Msc Thesis

Counting below $\#P$: Classes, problems and descriptive complexity

Angeliki Chalki

$\mu\Pi\lambda\forall$

Graduate Program in Logic, Algorithms and Computation

National and Kapodistrian University of Athens

Supervised by

Aris Pagourtzis

September 2016

Abstract

In this thesis, we study counting classes that lie below $\#P$.

One approach, the most regular in Computational Complexity Theory, is the machine-based approach. Classes like $\#L$, *span-L* [1], and $TotP$, $\#PE$ [38] are defined establishing space and time restrictions on Turing machine's computational resources.

A second approach is Descriptive Complexity's approach. It characterizes complexity classes by the type of logic needed to express the languages in them. Classes deriving from this viewpoint, like $\#\mathcal{FO}$ [44], $\#RH\Pi_1$ [16], $\#R\Sigma_2$ [44], are equivalent to $\#P$, the class of AP-interriducible problems to $\#BIS$, and some subclass of the problems owning an FPRAS respectively.

A great objective of such an investigation is to gain an understanding of how "efficient counting" relates to these already defined classes. By "efficient counting" we mean counting solutions of a problem using a polynomial time algorithm or an FPRAS.

Many other interesting properties of the classes considered and their problems have been examined. For example alternative definitions of counting classes using relation-based operators, and the computational difficulty of complete problems, since complete problems capture the difficulty of the corresponding class. Moreover, in Section 3.5 we define the log-space analog of the class $TotP$ and explore how and to what extent results can be transferred from polynomial time to logarithmic space computation.

Aknowledgments

First and foremost I thank my advisor A. Pagourtzis for the guidance and support throughout this thesis. In addition, I thank particularly Prof. Stathis Zachos for inspiring me through his teaching and advice and Prof. Dimitris Fotakis for the knowledge and inspiration during my studies in MPLA.

I would also like to thank all the members of CoReLab for their influence over my thinking.

Last but not least, I thank my family and friends for their support.

Contents

Chapter 1

Preliminaries

A decision problem can be considered as a language L. In many interesting cases, a relation R_L is associated with the language L, such that given some instance x of the problem the answer is "yes" if and only if $x \in L$ if and only if there is some certificate y for which $R_L(x, y)$.

Definition 1.0.1 *A **deterministic finite automaton** (DFA) M is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, consisting of a finite set of states Q, a finite set Σ of input symbols called the alphabet, a transition function $\delta : Q \times \Sigma \to Q$, an initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$. Let $w = a_1a_2...a_n$ be a string over the alphabet Σ. The automaton M accepts the string w if a sequence of states, $r_0, r_1, ..., r_n$, exists in Q with the following conditions: (1) $r_0 = q_0$, (2) $r_{i+1} = \delta(r_i, a_{i+1})$, for $i = 0, ..., n-1$, (3) $r_n \in F$.*
*A **nondeterministic finite automaton** (NFA) M is a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, where Q, Σ, q_0, F are as before and Δ is a transition function $\Delta : Q \times \Sigma \to P(Q)$ and $P(Q)$ denotes the power set of Q. For each state-symbol combination there may be more than one next steps or none. The automaton M accepts the string $w \in \Sigma^*$ if a sequence of states, $r_0, r_1, ..., r_n$, exists in Q with the following conditions: (1) $r_0 = q_0$, (2) $r_{i+1} = \Delta(r_i, a_{i+1})$, for $i = 0, ..., n-1$, (3) $r_n \in F$.*
A DFA (or an NFA) is considered as a finite directed graph with labels on its edges. This graph is called the state transition graph of the DFA (NFA).

Definition 1.0.2 *A **deterministic Turing machine** (DTM) is a quadruple $M = (K, \Sigma, \delta, s)$, where K is a finite set of states, $s \in K$ is the initial state, Σ is a finite set of symbols (the alphabet of M) and δ is a transition function, which maps $K \times \Sigma$ to $(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. We assume that K, Σ are disjoint sets and that h (the halting state), "yes" (the accepting state), "no" (the rejecting state), and the cursor directions $\leftarrow$, $\rightarrow$, and $-$ are not in $K \cup \Sigma$.*
In a similar way we define the k-tape Turing machine, $k \geqslant 1$, where δ is a program that decides the next state and, for each of the k tapes, the symbol overwritten and the direction of cursor motion by looking at the current state and the current symbol at each tape.
*A **nondeterministic Turing machine** (NTM) is a quadruple $M = (K, \Sigma, \Delta, s)$, where K, Σ, s are as before and Δ is a relation: $\Delta \subset (K \times \Sigma) \times [((K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow$*

$, -\}]$. *That is, for each state-symbol combination, there may be more than one appropriate next steps.*

A finitely long string $x \in \Sigma^*$ is required as input for a computation of a Turing machine. A **transducer** is a Turing machine with output. At the end of the computation the contents of an extra tape consist the output of the machine on a given input x. The output is valid only if the machine stops in an accepting state. Moreover, the input tape is read-only (the contents of the input tape cannot be overwritten) and the output tape is write-only (the cursor never moves to the left). If in addition we assume that the input cursor only moves to the right, the Turing machine is called one-way.
In many cases we describe the operation of a Turing machine using the notion of a configuration. Intuitively, a configuration contains a complete description of the current state of the computation (the state $q \in K$, the symbol scanned by the cursor and the contents of all tapes).
The space required by M on input x is the total number of work tape cells occupied at the end of the computation (we do not count the space used on the input or output tapes). We say that a Turing machine M operates within space bound $f(n)$, if for any input x, M requires space at most $f(|x|)$. We say that a language L is in the class $\mathbf{SPACE}(f(n))$ (resp. $\mathbf{NSPACE}(f(n))$) if there is a (nondeterministic) Turing machine with input and output that decides L and operates within space bound $f(n)$.

Definition 1.0.3 *(i) The complexity class L is the class containing decision problems that can be solved by a deterministic Turing machine using a logarithmic amount of space, i.e.* $L = \mathbf{SPACE}(logn)$.
(ii) $NL = \mathbf{NSPACE}(logn)$.
(iii) The class $1NL$ is the class of languages that can be decided by one-way nondeterministic log-space bounded Turing machines.

The complexity class $\mathbf{TIME}(f(n))$ ($\mathbf{NTIME}(f(n))$) is the set of languages that can be decided by some (nondeterministic) Turing machine within time bound $f(n)$.

Definition 1.0.4 *(i) The complexity class P is the class* $\bigcup_{k\in\mathbb{N}} \mathbf{TIME}(n^k)$.
(ii) $NP = \bigcup_{k\in\mathbb{N}} \mathbf{NTIME}(n^k)$. *Alternatively, a language L is in NP if and only if there is a polynomially decidable relation R and a polynomial p, such that* $L = \{x : \text{there is some } y, |y| \leqslant p(|x|) \text{ and, } (x, y) \in R\}$.

Definition 1.0.5 *A nondeterministic Turing machine is called **unambiguous** if it has the following property: For any input x there is at most one accepting computation.*
UP is the class of languages that can be decided by unambiguous polynomial-time bounded Turing machines.
UL is the class of languages that can be decided by unambiguous log-space bounded Turing machines.

$FewP$ and $FewL$ are the classes of languages accepted by NP, NL Turing machines respectively, having polynomial number of accepting paths.

Definition 1.0.6 *For any language L, we define $\bar{L} = \Sigma^* \setminus L$ and for any class $\mathcal{C}$ of languages, we define co-$\mathcal{C} = \{\bar{L} : L \in \mathcal{C}\}$.*

The previous classes contain languages, i.e. decision problems. At this point, we introduce classes that their members are functions. The function problem associated with the language L, denoted FL, is the following computational problem: Given x, find a string y such that $R_L(x, y)$ if such a string exists, otherwise return "no".

Definition 1.0.7 *The class of all function problems associated with languages in NP is called FNP. FP is the subclass of FNP containing only the problems that can be solved in polynomial time.*
FL is the class of function problems associated with languages in NL that can be solved in logarithmic space.

