parameter set $\{q, \Omega_V\}$, where $q = p - \text{value}_G(V)$ and Ω_V denotes the maximum vertex weight (Theorem 6.5, page 139). (see also VCG Subset Choice, Subset Choice, Subset Rejection).

Conflict Hypergraph (CH) Subset Choice [page 136] (van Rooij et al., 2003)

Input: A conflict hypergraph $H = (V, E)$. For every $v \in V$, $w_V(v) \in \mathbb{Z}$ and for every $e \in E$, $w_E(e) \in \mathbb{Z}^-$. A positive integer p.

Question: Does there exist a subset $V' \subseteq V$ such that $\text{value}_H(V') \geq p$?

We have shown that CH Subset Choice is in FPT for parameter set $\{q, \varepsilon, \Omega_V\}$, where $q = p - \text{value}_G(V)$, ε denotes the maximum span, and Ω_V denotes the maximum vertex weight (Theorem 6.6, page 144). (see also Subset Choice, Subset Rejection).

Discriminating Coherence [page 90] (e.g. Thagard, 2000)

Input: A network $N = (P, C)$, with $C \subseteq P \times P$. Here $P = H \cup D$, and $C = C^+ \cup C^-$ (with H, D being disjoint, and C^+, C^- being disjoint). For each $d \in D$ there is an associated positive integer weight $w_D(d)$, and for each $(p, q) \in C$ there is an associated positive integer weight $w_C(p, q)$. A positive integer c.

Question: Does there exist a partition of P into A and R such that DCoh$_N(A, R) \geq c$? Here DCoh$_N(A, R) = \sum_{(p,q)\in S_N(A,R)} w_C(p,q) + \sum_{d\in D, d\in A} w_D(d)$.

Discriminating Coherence is a generalization of Coherence. (see Coherence).

Dominating Set [page 35] (e.g. Garey & Johnson, 1979)

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Does there exist a dominating set V' for G with $|V'| \leq k$?

Dominating Set is known to be NP-complete (Garey & Johnson, 1979), and W[1]-hard for parameter k (Downey & Fellows, 1999). A graph $G = (V, E)$ has a dominating set of size k if and only if G has a non-blocking set of size $|V| - k$ (see Non-Blocker).

Double-Constraint Coherence [page 94]

Input: A double-constraint network $N = (P, C^+ \cup C^-)$, with $C^+ \subseteq P \times P$ and $C^- \subseteq P \times P$. For each $(p, q)^+ \in C^+$ there is an associated positive integer weight $w(p, q)^+$, and for each $(p, q)^- \in C^-$ there is an associated positive integer weight $w(p, q)^-$. A positive integer c.

Question: Does there exist a partition of E into $A \cup R$ such that Coh$_N(A, R) \geq c$?

Double-Constraint Coherence is a generalization of Coherence. We have shown that Double-Constraint Coherence is in FPT for parameter c (Theorem 5.2, page 103). (see also Coherence).

Edge-weighted Conflict Graph (ECG) Subset Choice [page 136] (e.g. van Rooij et al., 2003)

Input: An edge-weighted conflict graph $G = (V, E)$. For every $v \in V$, $w_V(v) \in \mathbb{Z}$ and for every $e \in E$, $w_E(e) \in \mathbb{Z}^-$. A positive integer p.

Question: Does there exist a subset $V' \subseteq V$ such that value$_G(V') \geq p$?

We have shown that ECG Subset Choice is in FPT for relational parameter $q = p -$ value$_G(V)$ (Corollary 6.7, page 137). (see also Subset Choice; Subset Rejection).

Exact-bound Subset Choice [page 151] (e.g. van Rooij et al., 2003)

Input: A weighted hypergraph $H = (V, E)$. For every $v \in V$, $w_V(v) \in \mathbb{Z}$ and for every $e \in E$, $w_E(e) \in \mathbb{Z}\backslash\{0\}$. Positive integers p and k.

Question: Does there exist a subset $V' \subseteq V$ such that value$_H(V') \geq p$ and $|V'| = k$?

We have shown that Exact-bound Subset Choice is NP-complete (Theorem 6.8, page 151), and W[1]-hard for parameter set $\{p, k\}$ (Corollary 6.15, page 153), even for unit-weighted surplus graphs. (see also Upper-bound Subset Choice, Lower-bound Subset Choice).

