1
Counting Problems

This book is a study of the computational complexity of counting problems, especially those that can be expressed as a sum-of-product computation. Our aim is to give a systematic presentation to a body of work, mostly from the past two decades, that gives some comprehensive classifications to these sum-of-product computations from the perspective of computational complexity. All these sum-of-product problems are computable within the level of the complexity class #P. The classification theorems are stated in the following form, known as a dichotomy: For a class of problems expressible within a framework, every problem in the class is either computable in polynomial time, or it is #P-hard to compute, i.e., it is as hard as any other problem in the class #P.

1.1 Counting Problems and Models of Computation

We assume the readers have some basic and preliminary knowledge about computational complexity such as P and NP, for example, at a level provided by an undergraduate course on the subject of the Theory of Computing. This knowledge is not crucial, however, as we will not use many general results but rather present the framework and develop the necessary proof techniques gradually. Any reader wishing to be more thoroughly acquainted with the full scope of computational complexity theory can consult a standard textbook, e.g., [Sip96, Pap94, AB09]. On the other hand, a reader with no prior exposure to complexity theory but with a strong mathematical background, for example, at a level provided by a solid undergraduate mathematics education, should be able to follow all proofs in the book, provided he or she is willing to accept a handful of results without proof. These results can be found elsewhere, and the particular proofs of these do not impact an understanding of the material in this book. Such a reader, however, would benefit from a wider exposure to

complexity theory to gain some insight as to why certain problems are investigated and what questions are asked.

Briefly, in computational complexity theory a *problem* consists of an infinite set of problem instances, rather than one particular instance. For example, the problem DETERMINANT is to compute the determinant of an arbitrarily given matrix, not one particular matrix. Solving a problem is to say that there is an algorithm that solves all problem instances. The formal notion of an algorithm is a Turing machine, an idealized computer that carries out step-by-step operations according to a finitary program, valid for all problem instances. The efficiency of the algorithm is measured in terms of the maximum number of steps $T(n)$ the Turing machine may take for all problem instances of size n. For example, for integer matrices, the problem instance of DETERMINANT is an integer square matrix in which the size is the sum of the bit length of all the matrix entries. Formally the complexity classes P and NP are defined for decision problems, those problems for which the answer to each instance is either Yes or No. For example, the problem VERTEX COVER is the following decision problem: Given a problem instance consisting of a graph $G = (V, E)$ and an integer parameter k, whether there is a subset of vertices $S \subseteq V$ such that $|S| \leq k$ and every edge in E is incident to at least one vertex from S. A decision problem Π is in the complexity class P if there is an algorithm solving the problem, i.e., gives a correct Yes or No answer to every problem instance, such that $T(n)$ is bounded above by a fixed degree polynomial in n. The class NP consists of all decision problems Σ for which there exists some problem Π in P, and some polynomial $p(\cdot)$, such that for all problem instances x of Σ, x is a Yes instance of Σ iff there exists some y with size $|y| \leq p(|x|)$ and the pair $\langle x, y\rangle$ is a Yes instance of Π. Intuitively, y is a certificate of size polynomially bounded in x, such that it can be verified in polynomial time that y certifies that x is a Yes instance of Σ.

There are historical as well as internal logical reasons, mainly from computability theory, why the definitions of P and NP are formulated in terms of decision problems. Even though a problem may be more naturally stated as a search problem or a numerical problem, it can often be restated as a decision problem such that solving the decision problem is equivalent to solving the search or numerical problem within a polynomial factor in efficiency. For example, if there is an algorithm that solves the decision problem VERTEX COVER, then by binary search on the value k one can find the minimum size k_0 of any vertex cover of G. Furthermore, by considering a sequence of at most $O(k_0 n)$ graphs obtained by removing some vertices and their incident edges from G of n vertices, and repeatedly invoking the decision algorithm, one can compute a minimum size vertex cover of G. For the DETERMINANT problem one can

formulate a decision problem as whether the determinant of an integer matrix is greater than a threshold value. It is easy to compute an a priori bound for a given determinant; a binary search will then find the exact determinant value in polynomial time from the decision algorithm. Such reductions show that there is a theoretical equivalence between a decision problem and its search or numerical version; however, in reality, a polynomial-time algorithm usually directly computes the output value, without going through the decision version.