Definition 1.0.8 *The counting problem associated with the language L, denoted $\#L$, is the following: Given x, how many y are there such that $(x, y) \in R_L$?*
$\#P$ is the class of counting problems associated with languages in NP.
Alternatively, for a machine M, let the function $acc_M : \{0,1\}^* \to \mathbb{N}$ be defined so that $acc_M(x)$ is the number of accepting computations of M on x. Define $\#P := \{f \mid f = acc_M$ for some NP-machine $M\}$.

Definition 1.0.9 *For a transducer M, let the function $span_M : \{0,1\}^* \to \mathbb{N}$ be defined so that $span_M(x)$ is the number of different valid outputs of M on x. Define span-$P := \{f \mid f = span_M$ for some NP-transducer $M\}$.*

Definition 1.0.10 *A language L is in the class PP if there is a nondeterministic polynomial-time bounded Turing machine M such that, for all inputs x, $x \in L$ if and only if more than half of the computations of M on input x end up accepting. We say that M decides L "by majority".*
A language L is in the class PL if there is a nondeterministic log-space bounded Turing machine M such that, for all inputs x, $x \in L$ if and only if more than half of the computations of M on input x end up accepting.

Proposition 1.0.1 *(i) $NP \subseteq PP$.*
(ii) $NL \subseteq PL$

Definition 1.0.11 ***Log-space reductions between functions***:

- *Log-space functional many-one reduction (log-space Karp) $f \leqslant_m^l g$: $\exists h \in FL$, $\forall x$ $f(x) = g(h(x))$.*

- *(Nonadaptive) Log-space functional Turing reduction (log-space Cook) $f \leqslant^l_T g$: Such a reduction is computed by a deterministic log-space transducer that asks queries to the oracle function g. The L-transducer has an additional unbounded oracle tape on which a query can be written one-way, the query-answer can be read two-way and each query-answer is erased before a new query is being constructed.*

- *Log-space metric reduction (log-space Cook[1]) $f \leqslant^l_{1-T} g$: It is a log-space functional Turing reduction in which the L-transducer may ask at most one query to the oracle function g.*

For simplicity we use the notions "Karp", "parsimonious", "Cook", "Cook[1]" reductions and we assume that the reader is familiar with these types of reductions.

Definition 1.0.12 *For function classes $\mathcal{C}$ and $\mathcal{C}'$, define*
$\mathcal{C} - \mathcal{C}' = \{h \mid there\ exist\ functions\ f \in \mathcal{C}\ and\ f' \in \mathcal{C}'\ such\ that\ h = f - f'\}$.

Definition 1.0.13 *For every $n \in \mathbb{N}$, a n-input, single-output **Boolean circuit** C is a directed acyclic graph with n sources (vertices with no incoming edges) and one sink (vertex with no outgoing edges). All nonsource vertices are called gates and are labelled with one of $\vee$, $\wedge$ or $\neg$ (i.e. the logical operations OR, AND, and NOT).*
*The vertices labelled with $\vee$ and $\wedge$ have fan-in (i.e., number of incoming edges) equal to 2 and the vertices labelled with $\neg$ have fan-in 1. The **size** of C, denoted as $|C|$, is the number of vertices in it. The **depth** of a circuit is the length of the longest directed path from an input node to the output node.*
If C is a Boolean circuit, and $x \in \{0,1\}^n$ is some input, then the output of C on x, denoted by $C(x)$, is defined in the natural way. More formally, for every vertex v of C, we give it a value $val(v)$ as follows: If v is the i^{th} input vertex then $val(v) = x_i$ and otherwise $val(v)$ is defined recursively by applying vs logical operation on the values of the vertices connected to v. The output $C(x)$ is the value of the output vertex.

Definition 1.0.14 *Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A **T(n)-size circuit family** is a sequence $\{C_n\}_{n\in\mathbb{N}}$ of Boolean circuits, where C_n has n inputs and a single output, and its size $|C_n| \leqslant T(n)$ for every n. We say that a language L is in $\mathbf{SIZE}(\mathbf{T}(\mathbf{n}))$ if there exists a $T(n)$-size circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that for every $x \in \{0,1\}^n$, $x \in L \Leftrightarrow C_n(x) = 1$.*

Definition 1.0.15 *A Boolean circuit is **monotone** if it contains only AND and OR gates, and no NOT gates. Such a circuit can only compute monotone functions, defined as follows. For $x, y \in \{0,1\}^n$, we denote $x \preceq y$ if every bit that is 1 in x is also 1 in y. A **function** $f : \{0,1\}^n \to \{0,1\}$ is **monotone** if $f(x) \leqslant f(y)$ for every $x \preceq y$.*

Definition 1.0.16 *Let C be a Boolean circuit. An AND gate v in C is said to be a **skew gate** if it has at most one input that is not an input of C. The input of v that is not an input to C is called a non-skew input of v. The circuit C is said to be a **skew circuit** if all AND gates in it are skew gates. A family $\{C_n\}$ of Boolean circuits is said to be a skew circuit family if all its members are skew circuits.*

Definition 1.0.17 *A circuit family $\{C_n\}$ is **P-uniform** if there is a polynomial-time Turing machine that on input 1^n outputs the description of the circuit C_n.*
*A circuit family $\{C_n\}$ is **logspace-uniform** if there is an implicitly logspace computable function mapping 1^n to the description of the circuit C_n.*

Definition 1.0.18 *(i) For every d, a language L is in NC^d if L can be decided by a family of circuits $\{C_n\}$ where C_n has poly(n) size, depth $\mathcal{O}(log^d n)$ and bounded fan-in. The class NC is $\bigcup_{i\geqslant 1} NC^i$.*
(ii) For every d, a language L is in AC^d if L can be decided by a family of circuits $\{C_n\}$ where C_n has poly(n) size, depth $\mathcal{O}(log^d n)$ and unbounded fan-in. The class AC is $\bigcup_{i\geqslant 1} AC^i$.

Proposition 1.0.2 *For every $i \in \mathbb{N}$, $NC^i \subseteq AC^i \subseteq NC^{i+1}$.*

Arithmetic circuits are defined just as Boolean circuits, except that the gates compute the sum and the product of their inputs instead of computing the OR and AND functions. We denote a gate computing the sum (product) of its inputs as a PLUS (MULT resp.) gate.

Definition 1.0.19 *We define the **degree** of an arithmetic circuit to be the algebraic degree of the polynomial computed by the circuit.*
Thus, in a Boolean circuit the constants have degree zero, the circuit inputs have degree one, the degree of an OR vertex is the maximum of the degrees of its inputs, and the degree of an AND vertex is the sum of the degrees of its inputs.
Analogously, in an arithmetic circuit the constants have degree zero, the circuit inputs have degree one, the degree of a PLUS vertex is the maximum of the degrees of its inputs, and the degree of a MULT vertex is the sum of the degrees of its inputs.

Chapter 2

Log-space counting classes

2.1 The classes #L and span-L

In this section, we introduce the log-space analogs of the classes #P and span-P. These classes were introduced by Àlvarez and Jenner. The results of this section can be found in [1].
Functions in #L count the number of accepting computations of a nondeterministic log-space-bounded Turing machine, whereas functions in span-L count the number of different output values of such a machine with additional output tape.

Definition 2.1.1 *Define* $\#L := \{f \mid f = acc_M \text{ } for\ some\ NL\text{-}machine\ M\}$.

Definition 2.1.2 *Define* $span\text{-}L := \{f \mid f = span_M \text{ } for\ some\ NL\text{-}transducer\ M\}$.

It is easily verified that $FP \subseteq \#P$. Moreover, in [31] it was shown that $\#P \subseteq span\text{-}P$. The corresponding statements hold for log-space classes.

Proposition 2.1.1 $FL \subseteq \#L \subseteq span\text{-}L$.