Foundational Coherence [page 90] (e.g. Thagard, 2000)

Input: A network $N = (P, C)$, with $C \subseteq P \times P$. Here $P = H \cup D$, and $C = C^+ \cup C^-$ (with H, D being disjoint, and C^+, C^- being disjoint). For each $(p, q) \in C$ there is an associated positive integer weight w(p, q). A positive integer c.

Question: Does there exist a partition of P into A and R such that $D \subseteq A$ and Coh$_N(A, R) \geq c$?

Foundational Coherence is a generalization of Coherence, and a special case of Annotated Coherence. (see Annotated Coherence; Coherence; Discriminating Coherence).

Halting problem [page 17] (e.g. Turing, 1936).

Input: A Turing machine M and an input i for M.

Question: Does M halt on i?

The Halting problem is known to be undecidable (Turing, 1936).

Inclusive Profit Domination [page 68] (e.g. Stege & van Rooij, 2001)

Input: A graph $G = (V, E)$ and a positive integer p.

Question: Does there exist a vertex set $V' \subseteq V$ such that profit$_{\text{IPD},G}(V') \geq p$? Here profit$_{\text{IPD},G}(V') = |\text{N}_G[V']| - |V'| = |\text{N}_G(V')|$.

Inclusive Profit Domination is known to be NP-complete, and W[1]-hard for parameter p (Stege & van Rooij, 2001; see also Stege & van Rooij, 2003).

Independent Set [page 66] (e.g. Garey & Johnson, 1979)

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Does there exist an independent set V' for G with $|V'| \geq k$?

Independent Set is known to be NP-complete (Garey & Johnson, 1979), and W[1]-complete for parameter k (Downey & Fellows, 1999). A graph $G = (V, E)$ has an independent of size k if and only if $G = (V, E)$ has a vertex cover of size $|V| - k$ (see Vertex Cover).

Knapsack [page 171] (e.g. Garey & Johnson, 1979).

Input: A finite set U. Each $u \in U$ has a size $s(u) \in Z^+$ and a value $v(u) \in Z^+$. Positive integers B and K.

Question: Does there exist a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$?

Knapsack is known to be NP-complete, and solvable in pseudo-polynomial time (e.g. Garey & Johnson, 1979).

Lower-bound Subset Choice [page 192]

Input: A weighted hypergraph $H = (V, E)$. For every $v \in V$, $w_V(v) \in \mathbb{Z}$ and for every $e \in E$, $w_E(e) \in \mathbb{Z} \backslash \{0\}$. Positive integers p and k.

Question: Does there exist a subset $V' \subseteq V$ such that $\text{value}_H(V') \geq p$ and $|V'| \geq k$?

Lower-bound Subset Choice has not been studied in this work, but it appears in the list of open problems on page 192 (see also Exact-bound Subset Choice, Upper-bound Subset Choice).

Max-Cut [page 83] (e.g. Garey & Johnson, 1979)

Input: An edge weighted graph $G = (V, E)$. For each edge $(u, v) \in E$ there is an associated positive integer weight $w(u, v)$. A positive integer k.

Question: Does there exist a partition of V into sets A and R such that $W_G(A, R) = \sum_{(u,v) \in \text{Cut}_G(A,R)} w(u, v) \geq k$? Here $\text{Cut}_G(R, A) = \{(u, v) \in E : u \in A \text{ and } v \in R\}$.

Max-Cut is known to NP-complete (Garey & Johnson, 1979). The problem remains NP-complete even if all edges have weight ‘1’ (called the Simple Max-Cut problem; Garey & Johnson, 1979).

Maximum Matching [page 55] (e.g. Gross & Yellen, 1999)

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Does there exist a matching $E' \subseteq E$ for G with $|V'| \geq k$? (Here E' is called a matching if for every two edges $(u, v), (x, y) \in E'$, if vertices u, v, x, and y are all distinct.

Maximum Matching is known to be solvable in $O(|V|^2)$ (e.g. Gross & Yellen, 1999).

Min-Cut [page 110] (e.g. Cormen, et al. 1990)

Input: An edge weighted graph $G = (V, E)$. A source $s \in V$ and a sink $t \in V$. For each edge $(u,v) \in E$ there is an associated positive integer weight $w(u, v)$. A positive integer k.

Question: Does there exist a partition of V into disjoint sets A and R such that, $s \in A$, $t \in R$, and $W_G(A, R) = \sum_{(u,v)\in \mathrm{Cut}_G(A,R)} w(u,v) \leq k$? Here $\mathrm{Cut}_G(R, A) = \{(u, v) \in E : u \in A \text{ and } v \in R\}$.