In this book we mostly study counting problems. Typically these include counting the number of specific combinatorial configurations, such as vertex covers, matchings, or proper colorings in a graph, or the evaluation of partition functions in statistical physics. For such problems, it is not expected that they are equivalent to their decision problem counterparts in polynomial time [Tod91]. Instead we define #P, a complexity class of functions for counting problems.

For every NP problem Σ one can formulate a corresponding counting problem using the problem Π in P that defines Σ in terms of certificates. For any problem instance x of Σ, we define the function $f(x)$ as the number of certificates y such that the pair $\langle x, y\rangle$ is a Yes instance of Π. The class of all such functions is denoted as #P. This complexity class, defined by Valiant [Val79a, Val79b] in the study of the complexity of the permanent function, will play a central role in this book. Natural counting problems corresponding to decision problems at the level of NP that have a nonnegative integer solution can all be formulated as a problem in #P. To include more problems at this level we also consider problems whose solutions are not necessarily nonnegative integers, such as those from statistical physics. To capture their computational complexity, we consider the closure of #P under polynomial-time Turing reductions, namely, those problems solvable by a polynomial-time algorithm given free access to a hypothetical algorithm to some problem in #P, where each query costs only the time it takes to write the query. This class is formally denoted as $\mathrm{FP}^{\#\mathrm{P}}$, where FP denotes the class of functions computable by a polynomial time Turing machine. Typical problems in #P include counting the number of satisfying assignments to a Boolean formula or the number of vertex covers in a graph. Weighted versions of these problems as well as sum-of-product computations such as partition functions from statistical physics can be easily formulated as problems that are polynomial-time reducible to #P.

The formal Turing machine model is naturally suited to the study of computation over discrete structures such as integers or graphs. However, in this book it is more natural to consider computation over the real or complex numbers. Doing so causes a technical issue of how one may represent exactly the individual real or complex numbers, and how to account for the complexity of various

operations on these numbers. This is a question relating to the model of computation. One can, for example, take the computational model on real numbers by Blum–Shub–Smale [BSS89, BCSS98]. This is more intuitive; however, in a strict logical sense this is not equivalent to the classical Turing machine model and the results we obtain would not have the same logical meaning. So, formally we still consider the classical Turing machine model, and restrict the objects of computation to algebraic (complex) numbers. Thus, technically, every number α is specified by a finite irreducible polynomial $f(X) \in \mathbb{Z}[X]$ with integer coefficients, and a small disk of rational radius containing a unique root α of $f(X)$. We discuss models of computation further in Section 1.5. But this is not a central issue for the type of sum-of-product computations we treat in this book; basically the theory can be developed in any reasonable model of computation in which sum and product can be efficiently computed.

Some basic notations. We denote by $\mathbb{N}$ the set of natural numbers $\{0, 1, 2, \ldots\}$, by $\mathbb{Z}$ the set of all integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, and by $\mathbb{Z}_+$ the set of positive integers $\{1, 2, 3, \ldots\}$. We denote by $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ the sets of rational (algebraic) real, and complex numbers respectively. We denote $\mathrm{i} = \sqrt{-1}$. All graphs are finite and undirected, unless stated otherwise. Graphs may or may not have loops and parallel edges, which should be clear from the context. If α and β are finite bit strings, then α_i denotes its ith bit, and $\alpha \oplus \beta$ denotes its bitwise XOR string. For a binary string α, its Hamming weight is denoted by $\mathrm{wt}(\alpha)$, i.e., the number of 1's in α. We will generally treat a column vector $v \in \mathbb{C}^n$ and the row vector v^{T}, its transpose, interchangeably. For a matrix $M \in \mathbb{C}^{m\times n}$ and a $v \in \mathbb{C}^n$, we write Mv as its matrix vector product. Sometimes we may write Mv even though we explicitly listed the elements of v as a row vector. The formal meaning is just Mv^{T}, if v is a row vector. Similar comments apply for vM. If $A = (a_{ij}) \in \mathbb{C}^{m\times n}$ and $B = (b_{st}) \in \mathbb{C}^{k\times \ell}$, we denote $A \otimes B$ as the tensor product matrix in $\mathbb{C}^{mk\times n\ell}$, indexed by the pairs $(i, s) \in [m] \times [k]$ for rows and $(j, t) \in [n] \times [\ell]$ for columns, both in lexicographic order.

We use $\leq_{\mathrm{T}}^{\mathrm{P}}$ or simply $\leq_{\mathrm{T}}$ to denote polynomial time Turing reducibility, and $\equiv_{\mathrm{T}}^{\mathrm{P}}$ or simply $\equiv_{\mathrm{T}}$ to denote polynomial time Turing equivalence.

