Part I

Introduction

1 Introduction

In this chapter we introduce the motivation for studying *Cognition and Intractability*. We provide an intuitive introduction to the problem of intractability as it arises for models of cognition using an illustrative everyday problem as running example: selecting toppings on a pizza. Next, we review relevant background information about the conceptual foundations of cognitive explanation, computability, and tractability. At the end of this chapter the reader should have a good understanding of the conceptual foundations of the Tractable Cognition thesis, including its variants: The P-Cognition thesis and the FPT-Cognition thesis, which motivates diving into the technical concepts and proof techniques covered in Chapters 2–7.

1.1 Selecting Pizza Toppings

Imagine you enter a pizzeria to buy a pizza. You can choose *any* combination of toppings from a given set, e.g., {*pepperoni*, *salami*, *ham*, *mushroom*, *pineapple*, . . . }. What will you choose?

According to one account of human decision-making, your choice will be such that you maximize utility. Here utility, denoted by $u(.)$, is to be understood as the subjective value of each possible choice option (e.g., if you prefer salami to ham, then $u(salami) > u(ham)$). Since you can choose combinations of toppings, we need to think of the choice options as subsets of the set of all available toppings. This includes subsets with only one element (e.g., {*salami*} or {*olives*}), but also combinations of multiple toppings (e.g., {*ham*, *pineapple*} or {*salami*, *mushrooms*, *olives*}). On this account of human decision-making, we can formally describe your toppings selection problem as an instance of the following computational problem:

Generalized Subset Choice
Input: A set $X = \{x_1, x_2, ..., x_n\}$ of n available items and a value function u assigning a value to every subset $S \subseteq X$.
Output: A subset $S \subseteq X$ such that $u(S)$ is maximized over all possible $S \subseteq X$.

Note that many other choice problems that one may encounter in everyday life can be cast in this way, ranging from inviting a subset of friends to a party or buying groceries in the grocery store to selecting people for a committee or prescribing combinations of medicine to a patient.

But how – using what algorithm – could your brain come to select a subset S with maximum utility? A conceivable algorithm could be the following: Consider each possible subset $S \subseteq X$, in serial or parallel (in whatever way we may think the brain implements such an algorithm), and select the one that has the highest utility $u(S)$. Conceptually this is a straightforward procedure. But it has an important problem. The number of possible subsets grows exponentially with the number of items in X. Given that in real-world situations one cannot generally assume that X is small, the algorithm will be searching prohibitively large search spaces.

Stop and Think

These days pizzerias may provide for 30 or more different toppings. How many distinct pizzas do you think can be made with 30 different pizza toppings?

If n denotes the number of items in X, then 2^n expresses the number of distinct possible subsets of X. In other words, with 30 toppings one can make $2^{30} > 1,000,000,000$ (a billion) distinct pizzas. Of course, in practice pizzerias typically list 20–30 distinct pizzas on their menus. But consider that some pizzerias also provide the option to construct your own pizza, implicitly allowing a customer to pick any of the billion pizza options available.

Stop and Think

Imagine that the brain would use the exhaustive algorithm described earlier for selecting the preferred pizza. Assume that the brain's algorithm would process 100 possible combinations, in serial or parallel, per second. How long would it take the brain to select a pizza if it could choose from 30 different pizza toppings? What if it could choose from 40 different toppings?

You may be surprised to find that the answer is 4 months. That is the time it takes in this scenario for your brain to consider all the distinct pizzas that can be made with 30 toppings in order to find the best tasting one (maximum utility). If the choice would be from 40 toppings, it would even take 3.5 centuries. Evidently, this is an unrealistic scenario. The pizzeria would be long closed before you would have made up your mind!