Proof. The first inclusion can be proved easily.
For the second inclusion, let $f \in \#L$ and M the NL-machine witnessing f. We can construct an NL-transducer M' that simulates M, accepts iff M accepts and outputs the computation path. Then $f = acc_M = span_{M'}$, and hence $f \in span\text{-}L$. ■

We present some complete functions for the classes $\#L$ and $span\text{-}L$, which are counting versions of NL-complete automata and graph problems.

The ranking function for DFA, $\#DFA$:
Input: An encoding of a DFA M and a string $x \in \{0,1\}^*$.
Output: The number of words lexicographically smaller than or equal to x accepted by M.

The ranking function for NFA, $\#NFA$:
Input: An encoding of a NFA M and a string $x \in \{0,1\}^*$.
Output: The number of words lexicographically smaller than or equal to x accepted by M.

Comment 2.1.1 *The decision version of these two problems is the nonemptiness problem for M, where M is a DFA or an NFA respectively. This problem is NL-complete with respect to log-space many-one reductions [27].*

Theorem 2.1.1 *(i) $\#DFA$ is log-space many-one complete for $\#L$.*
(ii) $\#NFA$ is log-space many-one complete for span-L.

Proof. (i) We can construct an NL-machine N, which on input a DFA M and $x \in \{0,1\}^*$ guesses a string $y \leqslant x$ bit-by-bit. For every new bit, N records the corresponding state of M and accepts if a final state is ever reached. Since M is a DFA, there is exactly one computation path of N for any guessed word y. Furthermore, the number of accepting paths of N corresponds to the number of words smaller or equal to x accepted by M. So, $\#DFA \in \#L$.
For the hardness property, let $f \in \#L$ and N the NL-machine witnessing f. Let $p(|x|)$ be the polynomial that bounds the running time of N on input x. There exists a function $h \in FL$ such that $f(x) = \#DFA(h(x))$. Define $h(x) := (< N_x >, 1^{p(|x|)})$, where $< N_x >$ denotes the encoding of a state transition graph of a DFA constructed as follows. Let $C_{(N,x)}$ denote the set of all configurations of N on input x, $c_{(start,x)}$ the start configuration, $C_{(acc,x)}$ the set of all accepting configurations, and sink denote an element not contained in $C_{(N,x)}$. Then, $N_x := (C_{(N,x)} \cup \{sink\}, C_{(N,x)} \cup \{sink\}, \delta, c_{(start,x)}, C_{(acc,x)})$, where for all $c_i, c_j \in C_{(N,x)} \cup \{sink\}$:

$$\delta(c_i, \beta) := \begin{cases} c_j & \text{if } c_i \text{ reaches } c_j \text{ in one step in a computation of N on x,} \\ sink & \text{otherwise.} \end{cases}$$

The automaton N_x is a DFA and can be constructed in logarithmic space. Furthermore, the number of accepting computation paths of N on input x equals the number of words accepted by N_x of length at most $p(|x|)$. The above construction can be made such that the alphabet of N_x is $\{0,1\}$, and then $f(x)$ equals the words accepted by N_x lexicographically smaller than or equal to $1^{q(|x|)}$, for some polynomial q.
(ii) To prove $\#NFA \in$ *span-L*, construct an NL-transducer N, which on input an NFA M and $x \in \{0,1\}^*$ does the following. N guesses and outputs a word $y \leqslant x$ bit-by-bit, guessing and recording a new state of M consistent with each guessed bit and the transition table of M. N accepts if an accepting state of M is ever reached. Since M is an NFA, a word y can cause different sequences of transitions and there may be more than one valid computation of N with the same output. But the number of different valid outputs of N, $span_N(< M, x >)$, corresponds to the number of words smaller or equal to x accepted by M.
To see that $\#NFA$ is span-L hard, consider a function $f \in$ *span-L* and the NL-transducer N that witnesses f via its span. There exists a function $h \in FL$ such that $f(x) = \#NFA(h(x))$.

Define $h(x) := (< N_x >, 1^{p(|x|)})$, where $N_x := (C_{(N,x)} \cup \{sink\}, \{0,1\}, \delta, c_{(start,x)}, c_{(acc,x)})$, where the notation is the same as in (i). For all $c_i, c_j \in C_{(N,x)}$ suct that c_i reaches c_j in one step in a computation of N on x, and $\beta \in \{0, 1, \lambda\}$ we define:
$\delta(c_i, \beta) := c_j$ if $\beta \in \{0, 1\}$ and the output in c_j is β or
$\beta = \lambda$ and in c_j no output occurs.
The number of different words smaller or equal to $1^{p(|x|)}$ that are accepted by N_x is exactly the number of different valid outputs the transducer N can produce on input x. ■

As a consequence of the previous proof, a restrictive version of $\#NFA$, the census function, is also *span-L*-complete.

The census function for NFA, $\#NFA_{\leqslant n}$:
Input: An encoding of a NFA M and 1^n.
Output: The number of words of length up to n accepted by M.

Another two complete functions for $\#L$ and *span-L* can be obtained by defining counting versions of the graph problem, REACHABILITY, which is NL-complete.

$\#Path$:
Input: A directed graph $G = (V, E)$ with vertex set $V = \{1, 2, ..., n\}$.
Output: The number of different paths of length at most n from vertex 1 to vertex n.

$\#SpecialPath$:
Input: A directed labelled graph $G = (V, E)$ with vertex set $V = \{1, 2, ..., n\}$ and edges labelled over L, and $L' \subseteq L$.
Output: The number of lexicographically different paths of length at most n from vertex 1 to vertex n, with labels in L' deleted.

Another way to define two graph problems complete for $\#L$ and *span-L* would be to count (simple) paths, allowing only acyclic graphs as inputs.

Corollary 2.1.1 *(i) $\#Path$ is log-space many-one complete for $\#L$.*
(ii) $\#SpecialPath$ is log-space many-one complete for span-L.

Proof. (i) Let $G = (V, E)$ and M the NL-machine that solves REACHABILITY on input G: Machine M guesses a path starting from vertex 1 of length at most n in logarithmic space. A computation path is accepting iff the path ends at vertex n, and rejecting otherwise. Then, the number of M's accepting computation paths is equal to $\#Path(G)$, hence $\#Path(G) \in \#L$.
For the hardness property, consider $f \in \#L$, and N the NL-machine witnessing f. Construct in logarithmic space the configuration graph G_N of N. The number of accepting computation paths of N is equal to $\#Path(G_N)$.

(ii) Let G=(V, E) and M the NL-transducer that does the following. M solves REACHABILITY on input G, but it outputs only the guessed edges labelled over $L \setminus L'$. Hence, $\#SpecialPath \in$ *span-L*.
Moreover, $\#SpecialPath$ is *span-L*-hard. Let the function f that, given the encoding of an NFA M and a string 1^n, returns the number of words lexicographically smaller than 1^n accepted by M. Convert the state transition graph of M to a graph G_M with n states. The edges of this graph are labelled over $L = \{0, 1, \lambda\}$. Define L' to be $\{\lambda\}$. Then,
$f(< M >, 1^n) = \#SpecialPath(G_M)$. Since, the first problem is *span-L*-hard and the new graph, G_M, can be constructed easily in logarithmic space, the proof is complete. ■

Comment 2.1.2 *The problem $\#Path$ can be defined as the problem of computing the number of (simple) paths from vertex 1 to vertex n in a directed acyclic graph (DAG). This problem is also $\#L$ complete.*
Contrary to the previous versions of counting paths in a graph, the problem of counting simple paths of a particular length in a (general) directed graph is $\#P$ complete.

We just showed membership in *span-L* for the counting versions of an automata and a graph problem. The next section present other problems that belong to *span-L*.

2.2 Problems belonging to span-L

Consider the functions

$\#3SAT$:
Input: A boolean formula ϕ in 3-conjunctive normal form.
Output: The number of satisfying assignments of ϕ.

$\#UN3SAT$:
Input: A boolean formula ϕ in 3-conjunctive normal form.
Output: The number of nonsatisfying assignments of ϕ.