Min-Cut is known to be solvable in time $O(|V|^3)$ (e.g. Cormen et al., 1990).

Min-Incomp-Lex [page 155] (e.g. Martignon & Schmitt, 1999)

Input: A set of objects $A = \{a_1, a_2, \ldots, a_m\}$ and a set of features $F = \{f_1, f_2, \ldots, f_n\}$. Each $a \in A$ has an associated value $b_i(a) \in \{0, 1\}$, $i = 1, 2, \ldots, n$, denoting the value that a takes on feature $f_i \in F$. A complete ordering $S(A)$, with $a_1 > a_2 > \ldots > a_m$. An integer $k \geq 0$.

Question: Does there exist a permutation $\pi(F) = \langle f_{\pi 1}, f_{\pi 2}, \ldots, f_{\pi_n} \rangle$, such that $\mathrm{Error}_{\mathrm{LEX}}(A, S(A), F, P, \pi(F)) \leq k$? Here $P \subseteq A \times A$ denotes the set of all pairs of distinct objects in A.

Min-Incomp-Lex is known to be NP-hard by reduction from Vertex Cover (e.g. Martignon & Schmitt, 1999; see also Theorem 7.1, page 161). We have shown that Min-Incomp-Lex is in FPT for parameter $m = |A|$ (page 163). (see also Annotated Min-Incomp-Lex)

Non-Blocker [page 66] (e.g. Stege & van Rooij, 2003).

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Does there exist a non-blocking set V' for G with $|V'| \geq k$?

Non-Blocker is known to be NP-complete and in FPT for parameter k (unpublished result by Fellows & McCartin, 1999). A graph $G = (V, E)$ has a non-blocking set of size k if and only if $G = (V, E)$ has a dominating set of size $|V| - k$ (see Dominating Set). A $G = (V, E)$ has non-blocking set of size $k = p$ if and only if there exists a subset $V' \subseteq V$ with $\text{profit}_{\text{IPD},G}(V') = p = k$ (see Inclusive Profit Domination).

Pos-Annotated Coherence [page 108]
Input: A network $N = (P, C)$, with $(P' \cup A' \cup R')^P$ and $(C^+ \cup C^-)^C$. For every $(p, q) \in C^-$, $p, q \in A' \cup R'$. For each $(p, q) \in C$ there is an associated positive integer weight $w(p, q)$. A positive integer c.
Question: Does there exist a partition of P' into A and R such that $A' \subseteq A$, $R' \subseteq R$, and $\text{Coh}_N(A, R) \geq c$?

We have shown that Pos-Annotated Coherence is solvable in time $O(|P|^3)$ by reduction to Min-Cut (Lemma 5.11, page 113). (see also Min-Cut)

Profit Cover [page 67] (e.g. Stege et al., 2002)
Input: A graph $G = (V, E)$ and a positive integer p.
Question: Does there exist a vertex set $V' \subseteq V$ such that $\text{profit}_{\text{PC},G}(V') \geq p$? Here $\text{profit}_{\text{PC},G}(V') = |\text{R}_G(V')| - |V'|$.

Profit Cover is known to be NP-complete, and in FPT for parameter p (Stege, et al., 2002). For a graph $G = (V, E)$, there exists a subset $V' \subseteq V$ with $\text{profit}_{\text{PC},G}(V') = p$ if and only if $G = (V, E)$ has a vertex cover of size $k = |E| - p$ (see Vertex Cover). Further, Profit Cover is equivalent UCG Subset Rejection (see UCG Subset Choice; Subset Rejection).

Profit Independence [page 68] (e.g. Stege & van Rooij, 2001)
Input: A graph $G = (V, E)$ and a positive integer p.
Question: Does there exist a vertex set $V' \subseteq V$ such that $\text{profit}_{\text{PI},G}(V') \geq p$? Here $\text{profit}_{\text{PI},G}(V') = |V'| - |\text{E}_G(V')|$.

Profit Independence is known to be NP-complete, and W[1]-hard for parameter p (Stege & van Rooij, 2001; see also Lemma 4.3, page 77). For a graph $G = (V, E)$, there exists a subset $V' \subseteq V$ with $\text{profit}_{\text{PI},G}(V') = p$ if and only if $G = (V, E)$ has an independent set of size $k = p$ (see Independent Set). Further, Profit Independence is equivalent to UCG Subset Choice (see UCG Subset Choice).