There is an important lesson to draw from our pizza example: Explaining how agents (human or artificial) can make decisions in the real world, where time is a costly resource and choice options can be plentiful, requires algorithms that run in a realistic amount of time. The exhaustive algorithm that we considered in our pizza scenario does not meet this criterion. It is an *exponential-time* algorithm. The time it takes grows exponentially with the input size n (i.e., grows as c^n for some constant $c > 1$). Exponential time grows faster than any polynomial function (a function of the form n^c for some constant c), and is therefore also referred to as non-polynomial time. Another example of non-polynomial time is factorial time (grows as $n!$). An example of a factorial-time algorithm would be an algorithm that exhaustively searches all possible orderings of n events or actions in order to select the best possible ordering. Consider, for instance, planning n activities in a day: going to the hairdresser, doing the laundry, buying groceries, cooking food, washing the dishes, posting a letter, answering an email, watching TV, etc. Even for as few as 10 activities, there would be 3.6 million possible orderings, and for 20 activities there would be more than 10^{18} possible orderings. Planning one's daily activities by exhaustive search would be as implausible as selecting pizza toppings by exhaustive search.

Table 1.1 illustrates why non-polynomial time (e.g., exponential or factorial) algorithms generally are considered *intractable* for all but small input sizes n, whereas polynomial-time algorithms (e.g., linear or quadratic) are considered *tractable* even for larger input sizes. Informally, intractable means that the

Table 1.1 Illustration of the running times of polynomial time versus super-polynomial time algorithms. The function $t(n)$ expresses the number of steps performed by an algorithm (linear, quadratic, exponential, or factorial). For illustrative purposes it is assumed that 100 steps can be performed per second.

Input size	Polynomial time		Non-polynomial time	
n	$t(n) = n$	$t(n) = n^2$	$t(n) = 2^n$	$t(n) = n!$
5	50 ms	250 ms	320 ms	1 sec
10	100 ms	1 sec	10 sec	10.1 hr
20	200 ms	4 sec	2.9 hr	7.7×10^6 centuries
30	299 ms	9 sec	4.1 months	8.4×10^{20} centuries
40	400 ms	16 sec	3.5 centuries	2.6×10^{36} centuries
50	500 ms	25 sec	3.6×10^3 centuries	9.6×10^{52} centuries
100	1 sec	1.7 min	4.0×10^{18} centuries	3.0×10^{146} centuries
500	5 sec	41.7 min	1.0×10^{139} centuries	4.0×10^{1124} centuries
1,000	10 sec	2.8 hr	3.4×10^{289} centuries	1.3×10^{2558} centuries

algorithm requires an unrealistic amount of computational resources (in this case, time) for its completion. This intractability is the main topic of this book. In this book we explore formal notions of (in)tractability to assess the computational-resource demands of different (potentially competing) scientific accounts of cognition, be they about decision-making, planning, perception, categorization, reasoning, learning, etc. Even though brains are quite remarkable, their speed of operation is limited, and this fact can be exploited to assess the plausibility of different ideas scientists may have about "what" and "how" the brain computes.

For illustrative purposes, Table 1.1 assumed that the listed algorithms could perform 100 steps per second. To see that this assumption has little effect on the large difference between polynomial and non-polynomial running times perform the next practice.

Practice 1.1.1 Recompute the contents of Table 1.1 under the assumption that the algorithms can perform as many as 1,000 steps per second.

Let us return to our pizza example. We saw that the exhaustive algorithm (searching all possible subsets) to maximize utility of the chosen subset of toppings is an intractable algorithm. Does this mean that the idea that humans maximize utility in such a situation is false? Possibly, but not necessarily. Note that the trouble may have arisen from the specific way in which we formalized the maximum utility account of decision-making for subset choice. In the Generalized Subset Choice problem we allowed for any possible utility function u that assigned any possible value to every subset X. As a result, there is only one way to be sure that we output a subset with maximum utility: We need to consider each and every subset.

The situation would be less dire if somehow there would be regularity in one's preferences over pizza toppings. This regularity could then perhaps be exploited to more efficiently search the space of choice options. For instance, if subjective preferences would be structured such that the utility of a subset could be expressed as the sum of the value of its elements (i.e., $u(S) = \sum_{x \in S} u(x)$), then we could change the formalization as follows:

Additive Subset Choice
Input: A set $X = \{x_1, x_2, ..., x_n\}$ of n available items and a value function u assigning a value to every element $x \in X$.
Output: A subset $S \subseteq X$ such that $u(S) = \sum_{x \in S} u(x)$ is maximized over all possible $S \subseteq X$.