Whereas the first function is $\#P$ complete with respect to log-space many-one reductions, we show that the second function $\#UN3SAT$ is in *span-L*. First, we give the following lemma which is proved using the same idea. Consider the set

$EVAL3SAT := \{\phi; b_1 b_2 ... b_n | b_i \in \{0, 1\}, \phi$ boolean formula in 3-conjunctive normal form, and $b_1 ... b_n$ is a satisfying assignment for $\phi\}$.

Lemma 2.2.1 *(i) $\overline{EVAL3SAT} \in 1NL$.*
(ii) $EVAL3SAT \in co\text{-}1NL$.
(iii) $EVAL3SAT \in L$, and hence $EVAL3SAT \in NL$.

Proof. (i) Let M be an 1NL-machine that on input $(\tilde{x}_{11}, \tilde{x}_{12}, \tilde{x}_{13}), ..., (\tilde{x}_{m1}, \tilde{x}_{m2}, \tilde{x}_{m3}); b_1 b_2 ... b_n$ guesses a clause. The head of the input tape is moved to that clause and its three literals $\tilde{x}_{i1}, \tilde{x}_{i2}, \tilde{x}_{i3}$ are stored on the working tape. The head keeps moving until the relevant bits of the assignment are reached and then checks whether the clause is unsatisfied. Similarly, that the input is not well-formed can be easily guessed and verified. M accepts if the clause is unsatisfied.

(ii) It follows from (i).

(iii) It can be easily proved. ■

Proposition 2.2.1 *$\#UN3SAT \in$ span-L.*

Proof. We construct an NL-transducer M such that on input a formula ϕ in 3-conjunctive normal form, $span_M(\phi)$ equals the number of nonsatisfying assignments of ϕ.

On input x, M first checks whether x is of the required form. If this is not the case M rejects and $span_M(x) = 0$. If x is a well-formed ϕ, M counts the number of different variables in ϕ, guesses a clause and stores its three literals $\tilde{x}_{i1}, \tilde{x}_{i2}, \tilde{x}_{i3}$ on its working tape. Then M guesses bit-by-bit an assignment of appropriate length, writing every guessed bit on the output tape. At the same time, when the relevant bits of the assignment are reached, M checks if they falsify the clause $\tilde{x}_{i1} \vee \tilde{x}_{i2} \vee \tilde{x}_{i3}$. If this is the case, then M accepts and outputs the guessed truth assignment for the formula ϕ. ■

$\#DNF$:
Input: A boolean formula F in DNF.
Output: The number of satisfying truth assignments for F.

Theorem 2.2.1 *$\#DNF \in$ span-L.*

Proof. A DNF formula is a disjunction of terms, each of which is a conjunction of literals of the form x or $\overline{x}$. Our goal is to estimate the number of assignments to the variables that satisfy this formula.

The algorithm proving that $\#DNF$ is in *span-L* is simple and clear: Let M be an NL-transducer, $D_1, ...D_m$ the terms of the input F and $x_1, ..., x_n$ the variables of F. On input F in DNF, M guesses a term D_i nondeterministically. Then, for every variable x_j, $1 \leqslant j \leqslant n$, M outputs a truth value for x_j in the following way: For j running from 1 to n, M outputs "1", if D_i contains x_j, outputs "0", if D_i contains $\overline{x_j}$, outputs nondeterministically "1" ("TRUE ") or "0" ("FALSE"), if x_j and its negation $\overline{x_j}$ don't appear in D_i, and stops rejecting if both x_j and $\overline{x_j}$ appear in D_i. The checking of $x_j \in D_i$ (and $\overline{x_j} \in D_i$) can be done by moving a cursor, so in logarithmic space. Moreover, different paths of this transducer lead to the same truth assignment, but the number of distinct outputs is equal to the number of truth assignments for F. ■

The next problem is the problem of counting the different nodes appearing in a path from node 1 to node n of a directed graph G.

$\#PATH\_NODES$:
Input: A directed graph $G = (V, E)$ with vertex set $V = \{1, 2, ..., n\}$.
Output: The number of nodes v, such that there is a path P from node 1 to n and $v \in P$.

Theorem 2.2.2 *$\#PATH\_NODES \in$ span-L.*

Proof. Given a graph G, an NL transducer M guesses a vertex v and a path from vertex 1 to vertex n. If v appears in the path, then M outputs vertex v. The computation can be done in logarithmic space since M deletes the current node of the path before guessing the next node. M stores on its working tape only the initial guessed vertex v. Some vertex can be seen on the output many times, but the number of different outputs is the number we would like to compute. ■

$\#CYCLE\_NODES$:
Input: A directed path $G = (V, E)$ with vertex set $V = \{1, 2, ..., n\}$.
Output: The number of nodes v, such that there is a cycle C in G and $v \in C$.

Theorem 2.2.3 *$\#CYCLE\_NODES \in$ span-L.*

Proof. The proof is very similar to the previous proof. A node v is guessed and a cycle starting from v. If such a cycle is found then v is written on the output. ■

2.3 #L is easy, but span-L is hard

With the theorems of this section, we show that the class $\#L$ is included in FP, but, surprisingly, that *span-L* is a hard log-space counting class. More precisely, if *span-L* $\subseteq FP$, then the polynomial time hierarchy collapses to P.

In [1], it was shown that $\#L \subseteq NC^2$, and since $NC^2 \subseteq FP$, $\#L \subseteq FP$. Here, we prove that the number of accepting paths of an NL-machine can be computed in polynomial time.

Theorem 2.3.1 *$\#L \subseteq FP$.*

Proof. To prove the theorem, we show that $\#Path$ can be computed by a deterministic Turing machine M in polynomial time.

Given a directed graph G(V, E) with vertex set $V = \{1, 2, ..., n\}$, M performs a procedure in n steps. In fact, M constructs a $n \times n$ matrix, in which the $(i, j)^{th}$ element contains the number of paths from vertex 1 to vertex i of length at most j. In the j-th step, $j = 0, 1, ..., n - 1$, M saves, for each vertex i, $i \in \{1, ..., n\}$, the number of paths of length at

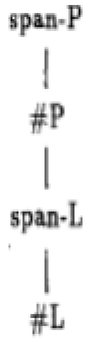

Figure 2.1: Inclusion relations among log-space and poly-time counting classes

most j from vertex 1 to vertex i. In order M to compute the paths of length at most j from 1 to some vertex k, it finds the vertices that have edges going to k and sums the number of paths of length at most $j-1$ from 1 to them. These numbers have already been saved in $j-1$-th step. At the end, M returns the content of the $(n,n)^{th}$ element, i.e. the number of paths from 1 to vertex n of length at most n. Obviously, this algorithm needs polynomial time.

Since FP is closed under log-space functional many-one reductions, $\#L \subseteq FP$. ■

Some interesting observations about $\#L$, *span-L* and the class FL^{NL} are proved in [1].

Proposition 2.3.1 *(i)* $\#L \subseteq NC^2$.
(ii) $FL^{NL} \subseteq NC^2$.
(ii) $FL^{NL} \subseteq \#L$ *if and only if* $NL = UL$.
(iii) $FL^{NL} \subseteq$ *span-L.*

In the following we will show that the complexity of *span-L* is closely tied to the complexity of $\#P$. The two classes are log-space metric reducible to each other, i.e. $FL_1(\#P) = FL_1(\textit{span-L})$.

Theorem 2.3.2 *span-L* $\subseteq \#P$.

Proof. Let f be a function in *span-L* such that on input x, $f(x) = \textit{span}_N(x)$, for some NL-transducer N. Any output value y of N, on input x, satisfies $|y| \leqslant p(|x|)$, where p is a polynomial. Obviously, the set $A := \{x;y|x,y \in \{0,1\}^*, y \text{ is a valid output of N on } x\}$ is in NL. Since $NL \subseteq P$, there is a P-algorithm for deciding membership in A. Construct an NP-machine M which on input x guesses a string y, $|y| \leqslant p(|x|)$, and verifies that $x;y \in A$ executing the P-algorithm for A on $x;y$. Then M has exactly one accepting computation for each valid output y of N. Thus, $f \in \#P$. ■

In the rest of this section, we present results we need to prove that $\#P$ is log-space metric reducible to *span-L*.