1.2 Three Classes of Counting Problems

We now formally define three frameworks for the types of sum-of-product computations in this book. Fix a finite domain $[q] = \{1, 2, \ldots, q\}$, for a positive integer q. If $q = 2$ it is called the Boolean domain. We consider any set of functions $\mathcal{F}$ on domain $[q]$, where each $f \in \mathcal{F}$ maps from $[q]^k$ to a commutative semiring $\mathbf{R}$, for some $k \geq 0$, called the arity of f. If $k = 0$, then f is a constant. If $k = 1$, 2, or 3, then f is called a unary, binary, or ternary function

respectively. In this book, by default we will take the semiring $\mathbf{R}$ to be the (algebraic) complex numbers, but still denote it as $\mathbb{C}$. Functions in $\mathcal{F}$ are also called *signatures* or local constraint functions. A signature f is symmetric if its value is invariant under permutation of its variables. For the Boolean domain this means that the value of f depends only on the Hamming weight of its input.

1.2.1 Spin Systems or Graph Homomorphism Problems

A spin system on a graph $G = (V, E)$ is the following model and it comes from statistical physics. Let $[q] = \{1, \ldots, q\}$ be a finite domain, where $q \geq 1$ is an integer. We consider all vertex assignments $\sigma : V \to [q]$. There is an edge function $f : [q]^2 \to \mathbb{C}$. For each assignment σ we have an evaluation $\prod_{(u,v)\in E} f(\sigma(u), \sigma(v))$, a product over every edge $(u, v) \in E$. Then we define the *partition function*

$$Z_f(G) = \sum_{\sigma:V\to[q]} \prod_{(u,v)\in E} f(\sigma(u), \sigma(v)). \tag{1.1}$$

In physics, the partition function represents the aggregate of thermodynamic variables or a normalizing factor of a system, as one sums over all possible configurations of the particles.

The value $f(\sigma(u), \sigma(v))$ is the local contribution, the product $\prod_{(u,v)\in E} f(\sigma(u), \sigma(v))$ is the weight of the assignment σ, and the partition function is the sum of weights over all assignments. If G is an undirected graph, as is typically the case, we require f to be a symmetric function, $f(i, j) = f(j, i)$, for all $i, j \in [q]$.

If $q = 2$ this is called a 2-spin system, and for general q it is called a q-spin system.

Well-known examples of a 2-spin system include the Ising model [Isi25], where $f(0, 0) = f(1, 1) = a$ and $f(0, 1) = f(1, 0) = b$ for some two constants a and b. Sometimes there is also a vertex weight function, represented by a unary function normalized to $u(0) = 1$ and $u(1) = \lambda$. Then the partition function of the Ising model is

$$Z_{a,b,\lambda}(G) = \sum_{\sigma:V\to[2]} a^{|\{(u,v)\in E:\sigma(u)=\sigma(v)\}|} b^{|\{(u,v)\in E:\sigma(u)\neq\sigma(v)\}|} \lambda^{|\{v\in V:\sigma(v)=1\}|}. \tag{1.2}$$

Exercise: Fix any constant J. One can define the Hamiltonian function $H(\sigma)$ for any assignment $\sigma : V \to \{-1, +1\}$,

$$H(\sigma) = -J \sum_{(u,v)\in E} \sigma(u)\sigma(v).$$

(It is called ferromagnetic if $J > 0$ and antiferromagnetic if $J < 0$.) Then the partition function of the Ising model can be defined by $Z(G) = \sum_{\sigma: V \to \{-1,+1\}} e^{-\beta H(\sigma)}$, where $\beta \geq 0$ is called the inverse temperature. Show that this function $Z(G)$ can be realized by the sum-of-product expression $Z_{a,b,\lambda}(G)$.

A generalization of the Ising model is called the Potts model [Pot52], where for $q \geq 2$ and $i, j \in [q]$, $f(i, i) = a$ and $f(i, j) = f(j, i) = b$ for $i \neq j$. One can normalize $b = 1$ and write a as $1 + \gamma$ for a parameter γ. In this parameterization the partition function of the Potts model is

$$Z_{\text{Potts}}(G; q, \gamma) = \sum_{\sigma: V \to [q]} \prod_{(u,v) \in E} (1 + \gamma \delta(\sigma(u), \sigma(v))), \tag{1.3}$$

where $\delta(i, j) = 1$ if $i = j$, and 0 otherwise.