If the pizza selection problem would be an instance of this formal problem, then the brain could select a maximum utility subset by using the following

simple linear-time algorithm: Consider each item $x \in X$, and if $u(x) \geq 0$ then add x to the subset S, otherwise discard the option x. Since each item in X has to be considered only once, and the inequality $u(x) \geq 0$ checked for each item only once, the number of steps performed by this algorithm grows at worst linearly with the number of options in X. As can be seen in Table 1.1, such a linear-time algorithm is clearly tractable in practice, even when you have larger numbers of toppings to choose from.

The maximum utility account of decision-making would thus be saved from intractability, if indeed real-world subset choice problems could all be cast as instances of the ADDITIVE SUBSET CHOICE problem. But is this a plausible possibility?

Stop and Think

Consider selecting pizza toppings for your pizza using the linear-time algorithm described earlier? Why may you not be happy with the actual result?

If you would use the linear-time algorithm to select your pizza toppings, you would always end up with all positive valued toppings on your pizza. Besides that this may make for an overcrowded pizza, it also fails to take into account that you may like some toppings individually but not in particular combinations. For instance, each of the items in the set $\{pepperoni, salami, ham, mushroom, pineapple\}$ could have individually positive value for you, in the sense that you would prefer a pizza with any one of them individually over a pizza with no toppings. Yet, at the same time, you may prefer $\{ham, pineapple\}$ or $\{salami, mushrooms, olives\}$ over a pizza will all the toppings (e.g., because you dislike the taste of the combination of pineapple with olives). In other words, in real-world subset choice problems, there may be interactions between items that affect the utility of their combinations. This makes $u(S) = \sum_{x \in S} u(x)$ an invalid assumption. From this exploration, we should learn an important lesson: Intractable formalizations of cognitive problems (decision-making, planning, reasoning, etc.) can be recast into tractable formalizations by introducing additional constraints on the input domains. Yet, it is important to make sure that those constraints do not make the new formalization too simplistic and unable to model real-world problem situations.

A balance may be struck by introducing the idea of pair-wise interactions between k items in the choice set. Then we can adapt the formalization as follows:

BINARY SUBSET CHOICE
Input: A set $X = \{x_1, x_2, ..., x_n\}$ of n available items. For every item $x \in X$ there is an associated value $u(x)$, and for every pair of items (x_i, x_j) there is an associated value $\delta(x_i, x_j)$.
Output: A subset $S \subseteq X$, such that $u(S) = \sum_{x \in S} u(x) + \sum_{x, y \in S} \delta(x, y)$ is maximum.

If situations allow for three-way interactions, this model may also fail as a computational account of subset choice. It is certainly conceivable that three-way interactions can occur in practice (see, e.g., van Rooij, Stege, and Kadlec, 2005). Leaving that discussion for another day, we may ask ourselves the following question: Would computing this BINARY SUBSET CHOICE problem be in principle tractable? It is not so easy to tell as for GENERALIZED SUBSET CHOICE, because the utility function is constrained. But is it constrained enough to yield tractability of this computation? Probably not. Using the tools that you will learn about in this book, you will be able to show that this problem belongs to class of so-called NP-hard problems. This is the class of problems for which no polynomial-time algorithms exist unless a widely conjectured inequality $\mathsf{P} \neq \mathsf{NP}$ would be false. This $\mathsf{P} \neq \mathsf{NP}$ conjecture, although formally unproven (and perhaps even unprovable), is widely believed to be true among computer scientists and cognitive scientists alike (see Chapter 4 for more details). Likewise, we will adopt this conjecture in the remainder of this book.

1.2 Conceptual Foundations

In our pizza example we have introduced many of the key scientific concepts on which this book builds. For instance, we used the distinction made in cognitive science between explaining the "what" and the "how" of cognition, the notion of "algorithm" as agreed upon by computer scientists, and the idea that "intractability" can be characterized in terms of the time complexity of algorithms. In this section, we explain the conceptual foundations of these concepts in a bit more detail.