Theorem 2.3.3 *(Valiant [49])*
$\#3SAT$ is complete for $\#P$ with respect to log-space many-one reductions.

Theorem 2.3.4 *$\#P \subseteq FL_1(span\text{-}L)$.*

Proof. Since $\#3SAT$ is complete for $\#P$ with respect to log-space many-one reductions, it suffices to show that $\#3SAT \in FL_1(span\text{-}L)$.

We will show that $\#3SAT$ can be computed by a deterministic L-transducer M that asks one query to $\#UN3SAT$. This yields $\#3SAT \in FL_1(span\text{-}L)$. On input x, M checks that x is a formula of the required form. If this is not the case, M outputs 0. If x is a formula ϕ in 3-conjunctive normal form, M copies ϕ on the oracle tape, queries its oracle for $\#UN3SAT(\phi)$ and has two-way access to $\#UN3SAT(\phi)$ on its oracle tape. M now computes the difference $2^n - \#UN3SAT(\phi)$, where n denotes the number of different variables in ϕ and outputs this value. Clearly, $\#3SAT = 2^n - \#UN3SAT(\phi)$.

This difference can be computed in logarithmic space (since subtraction can be done in deterministic logspace), but the value 2^n must be stored in $\mathcal{O}(logn)$ space, and this can be done by storing just n in $\mathcal{O}(logn)$ space. ■

Corollary 2.3.1 *$FL_1(span\text{-}L) = FL_1(\#P)$, i.e. every function in $\#P$ is metric reducible to a function in span-L and vice versa.*

This means that *span-L* and $\#P$ share the same complete functions with respect to metric reducibility. Thus, we can add the functions $\#NFA$ and $\#SpecialPath$ to the list of functions complete for $\#P$ with respect to log-space metric reducibility. We mention here again that these two functions are counting versions of NL-complete problems.

Let us define the "complements" $\#co\text{-}DFA$ and $\#co\text{-}NFA$ of the already introduced functions $\#DFA$ and $\#NFA$:

#co-DFA:
Input: An encoding of a DFA M and a string 1^n.
Output: The number of words of length up to n that are not accepted by M.

#co-NFA:
Input: An encoding of an NFA M and a string 1^n.
Output: The number of words of length up to n that are not accepted by M.

Following the proof of Theorem 2.1.1 (i) and replacing rejecting with accepting states of N_x and vice versa, we can easily prove that $\#co\text{-}DFA$ is $\#L$-complete. On the contrary, $\#co\text{-}NFA$ is not *span-L*-complete with respect to log-space many-one reducibility unless $NL = P = NP$.

Theorem 2.3.5 *$\#co\text{-}NFA$ is log-space many-one complete for $\#P$.*

Although *span-L* and $\#P$ are very similar in computation power, they do not seem to be the same class. The inclusion $\#P \subseteq$ *span-L* would imply that there exists an NL-transducer M witnessing the $\#P$-complete function $\#3SAT$. Then, by simulating M and checking that M has a valid output on input ϕ, we could decide $3SAT$ with an NL computation.

Proposition 2.3.2 *If span-L* $= \#P$, *then NL=P=NP.*

Combining $PH \subseteq P^{PP}$, which was proved by Toda in [46], $P^{\#P} = P^{PP}$ [6] and Corollary 2.3.1, we have the following.

Corollary 2.3.2 $PH \subseteq P^{span-L} = P^{\#P} = P^{PP}$.

Furthermore, since $P^{NP} \subseteq P^{\#P}$, Corollary 2.3.1 implies that *span-L* is Turing hard for $\Delta_2^p = P^{NP}$, the second deterministic level of the polynomial-time hierarchy.

It is also unlikely that every function in *span-L* can be computed in polynomial time.

Corollary 2.3.3 *span-L* $\subseteq FP$ *if and only if* $P = NP = PH = P^{\#P}$.

Chapter 3

Properties of counting classes below #P

3.1 #P, span-P and the operator $\#\cdot$

We would like to study some properties of counting classes. For this purpose, we will need the machine-based definition of the class $\#\mathcal{C}$ by Valiant [50] and the predicate-based definition of the operator $\#\cdot$ by Toda [47]. This line was followed in many papes later, like [51].

An excellent review of such "definition adventures" can be found in [24].

Definition 3.1.1 *For any class $\mathcal{C}$, define $\#\mathcal{C} = \bigcup_{A\in\mathcal{C}}(\#P)^A$, where by $(\#P)^A$, we mean the functions counting the accepting paths of nondeterministic polynomial-time Turing machines having A as their oracle.*

Definition 3.1.2 *For any class $\mathcal{C}$, define $\# \cdot \mathcal{C}$ to be the class of (total) functions f, such that for some $\mathcal{C}$-computable, two-argument predicate $\mathcal{R}$ and some polynomial p, for every string x it holds that: $f(x) = |\{y \mid |y| = p(|x|) \text{ and } \mathcal{R}(x,y)\}|$.*

Comment 3.1.1 *(i) Because of the characterization of the class NP via P predicates, it holds that $\#P = \# \cdot P$.*

(ii) When using an oracle, it does not matter whether the oracle or its complement is used, so $\#NP = \#coNP$.

(iii) The class $\# \cdot NP$ is the class of functions "counting" the number of appropriate-length second arguments that cause some predicate, computable by an NP machine, to be true for the given first argument. It is easy to prove that this class is exactly the class of functions "counting" the number of accepting paths of an NP machine that for every input and every possible computation path asks one question to an NP oracle at the end of the path and returns the oracle's answer.

We give here a sketch of the proof: *A function which counts the y's that make true an NP-computable predicate $\mathcal{R}$, can be witnessed by an NP machine that guesses a y of appropriate length and asks the question "is $\mathcal{R}(x,y)$ true?" to the NP oracle for $\mathcal{R}$.*

Conversely, a function which "counts" the accepting paths of an NP machine M with a final NP oracle call, can be viewed as a function that "counts" the y's of the NP-computable predicate $\mathcal{R} = \{(x, y) \mid y \text{ is an accepting path of } M \text{ on input } x\}$*. The predicate* $\mathcal{R}$ *can be computed by the NP machine* M'*, which on input* (x, y) *simulates the M's path y on input x and at the end instead of calling the oracle for an NP problem,* M' *simulates the NP machine that solves this problem. Clearly,* $\mathcal{R}(x, y)$ *is true iff* M' *has at least one accepting path on input* (x, y)*.*

(iv) Similarly, $\# \cdot coNP$ *is the class of functions "counting" the number of accepting paths of an NP machine which can ask only one question to a coNP oracle at the end of any computation path and returns the oracle's answer.*

(v) It is not hard to see that the class that "counts" the accepting paths of NP^{NP} *machines is* $\# \cdot P^{NP}$*. Thus,* $\#NP = \# \cdot P^{NP}$*.*

We continue with some results that demonstrate the connection between classes defined with the machine-based approach and classes defined with the operator $\#\cdot$.

Theorem 3.1.1 *(i)* $\#P = \# \cdot NP \iff UP = NP$*.*
(ii) $\# \cdot coNP = \# \cdot P^{NP} = \#NP$*.*
(iii) $\# \cdot NP = \# \cdot coNP \iff NP = UP^{NP} \iff NP = coNP$*.*
(iv) $P^{\# \cdot NP} = P^{\# \cdot coNP}$*.*

Proof. (i) ($\Longrightarrow$) It is immediate from the definitions of the classes UP and NP that $UP \subseteq NP$. Let $A \in NP$. We show that its characteristic function belongs to $\# \cdot NP = \#P$. This means that there is an NP machine which on input x has one accepting path if $x \in A$ and zero accepting paths if $x \notin A$. Equivalently, $A \in UP$.