The partition function of the Potts model can be linked to the Tutte polynomial $\mathrm{T}(G; x, y)$ by setting $\gamma = y - 1$ and $q = (x - 1)(y - 1)$. Indeed, one way to define the Tutte polynomial in terms of q and γ is that $(x - 1)^{\kappa(V,E)}(y - 1)^{|V|}\, \mathrm{T}(G; x, y)$ is equal to (cf. Definition 6.27)

$$Z_{\text{Tutte}}(G; q, \gamma) = \sum_{F \subseteq E} q^{\kappa(V,F)} \gamma^{|F|},$$

where $\kappa(V, F)$ is the number of connected components in the spanning subgraph (V, F).

Although the Tutte polynomial is defined for any q, if we restrict to a positive integer q, then

$$Z_{\text{Tutte}}(G; q, \gamma) = Z_{\text{Potts}}(G; q, \gamma).$$

To prove this equality, we consider expanding the product $\prod_{(u,v) \in E}(1 + \gamma \delta(\sigma(u), \sigma(v)))$ in (1.3) as a sum indexed by $F \subseteq E$, which collects a factor $\gamma^{|F|}$, namely

$$\prod_{(u,v) \in E} (1 + \gamma \delta(\sigma(u), \sigma(v))) = \sum_{F \subseteq E} \gamma^{|F|} \prod_{(u,v) \in F} \delta(\sigma(u), \sigma(v)).$$

The product term indexed by F is 1 iff $\sigma(u) = \sigma(v)$ for all u and v that belong to the same connected component of the subgraph (V, F), and 0 otherwise. In (1.3) the sum $\sum_{\sigma: V \to [q]}$ has exactly $q^{\kappa(V,F)}$ such terms.

Spin systems are also called graph homomorphisms, and they come from a different source. Given two graphs G and H, a graph homomorphism from G to H is a map σ from the vertex set of G to the vertex set of H such that for every edge (u, v) in G, the image is also an edge in H. In general, multigraphs are allowed; thus, for example, an edge in G can be mapped to a loop in H. The

(a) #VertexCover (b) #Antichain

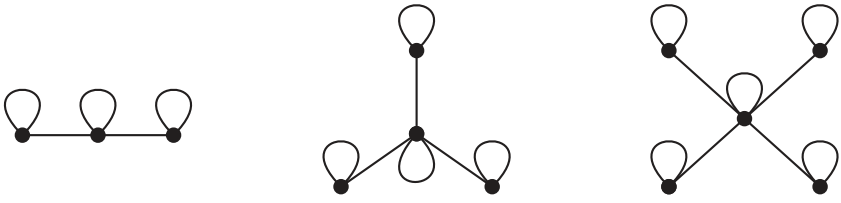
Figure 1.1. Target graphs H and the combinatorial counting problems they define as #H-coloring problems.

counting problem is to compute the number of graph homomorphisms from G to H.

We will fix an H, and consider the computational problem where G is the input. A special case is when $H = K_q$, the complete graph on q vertices, without self-loops. In this case, a graph homomorphism from G to H is a valid coloring of the vertices of G, using at most q distinct colors, as any adjacent pair of vertices of G must be mapped to distinct vertices of H. Owing partly to this special case, counting graph homomorphisms to H in general is also called #H-colorings, and H is called the target graph.

Graph homomorphisms can express a variety of combinatorial problems. In fact, their principal purpose is to express and then treat a wide variety of locally defined graph properties in a uniform way. For example, if H is a graph consisting of two vertices $\{\mathtt{T}, \mathtt{F}\}$ and two edges $\{(\mathtt{T}, \mathtt{T}), (\mathtt{T}, \mathtt{F})\}$, a loop and an edge between the two vertices (see Figure 1.1a), then the #H-coloring problem is the counting problem of Vertex Cover, i.e., to count the number of vertex covers in graph G, denoted as #VC. Indeed a homomorphism from G to this H is a two-coloring of vertices of G such that the subset of vertices of G mapped to $\mathtt{T}$ forms a vertex cover. By flipping the intended meaning of $\mathtt{T}$ and $\mathtt{F}$, this also counts the number of independent sets in G. Of course this simply reflects the fact that a subset $S \subseteq V(G)$ is a vertex cover iff its complement $S^c \subseteq V(G)$ is an independent set. As another example, suppose H is the two-vertex directed graph connected by a single directed edge and both vertices have one directed self-loop (see Figure 1.1b). Then the #H-coloring problem takes directed graphs as input. If the input is a directed acyclic graph, then it defines a partial order, and the #H-coloring problem is to compute the number of antichains (or equivalently, the number of lower sets, i.e., downward closed sets in the partial order, or equivalently, the number of upper sets) in this partial order. If the input is not acyclic, then every strongly connected component must be mapped to one vertex in H, and the #H-coloring problem is to count the number of antichains in the induced partial order after collapsing each strongly connected component.