Satisfiability (SAT) [page 35] (e.g. Garey & Johnson, 1979).

Input: A set of variables U and a collection of clauses C over U. Each variable $u \in U$ has two associated literal u and $\overline{u}$ (the latter representing the negation of u). Each clause $(u_1, u_2, \ldots, u_l)$ in C contains represents the disjunction of the literals $u_1, u_2, \ldots, u_l$. Each variable in U can be assigned the truth-value T or F, and a clause is satisfied if the disjunction of the literals in the clause yields the truth-value T.

Question: Does there exists a truth assignment to variables in U such that all clauses in C are satisfied?

Satisfiability is known to be NP-complete (Cook, 1971). The special case, 3-SAT, with at most three literals per clause, remains NP-complete; the special case, 2-SAT, with at most two literals per clause, is known to be in P (e.g. Garey & Johnson, 1979).

Single-Element Discriminating Coherence [page 92]

Input: A network $N = (P, C)$, with $C \subseteq P \times P$. Here $P = H \cup D$, and $C = C^{+} \cup C^{-}$ (with H, D being disjoint, and C^{+}, C^{-} being disjoint). For each $d \in D$ there is an associated positive integer weight $w_D(d)$, and for each $(p, q) \in C$ there is an associated positive integer weight $w_C(p, q)$. A special element $s \in H$ and a positive integer c.

Question: Does there exist a partition of P into A and R such that $\mathrm{DCoh}_N(A, R)$ is maximum and $s \in A$?

We have shown that if $D = \varnothing$ then Single-Element Discriminating Coherence is in P (Corollary 5.8, page 93). The classical complexity of the general problem remains unknown (see also Single-Element Foundational Coherence).

Single-Element Foundational Coherence [page 92]

Input: A network $N = (P, C)$, with $C \subseteq P \times P$. Here $P = H \cup D$, and $C = C^{+} \cup C^{-}$ (with H, D being disjoint, and C^{+}, C^{-} being disjoint). For each $(p, q) \in C$ there is an associated positive integer weight w(p, q). A special element $s \in H$ and a positive integer c.

Question: Does there exist a partition of P into A and R such that $\mathrm{Coh}_N(A, R)$ is maximum and such that $D \cup s \subseteq A$?

We have shown that Single-Element Foundational Coherence with $D = \varnothing$ is in P (Corollary 5.9, page 93). The classical complexity of the general problem remains unknown (see also Single-Element Discriminating Coherence).

Subset Choice [page 117] (e.g. van Rooij et al., 2003)

Input: A *weighted* hypergraph $H = (V, E)$, $E \subseteq \bigcup_{2 \leq h \leq |V|} V^h$. For every $v \in V$ there is a weight $\mathrm{w_V}(v) \in \mathbb{Z}$ and for every $e \in E$ there is a weight $\mathrm{w_E}(e) \in \mathbb{Z} \backslash \{0\}$. A positive integer p.

Question: Does there exist a subset $V' \subseteq V$ such that $\mathrm{value}_H(V') \geq p$? Here $\mathrm{value}_H(V') = \sum_{v \in V'} \mathrm{w_V}(v) + \sum_{e \in \mathrm{E}_H(V')} \mathrm{w_E}(e)$, with $\mathrm{E}_H(V') = \{(v_1, v_2, ..., v_h) \in E \mid v_1, v_2, ..., v_h \in V'\}$.

We have shown that Subset Choice is NP-hard by several independent reductions from Independent Set (Lemma 6.1 on page 123, Lemma 6.2 on page 125, and Lemma 6.5 on page 138; see also Fishburn & LaValle, 1996), as well as by reduction from Coherence (Lemma 6.3, page 126). Further, Subset Choice is not to be in FPT (unless FPT = W[1]) for parameter set $\{p, \varepsilon, \omega_\mathrm{V}, \Omega_\mathrm{V}, \omega_\mathrm{E}, \Omega_\mathrm{E}\}$ (Corollary 6.13, page 148) and for parameter set $\{q, \varepsilon, \omega_\mathrm{V}, \omega_\mathrm{E}, \Omega_\mathrm{E}\}$ (Corollary 6.14, page 148). For more results on Subset Choice refer to Chapter 6 and van Rooij et al. (2003). (see also UCG Subset Choice; ECG Subset Choice; VCG Subset Choice; CG Subset Choice; CH Subset Choice; SH Subset Choice; Exact-Bound Subset Choice; Subset Rejection)

Subset Rejection [page 129] (e.g. van Rooij et al., 2003)

Input: A *weighted* hypergraph $H = (V, E)$, $E \subseteq \bigcup_{2 \leq h \leq |V|} V^h$, for every $v \in V$ a weight $\mathrm{w_V}(v) \in \mathbb{Z}$, for every $e \in E$ a weight $\mathrm{w_E}(e) \in \mathbb{Z} \backslash \{0\}$, and a positive integer q.