Since $A \in NP$, there is an NP machine M for A. There is also a polynomial p that gives the length of the computation paths of M. We define the predicate $\mathcal{R}$ such that $\mathcal{R}(x, 0^{p(|x|)})$ is true if M has at least one accepting path on input x, false if M has no accepting path on input x and $\mathcal{R}(x, y)$, $y \neq 0^{p(|x|)}$, is always false. The predicate $\mathcal{R}$ is NP-computable. Let the NP machine M' which on input $(x, 0^{p(|x|)})$ simulates M on input x, whereas on input (x, y), $y \neq 0^{p(|x|)}$, rejects the input. Obviously M' computes $\mathcal{R}$.

So $f(x) = |\{y \mid p(|x|) = |y| \text{ and } \mathcal{R}(x, y)\}|$ is the characteristic function of A and belongs to $\# \cdot NP$.

($\Longleftarrow$) From the definitions 3.1.1 and 3.1.2 we have that $\#P \subseteq \# \cdot NP$.
Let $f \in \# \cdot NP$ and $f(x) = |\{y \mid p(|x|) = |y| \text{ and } \mathcal{R}(x, y)\}|$, for some $\mathcal{R}$ and p. There is an NP machine M that computes $\mathcal{R}$. Because of our assumption that $UP = NP$, we can have a UP machine, M', for $\mathcal{R}$. This means that on input (x, y), M' has exactly one accepting path if $\mathcal{R}(x, y)$ is true, and no accepting path if $\neg\mathcal{R}(x, y)$ is true. Define the NP machine N, which on input x guesses a string y, $|y| = p(|x|)$ and simulates M' on input (x, y). Then the accepting paths of N are equal to the number of y's that make $\mathcal{R}(x, y)$ true.

So $f \in \#P$.

(ii) We use here the characterization of $\# \cdot coNP$ mentioned in the comment 3.1.1 (iv). It holds that $\# \cdot coNP \subseteq \#NP$. A function witnessed by an NP machine M which at the end asks one question of the form "$x \in A$?" to a coNP oracle, can be witnessed by an NP machine M' which at the end asks the question "$x \in \overline{A}$?" to an NP oracle and returns the opposite of the oracle's reply. Obviously, M and M' are the same except for their oracles.

Conversely, $\#NP \subseteq \# \cdot coNP$. Let M the NP^{NP} machine which witnesses a function $f \in \#NP$ asking questions to an NP oracle A. We describe the construction of the NP machine M', which has the same number of accepting paths as M, asking only one coNP oracle question at the end, and returning the oracle's answer.

The machine M' simulates M on the same input x. When M makes an oracle call, M' guesses the answer of the oracle and continues the simulation of M with this oracle answer. At the end, M' remembers the "yes" guesses and the "no" guesses.

We note here that there is an NP machine N_1 that on input $< x_1, x_2, ..., x_n >$ accepts iff $x_1 \in A \wedge x_2 \in A \wedge ... \wedge x_n \in A$ and a coNP machine N_2 that on input $< y_1, y_2, ..., y_m >$ accepts iff $y_1 \in \overline{A} \wedge ... \wedge y_m \in \overline{A}$. Here every "$x_i \in A$?", $1 \leqslant i \leqslant n$, symbolizes an NP oracle call that was answered positively by M''s guess and every "$y_i \in A$?", $1 \leqslant i \leqslant m$, symbolizes an NP oracle call that was answered negatively by M''s guess.

1. If M with M''s guesses for the oracle answers rejects the input, then M' makes a trivial question "$x \in B$?" to a coNP oracle, where $x \notin B$. The machine M' returns the answer of the coNP oracle and rejects the input.

2. If M with M''s guesses for the oracle answers accepts the input, then M' accepts the input only if its guesses are correct. For this, M' does the following:

   - M' checks if its "yes" guesses are correct. M' simulates a computation path p of N_1 on input $< x_1, x_2, ..., x_n >$. If N_1 rejects, then M' halts and rejects. But if N_1 accepts, then M' continues the computation and checks that p is the lexicographically smallest accepting path of N_1 on input $< x_1, x_2, ..., x_n >$. This checking can be done with a coNP oracle call and we include it in the next step.

     The set $S = \{(x, p) \mid$ *p is the lexicographically smallest accepting path of* N_1 *on input* $x\}$ belongs to coNP, since there is a coNP machine N for S. N guesses a path p$'$ of N_1, simulates N_1's computation on this path and accepts iff p$\leqslant$ p$'$ or p$'$ is a rejecting path.

   - M' checks if its "no" guesses are correct.
     M' poses the question "$y_1 \in \overline{A} \wedge ... \wedge y_m \in \overline{A} \wedge (< x_1, x_2, ..., x_n >$,p$) \in S$?" to a coNP oracle. The last part of the question is for the accepting path p of N_1. If the oracle's answer is "no", then M' rejects the input. If the coNP oracle answers positively, then all the guesses of M' are correct and M' accepts the input. Furthermore, because of the checking $(< x_1, x_2, ..., x_n >$,p$) \in S$, each accepting path of M corresponds to exactly one accepting path of M'.

Thus, f belongs to $\# \cdot coNP$.

(iii) Here, we give only the proof for $\# \cdot NP = \# \cdot coNP \Longleftrightarrow NP = coNP$.

($\Longrightarrow$) Let $A \in NP$ and M the machine that on input x asks the NP oracle for A and accepts iff the oracle answers "yes". Then, M witnesses the function f, $f(x) = 1 \Longleftrightarrow x \in A$. So $f \in \# \cdot NP$. From our assumption, it holds that $f \in \# \cdot coNP$. Thus, $x \in A$ is equivalent with one accepting path of some machine N that asks a coNP oracle at the end and receives a positive answer. This accepting path corresponds to a coNP computation that accepts x iff $x \in A$. So, $A \in coNP$. We conclude that $NP \subseteq coNP$, which means that $NP = coNP$.

($\Longleftarrow$) If $NP = coNP$, an NP oracle used by a machine at the end of some computation can be replaced by a coNP oracle and vice versa.

(iv) It is not hard to see that if $f \in \# \cdot NP$, then for some polynomial q it holds that $f'(x) = 2^{q(x)} - f(x)$ is in $\# \cdot coNP$. ■

As a result of Theorem 3.1.1 (ii), if the function f "counts" the number of the accepting paths of a NPTM M with an NP oracle A, we can assume that the queries to the oracle A is replaced by only one oracle query to some oracle $A' \in coNP$.

Köbler et al. defined the class *span-P* of all the classes counting the number of different outputs of nondeterministic polynomial-time transducers.

They also proved basic relationships among the classes $\#P$, *span-P* and $\#NP$.

Proposition 3.1.1 *(i)* $\#P \subseteq$ *span-P.*
(ii) span-P $\subseteq \#NP$.
(iii) span-P $= \# \cdot NP$.

Proof. (i) Similarly to Proposition 2.1.1.

(ii) Let $f \in$ *span-P*, i.e. $f = span_M$ for some nondeterministic polynomial-time transducer M. Let M' be the NPTM which on input x guesses a string y and asks an NP oracle if M outputs y on input x. The problem "The string y is an output of M on input x?" can be solved by an NPTM which on input x guesses a path w, simulates M's computation and checks that at the end M outputs y.

(iii) *span-P* $\subseteq \# \cdot NP$ can be proved as in (ii) above.

We show that $\# \cdot NP \subseteq$ *span-P*. Let $f \in \# \cdot NP$ and M the machine witnessing f. We construct an NP transducer M' witnessing f. On input x, M' guesses a string y and simulates M's computation path y on input x. At the end, instead of M's oracle call, M' answers the NP oracle question on its own and outputs y if and only if the answer is "yes". Clearly, the number of different outputs of M' is equal to the number of accepting paths of M. ■

Proposition 3.1.2 $\#NP = \{f : f = span_{M-M'}$, *for some pair of nondeterministic, polynomial-time Turing transducers M, M'*$\}$, *where* $span_{M-M'}(x)$ *is the number of different outputs that* M *on input* x *can produce that cannot be produced by* M'.