Exercise: Show that #VC can be expressed as #H-coloring by defining explicitly the binary constraint function.



(a) $q = 2$ (b) $q = 3$ (c) $q = 4$

Figure 1.2. Target graph H for counting q-particle Widom–Rowlinson configurations for $q \in \{2, 3, 4\}$ as an $\#H$-coloring problem.

Exercise: Show that there is a bijection between the set of antichains and the set of lower sets. Conclude that the number of antichains is the same as the number of lower sets. By symmetry the same is proved for upper sets.

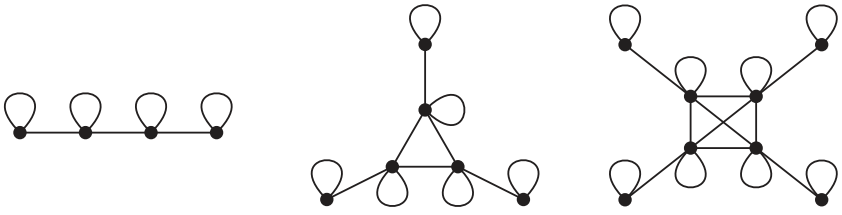
More generally, we consider weighted graph homomorphisms . Let A be a q-by-q matrix over $\mathbb{C}$. Given a graph or a directed graph $G = (V, E)$, the (weighted) graph homomorphism problem is to compute

$$Z_A(G) = \sum_{\sigma: V \to [q]} \prod_{(u,v) \in E} A_{\sigma(u),\sigma(v)}. \tag{1.4}$$

The target graph H is now defined by the matrix A, which is the weighted adjacency matrix of H. When A is a 0-1 matrix, then this is the unweighted version of graph homomorphism. The matrix A can also be identified as a binary function. In the case of Vertex Cover the function is the binary Boolean Or function. For Independent Set this is the Nand function. For the problem of counting antichains, the function is the binary Implication function. For q-coloring, this is the binary Disequality function on domain $[q]$.

This is exactly the same notion of a partition function in statistical physics. The Ising model corresponds to the partition function with matrix $A = \begin{bmatrix} a & b \\ b & a \end{bmatrix}$. The Potts model corresponds to the partition function with matrix $A = J_q + \gamma I_q$, were J_q is the $q \times q$ matrix of all 1's and I_q is the $q \times q$ identity matrix, and γ is a parameter.

The q-particle Widom–Rowlinson model [WR70] corresponds to the $\#H$-coloring problem in which the domain size is $q + 1$, H is the star graph on $q + 1$ vertices and all vertices have self-loops (see Figure 1.2). The q-type Beach model [BS94, BS95] corresponds to the $\#H$-coloring problem in which the domain size is $2q$, H is the complete graph on q vertices, each of these q vertices has a pendant vertex, and all $2q$ vertices have a self-loop (see Figure 1.3).



(a) $q = 2$ (b) $q = 3$ (c) $q = 4$

Figure 1.3. Target graph H for counting q-type Beach model configurations for $q \in \{2, 3, 4\}$ as an #H-coloring problem.

For more on this connection with statistical physics, see [Wel93, Chapter 4] or [Thu09, Chapter 2] (as well as [Bax82]).

Graph homomorphism can be viewed as a special case of counting constraint satisfaction problems, where instead of one binary constraint function there can be a set of constraint functions.