1.2.1 Conceptual Foundations of Cognitive Explanation

One of the primary aims of cognitive science is to explain human cognitive capacities. Ultimately, the goal is to answer questions such as: How do humans make decisions? How do they learn language, concepts, and categories? How

do they form beliefs, based on reasons or otherwise? In order to come up with answers for such "how"-questions it can be useful to first answer "what"-questions: What is decision-making? What is language learning? What is categorization? What is belief fixation? What is reasoning?

This distinction between "what is being computed" (the *input-output mapping*) and "how it is computed" (the *algorithm*) is also reflected in the influential and widely used explanatory framework proposed by David Marr (1981). Marr proposed that, ideally, theories in cognitive science should explain the workings of a cognitive system (whether natural or artificial) on three different levels (see Table 1.2). The first level, called the *computational level*, specifies the nature of the input-output mapping that is computed (we will also refer to this as the *cognitive function*).[1] The second level, the algorithmic level, specifies the nature of the algorithmic process by which the computation described at the computational level is performed (*cognitive process*). The third and final level, the implementation level, specifies how the algorithm defined at the second level is physically implemented by the "hardware" of the system (or "wetware" in the case of the brain) performing the computation (*physical implementation* of the cognitive process/function).

Hence, in David Marr's terminology, the description of a cognitive system in terms of the function that it computes (or problem that it solves)[2] is called a *computational-level theory*. We already saw examples when we discussed the pizza example: i.e., Generalized, Additive, and Binary Subset Choice were three different candidate computational-level theories of how humans choose subsets of options. Since one and the same function can be computed by

[1] We should note that Marr also intended the computational-level analysis to include an account of "why" the cognitive function is the appropriate function for the system to compute, given its goals and environment of operation. This idea has been used to argue for certain computational-level explanations based on appeals to rationality and/or evolutionary selection – i.e., that specific functions would be rational or adaptive for the system to compute. The intractability analysis of computational-level accounts as pursued in this book are neutral with respect to such normative motivations for specific computational-level accounts, in the sense that tractability and rational analysis are compatible, but the former can be done independent of the latter (see Section 8.5).

[2] Since the words "function" and "problem" refer to the same type of mathematical object (an input-output mapping) we will use the terms interchangeably. A difference between the terms 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). The reader may notice that we will tend to adopt the convention of speaking of "problems" whenever we discuss computational complexity concepts and methods from computer science (e.g., in Chapters 2–7), and adopt the terms "function" or "computational-level theory" in the context of applications and debates in cognitive science (e.g., in Chapters 8–12).

Table 1.2 Marr's levels of explanation: What is the type of question asked at each level, what counts as an answer (the explanans), and labels for the thing to be explained (explanandum) per level.

Level	Question	Answer	Label
Computation	What is the nature of the computational problem solved?	An input-output mapping F: $I \rightarrow O$	Cognitive function
Algorithm	How is the computational problem solved?	An algorithm A that computes F	Cognitive process
Implementation	How is the algorithm implemented?	A specification of how the computational steps of A are realizable by the relevant "stuff" (e.g., neuronal processes)	Physical implementation

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 of physical considerations.

David Marr, in his seminal 1981 book, illustrated this idea with the example of a cash register, i.e., a system that has the ability to perform *addition* (see Figure 1.1). A computational-level theory for a cash register would be the Addition function $F(a,b) = a + b$. An algorithmic-level theory could, for instance, be an algorithm operating on decimal numbers or an algorithm operating on binary numbers. Either algorithm would compute the function Addition, albeit in different ways. The implementational-level theory would depend on the physical make-up of the system. For instance, different physical systems can implement algorithms for Addition: cash registers, pocket calculators, and even human brains. An implementational-level theory would specify by some sort of blueprint how the algorithm could be realized by that particular physical system.

Practice 1.2.1 Study the cash-register example in Figure 1.1. Can you come up with different computational-, algorithmic- and implementational-level