Proposition 3.1.3 *If* $f \in \#NP$ *then there are two functions* $g_1, g_2 \in$ *span-P, such that for every* x, $f(x) = g_1(x) - g_2(x)$.

Proof. Let $f \in \#NP$. Then there is a pair of nondeterministic polynomial-time transducers M, M' such that $f = span_{M-M'}$. Let $g_1 = span_M$ and $g_2 = span_{M''}$, where M'' is the nondeterministic polynomial-time transducer that on input x guesses a string y and if $y \in span_{M(x)} \cap span_{M'(x)}$ then outputs y. Obviously the problem "The string y is an output of M and an output of M' on input x?" can be solved by an NPTM which guesses a path w_1 of M and a path w_2 of M' and checks that at the end of these computation paths both transducers output y. It follows that $f(x) = g_1(x) - g_2(x)$. ■

We have already defined the class UP to be the class of NP sets having a unique accepting computation. It is well known that $P \neq UP$ if and only if one-way functions exist [22]. Note that in [6] it is also shown that under the assumption $P \neq NP$, $UP \neq NP$ if and only if there exist one-way functions whose range is an NP-complete set.

It is not likely that the class $\#P$ equals the class *span-P* since this question can be reduced to the question whether UP equals NP.

Proposition 3.1.4 *(i) span-P* $= \#NP \Longleftrightarrow NP = coNP$.
(ii) $\#P =$ *span-P* $\Longleftrightarrow UP = NP$.

Proof.

(i) It is immediate from Proposition 3.1.1 (iii) and Theorem 3.1.1 (ii) and (iii).

(ii) Suppose $\#P =$ *span-P*, and let $A \in NP$ and M the corresponding NPTM. Define M' such that it outputs "1" if M accepts, and nothing otherwise. Then, $span_{M'}$ is the characteristic function of A, which by the assumption, is in $\#P$. That is, there is a nondeterministic, polynomial-time Turing machine which has one accepting computation on inputs $x \in A$, and none on $x \notin A$. Thus $A \in UP$.

Conversely, assume $UP = NP$ and let $f = span_M$ be a member of *span-P*. Then the set $\{(x, y) :$ M on input x outputs $y\}$ is obviously in NP, and hence in UP by the assumption. That is, there is a nondeterministic polynomial-time machine M' for this set: If M' accepts an input (x, y) then it has a unique accepting computation. Define the nondeterministic machine M'' which on input x guesses y and simulates M' on input (x, y). Then we have $f = acc_{M''}$, which shows that $f \in \#P$. ■

Comment 3.1.2 *Proposition 3.1.4 (ii) is immediate consequence of Theorem 3.1.1 (i) and Proposition 3.1.1 (iii).*

3.2 #L, span-L and the operator #·

The question we are concerned here is: Can we find a natural operator such that, similar to the above result for polynomial time, *span-L* $= \# \cdot NL$? The answer is in the affirmative, using a variation of an NL Turing machine.

Definition 3.2.1 *A* 2-1-*TM is a Turing machine with two input tapes. The first is a two-way (that can be read as often as necessary) and the second is a one-way input tape (that can be read only once).*

Definition 3.2.2 *(i) Let* 2-1-L *be the class of polynomially length-bounded two-argument predicates (binary relations)* $\mathcal{R}$ *that are accepted by deterministic logspace* 2-1-*Turing machines.*
(ii) Let 2-1-NL *be the class of polynomially length-bounded two-argument predicates* $\mathcal{R}$ *that are accepted by nondeterministic logspace* 2-1-*Turing machines.*
(iii) In the same way, we define 2-1-$\Sigma_k L$ *(*2-1-$\Pi_k L$*) to be the class of polynomially length-bounded two-argument predicates* $\mathcal{R}$ *that are accepted by log-space* 2-1-Σ_k*-Turing machines (*2-1-Σ_k*-Turing machines resp.). A* 2-1-Σ_k*-Turing machine (*2-1-Σ_k*-Turing machine) is an alternating Turing machine that makes at most* $k-1$ *alternations and it starts in an existential (universal resp.) configuration.*

It can be easily proved that a language L is in NL if and only if there is a 2-1-TM M, such that for every x: $x \in L \Leftrightarrow \exists y, |y| = p(|x|)$ and M accepts (x, y). In other words, $x \in L \Leftrightarrow \exists y R(x, y)$ for some relation R in the class 2-1-L.

For the corresponding counting classes we give the following definitions. The class *span*-L is equivalent to counting the second arguments of 2-1-NL relations and not just 2-1-L (deterministic) relations.

Definition 3.2.3 *(i)* $\# \cdot L$ *(or* $\#\Sigma_0 L$*) is the class of (total) functions* f *such that, for some two-argument predicate* $\mathcal{R} \in$ 2-1-L *and some polynomial p, for every string x it holds that:*
$f(x) = |\{y : |y| = p(|x|) \text{ and } \mathcal{R}(x, y)\}|$.
(ii) $\# \cdot NL$ *(or* $\#\Sigma_1 L$*) is the class of (total) functions* f *such that, for some two-argument predicate* $\mathcal{R} \in$ 2-1-NL *and some polynomial p, for every string x it holds that:*
$f(x) = |\{y : |y| = p(|x|) \text{ and } \mathcal{R}(x, y)\}|$.
(iii) $\#\Sigma_k L$ *(*$\#\Pi_k L$*) is the class of (total) functions* f *such that, for some two-argument predicate* $\mathcal{R} \in$ 2-1-$\Sigma_k L$ *and some polynomial p, for every string x it holds that:*
$f(x) = |\{y : |y| = p(|x|) \text{ and } \mathcal{R}(x, y)\}|$.

Comment 3.2.1 *(i) Similarly, one can define* $\# \cdot coNL$ *(or* $\#\Pi_1 L$*). Although,* $NL = coNL$ *[26] [45], this closure complementation does not necessarily hold for* 2-1-NL*. Here the input cannot be read several times. Thus, it is not known whether* $\# \cdot NL = \# \cdot coNL$*. In fact we show that this equality is very unlikely.*

(ii) Observe that all classes $\mathcal{C}$ *that we considered in the previous section are supersets of* P *for which defining* $\# \cdot \mathcal{C}$ *as in Definition 3.1.2 or by using* 2-1-$\mathcal{C}$ *predicates analogous to the approach of Definition 3.2.3 makes no difference (as the machine could start by deterministically copying its second tape's contents onto its work tape). This is the reason why we use the same notation* $\#\cdot$*, without becoming inconsistent.*

(iii) An alternative way of defining $\# \cdot NL$ *is as the class of functions "counting" the accepting paths of some* NL *machine* M *which makes one* 2-1-NL *oracle call at the end of every computation path and returns the oracle's answer. We assume that* M *has two oracle tapes, which are an instance of a* 2-1-NL *problem. For every input* x *the machine* M *copies* x *on the first oracle tape and writes some string* y *on the second oracle tape. The proof that*

the two definitions are equivalent is the same as the proof for the two definitions for $\# \cdot NP$ in the Comment 3.1.1 (iii).

(iv) The class $\#\Sigma_k L$ ($\#\Pi_k L$), for some $k \geqslant 0$, is the class $\# \cdot$ 2-1-$\Sigma_k L$ ($\# \cdot$ 2-1-$\Pi_k L$ resp.). When we refer to this class, we use the former notation.

Theorem 3.2.1 *(i) $\# \cdot L = \#L$.*
(ii) $\# \cdot NL = span\text{-}L$.