1.2.2 Constraint Satisfaction Problems

A counting constraint satisfaction problem (#CSP) [CKS01] is parametrized by a set of local constraint functions $\mathcal{F}$. It is denoted by #CSP$_q(\mathcal{F})$ when the constraint functions in $\mathcal{F}$ are defined over a domain $[q]$ of size q. An instance of #CSP$_q(\mathcal{F})$ is a finite set of variables $x_1, x_2, \ldots, x_n$, and a finite set C of clauses. Each clause is a constraint $f \in \mathcal{F}$ of some arity k depending on f together with a sequence of k (not necessarily distinct) variables $x_{i_1}, \ldots, x_{i_k} \in \{x_1, x_2, \ldots, x_n\}$. The output is

$$\sum_{x_1, \ldots, x_n \in [q]} \prod_{(f, x_{i_1}, \ldots, x_{i_k}) \in C} f(x_{i_1}, \ldots, x_{i_k}). \tag{1.5}$$

In the study of #CSP$_q(\mathcal{F})$, the set $\mathcal{F}$ is usually finite and considered fixed. In particular, there is a maximum arity k among functions in $\mathcal{F}$. An input instance on n variables can be described by n^k bits for a fixed k. Hence we consider the input size is n. When it is the Boolean domain (i.e., $q = 2$), we denote it simply as #CSP$(\mathcal{F})$. If $\mathcal{F}$ consists of a single function f we write #CSP(f) for #CSP$(\{f\})$. We write similarly #CSP(f, g) if $\mathcal{F} = \{f, g\}$, and #CSP$(\mathcal{F}, g)$ for #CSP$(\mathcal{F} \cup \{g\})$, etc.

The canonical example of a #CSP is #Sat, or counting Boolean Satisfiability, the problem of counting the number of satisfying assignments to a given Boolean formula. As a constraint satisfaction problem, it is #CSP$(\mathcal{F})$, with

$\mathcal{F} = \{\text{OR}_k | k \geq 1\} \cup \{\neq_2\}$, where OR_k is the OR function of arity k and $(\neq_2)$ is the binary DISEQUALITY function. Here are several more well-known examples of #CSP($\mathcal{F}$)'s over the Boolean domain and their corresponding constraint sets:

SAT	has	$\mathcal{F} = \{\text{OR}_k \mid k \geq 1\} \cup \{\neq_2\}$
3SAT	has	$\mathcal{F} = \{\text{OR}_3, \neq_2\}$
1-IN-3SAT	has	$\mathcal{F} = \{\text{EXACT-ONE}_3, \neq_2\}$
NAE-3SAT	has	$\mathcal{F} = \{\text{NOT-ALL-EQUAL}_3, \neq_2\}$
MON-SAT	has	$\mathcal{F} = \{\text{OR}_k \mid k \geq 1\}$
MON-3SAT	has	$\mathcal{F} = \{\text{OR}_3\}$
MON-1-IN-3SAT	has	$\mathcal{F} = \{\text{EXACT-ONE}_3\}$
MON-NAE-3SAT	has	$\mathcal{F} = \{\text{NOT-ALL-EQUAL}_3\}$

By #CSP$^d(\mathcal{F})$, we denote the special case of #CSP($\mathcal{F}$) in which every variable appears a multiple of d times. Note that #CSP($\mathcal{F}$) is the same as #CSP$^1(\mathcal{F})$, and #CSP$^2(\mathcal{F})$ is the same as every variable appearing an even number of times.

1.2.3 Holant Problems

A Holant problem is parametrized by a set of local constraint functions $\mathcal{F}$, also called signatures . A *signature grid* $\Omega = (G, \pi)$ over $\mathcal{F}$ consists of a graph $G = (V, E)$ and a mapping π that assigns to each vertex $v \in V$ an $f_v \in \mathcal{F}$ and a linear order of the incident edges at v. The arity of f is equal to the degree at v, and the incident edges at v are associated with the input variables of f_v. If all signatures in $\mathcal{F}$ are symmetric then there is no need to assign an order for incident edges at any v.

Definition 1.1. For a set $\mathcal{F}$ of signatures over a domain $[q]$, we define Holant$_q(\mathcal{F})$ as

Input: A *signature grid* $\Omega = (G, \pi)$ over $\mathcal{F}$

Output:

$$\text{Holant}_q(\Omega; \mathcal{F}) = \sum_{\sigma: E \to [q]} \prod_{v \in V} f_v(\sigma \mid_{E(v)}),$$

where

- $G = (V, E)$, and $E(v)$ denotes the incident edges of v and
- $\sigma \mid_{E(v)}$ denotes the restriction of σ to $E(v)$, and $f_v(\sigma \mid_{E(v)})$ is the evaluation of f_v on the ordered input tuple $\sigma \mid_{E(v)}$.

In Volume I of this book we exclusively present the theory over the Boolean domain $q = 2$. We use Holant($\mathcal{F}$) to denote Holant$_2(\mathcal{F})$. We also denote