Question: Does there exist a subset $V' \subseteq V$ such that reject$_H(V') \geq q$? Here reject$_H(V')$ = value$_H(V/V')$ – value$_H(V)$ = $-\sum_{v \in R} \mathrm{w}_{\mathrm{V}}(v) - \sum_{e \in \mathrm{R}_H(V')} \mathrm{w}_{\mathrm{E}}(e)$, with $\mathrm{R}_H(V')$ = $\{(v_1, v_2, ..., v_h) \in E \mid v_1$ or v_2 or ... or $v_h \in V'\}$.

The problem Subset Rejection is equivalent to Subset Choice with p = value$_H(V') + q$. (see Subset Choice)

Top-down Visual Matching [page 169] (e.g. Tsotsos, 1990)

Input: An image I and a target T. Each pixel $p_i = (x, y, b_i)$, $p_i \in I$, with $p_t = (x, y, b_t)$, $p_t \in T$, has an associated value diff$(p_i) = |b_i - b_t|$ and an associated value corr$(p_i) = b_i b_t$. Two positive integers θ and ϕ.

Question: Is $\sum_{p \in I} \mathrm{diff}(p) \leq \theta$ and $\sum_{p \in I} \mathrm{corr}(p) \geq \phi$?

Top-down visual matching is solvable in time $O(|I|) = O(|T|)$ (e.g. Tsotsos, 1990; see also page 169).

Unit-weighted Conflict Graph (UCG) Subset Choice [page 124] (e.g. van Rooij et al., 2003)

Input: A unit-weighted conflict graph $G = (V, E)$. For every $v \in V$, $\mathrm{w}_{\mathrm{V}}(v) = 1$ and for every $e \in E$, $\mathrm{w}_{\mathrm{E}}(e) = -1$. A positive integer p.

Question: Does there exist a subset $V' \subseteq V$ such that value$_G(V') \geq p$?

We have shown that UCG Subset Choice is in FPT for relational parameter $q = p -$ value$_G(V)$ (Theorem 6.3, page 130). (see also Subset Choice; Subset Rejection)

Upper-bound Subset Choice [page 192]

Input: A weighted hypergraph $H = (V, E)$. For every $v \in V$, $\mathrm{w}_{\mathrm{V}}(v) \in \mathbb{Z}$ and for every $e \in E$, $\mathrm{w}_{\mathrm{E}}(e) \in \mathbb{Z} \backslash \{0\}$. Positive integers p and k.

Question: Does there exist a subset $V' \subseteq V$ such that value$_H(V') \geq p$ and $|V'| \leq k$?

Upper-bound Subset Choice has not been studied in this work, but it appears in the list of open problems in Appendix D on page 192 (see also Exact-bound Subset Choice, Lower-bound Subset Choice).

Vertex Cover [page 11] (e.g. Garey & Johnson, 1979)

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Does there exist a vertex cover V' for G with $|V'| \leq k$?

Vertex Cover is known to be NP-complete (Garey & Johnson, 1979), and in FPT for parameter k (Downey & Fellows, 1999). Vertex Cover remains NP-complete on graphs of maximum degree 4 (Garey & Johnson, 1979). A graph $G = (V, E)$ has a vertex cover of size k if and only if $G = (V, E)$ has an independent set of size $|V| - k$ (see Independent Set).

Vertex-weighted Conflict Graph (VCG) Subset Choice [page 136] (e.g. van Rooij et al., 2003)

Input: A vertex-weighted conflict graph $G = (V, E)$. For every $v \in V$, $w_V(v) \in \mathbb{Z}$ and for every $e \in E$, $w_E(e) = -1$. A positive integer p.

Question: Does there exist a subset $V' \subseteq V$ such that $\text{value}_G(V') \geq p$?

We have shown that VCG Subset Choice is W[1]-hard for parameter Ω_V (Lemma 6.5, page 138). (see also CG Subset Choice, Subset Choice, Subset Rejection).