Proof. (i) $\#L \subseteq \# \cdot L$: Let M be an NL machine witnessing $f \in \#L$. The predicate $\mathcal{R} = \{(x, y) \mid y \textit{ is an accepting path of } M \textit{ on input } x\}$ is computable by the 2-1-L machine M' which on inputs x, y simulates M's path y on input x and accepts iff M accepts.
$\#\cdot L \subseteq \#L$: Let $f \in \#\cdot L$ and $f(x) = |\{y \mid p(|x|) = |y| \text{ and } \mathcal{R}(x, y)\}|$, where $\mathcal{R}$ is computable by some 2-1-L-TM N. Define the NL machine M' which on input x, guesses a y bit by bit, since $|y| = p(|x|)$, and simulates N on inputs x, y. M' accepts iff the simulation of N halts accepting.

(ii) *span-L* $\subseteq \# \cdot NL$: Let $f \in$ *span-L*, i.e. $f = span_M$ for some nondeterministic log-space transducer M. Let M' be the NL machine which on input x copies x on the first oracle tape, guesses and writes a string y on the second oracle tape and asks a 2-1-NL oracle if M outputs y on input x. The problem "The string y is an output of M on input x?" can obviously be solved by an 2-1-NL Turing machine.
$\# \cdot NL \subseteq$ *span-L*: Let $f \in \# \cdot NL$ and the NL machine M with one 2-1-NL oracle call at the end of every computation path, witnessing f. In addition, there is a 2-1-NL Turing machine N which can "answer" the oracle questions. We convert each computation path of M to a path of an NL transducer M'. A path of M on input x is a log-space computation ending to writing x and y on the oracle tapes. The corresponding path of M' does not call the oracle, but answers the oracle question on its own. The only problem with this idea is that M' cannot store y on its work tape, because this string may be much longer than $log|x|$. Thus M' on input x simulates N on input (x, y) and each time N requires the next bit of y, M' simulates M on input x until this bit is generated. After the current bit is used by N's computation, it is deleted from M''s work tape. As y is read only one-way by N the i^{th} bit of y is not used after the $i + 1^{th}$ bit and the computation can be done by a nondeterministic log-space Turing machine. If the simulation of N ends accepting (x, y), then the transducer M' outputs the encoding of its own computation path. The number of different outputs of M' is the number of accepting paths of M.

It is worth noting that this part of the proof would fail if the NL machine M asked an NL oracle at the end of some computation path (instead of a 2-1-NL oracle). ■

Can we compare polynomial-time classes to logarithmic-space classes? The first easy answer to this question, is that although $\#P = \#\cdot coNL$, it is very unlikely that $\#P = \#\cdot NL$.

Following the definitions of the previous subsection, let $\#NL$ denote the class of functions that witness the number of accepting computations of NL machines with oracle calls to NL. It holds that (1) $EVAL3SAT \in L$, and hence $NP \subseteq NL^L \subseteq NL^{NL}$ and (2) it is clear that $NL^{NL} \subseteq P \subseteq NP$. From (1) and (2) the "decision" classes NP and NL coincide.

The same equality is true for the corresponding counting classes, i.e. $\#NL = \#P$, since (1) $\#3SAT \in \#NL \Longrightarrow \#P \subseteq \#NL$ and (2) let M the Turing machine witnessing some $f \in \#NL$, then each computation of M belongs to NL^{NL}, which means that it also belongs to P. Hence $\#NL^{NL} \subseteq \#P$.

Theorem 3.2.2 *(i)* $\#P = \#NL$.
(ii) $\#P = \# \cdot coNL$.
(iii) $\#P = \# \cdot NL = span\text{-}L \Longrightarrow NL = UP$.

Proof (iii) Let $A \in UP$ and M the corresponding Turing machine. The function $f = acc_M \in \#P$ is the characteristic function of the set A, since $f(x) = 1$ if $x \in A$, and $f(x) = 0$ if $x \notin A$. Because of our hypothesis, $f \in \# \cdot NL$. Thus, there is a nondeterministic log-space machine M' with one 2-1-NL oracle call at the end of its computation, and with only one accepting path iff $x \in A$. The existence of M leads to a nondeterministic log-space computation with one 2-1-NL oracle call for A, which can be transformed to an NL computation for A.

Sketch proof for (ii). The $\#P$-complete problem $\#3SAT$ can be solved by an NL machine that asks a 2-1-$coNL$ oracle question after some computation. Given a $3CNF$ formula this machine guesses an assignment for the formula and writes this assignment on its oracle tape. The problem of verifying that an assignment satisfies a $3CNF$ formula is in 2-1-coNL. For the inverse inclusion the proof of the Proposition 3.3.3 below is clarifying. ■

3.3 Restrictions of log-space classes

In the current section we consider three kinds of restriction of a log-space class:

- The Turing machine can read the input only one-way.
- The Turing transducer can output only logarithmic number of bits.
- The Turing machine has polynomial number of accepting paths.

The difference between #- and *span*-classes exactly corresponds to the difference between unambiguous and ambiguous computation. The proof that $NP = UP$ if and only if $\#P = span\text{-}P$ can be translated to log-space counting classes when one-way machines are considered.

For two-way log-space classes the proof technique does not work. Furthermore, in [41] was stated that "NL and UL are probably equal". If this intuition is right and the equality $NL = UL$ implies $span\text{-}L = \#L$, then the polynomial hierarchy would collapse. But although we can prove $span\text{-}L = \#L \Rightarrow NL = UL$, the inverse implication is still open.

The class UL contains all NL sets having a unique accepting computation and $1UL$, $1NL$, *span*-$1L$, $\#1L$ are the one-way analogs of UL, NL, *span*-L and $\#L$ respectively.

Theorem 3.3.1 $1UL = 1NL$ *if and only if* *span*-$1L = \#1L$.

Proof. Assume $1NL \subseteq 1UL$ and let $f = span_M$, where M is a $1NL$-transducer. Consider the set $A = \{x_1\#y_1!x_2\#y_2!...x_n\#y_n!|x_i \in \{0,1\}, y_j \in \{0,1\}^*$ and on input $x = x_1x_2...x_n$ there is a computation path on which M outputs $y_1, ..., y_n$ such that output y_i occurs after reading bit x_i and before reading input bit $x_{i+1}\}$.
As M is a one-way transducer, it can be shown that $A \in 1NL$ and, hence, $A \in 1UL$ by assumption. Let M' be a $1UL$-machine for A. M' has a unique accepting computation, when it accepts. Construct a $1NL$-machine M'' that on input x does the following: M'' guesses $x_1\#y_1!x_2\#y_2!...x_n\#y_n!$ bit-by-bit, simulates M' on this word, and checks that $x = x_1...x_n$. Then $f(x) = acc_{M''}(x)$, and $f \in \#1L$.

Conversely, assume that $span\text{-}1L = \#1L$. Let $A \in 1NL$, and let M be an $1NL$-machine that accepts A. Construct an $1NL$-transducer M' that on input x simulates M and outputs "1" iff M accepts. Then $span_{M'}$ is the characteristic function c_A of A. By assumption, $c_A \in \#1L$. Thus, there exists a $1NL$-machine that on input x has one accepting computation, if $x \in A$, and none, if $x \notin A$. Hence, $A \in 1UL$, and $1NL = 1UL$. ■

Corollary 3.3.1 *$span\text{-}1L = \#1L$ implies $P = NP = P^{\#P} = P^{span-L}$.*

The restriction to one-way classes was one approach in order to prove Proposition 3.1.4 (ii) for log-space classes. A different approach is the following restriction of *span-L*.

Definition 3.3.1 *For a class of functions F, define $F[logn] = \{f \in F : \exists$ a constant c $\forall x$ $|f(x)| \leqslant c \cdot log|x|\}$.*

The class *span-L*$[logn]$ consists of the functions witnessed by NL-transducers that can output only $c \cdot log(|x|)$ bits at the end of each computation.

Proposition 3.3.1 *$\#L = span\text{-}L[logn] \Longleftrightarrow UL = FewL$.*

Proof. Suppose $\#L = span\text{-}L[logn]$, and let $A \in FewL$ and M the corresponding NL Turing machine. Define M' such that it outputs "1" if M accepts, and nothing otherwise. Then, $span_{M'}$ is the characteristic function of A, c_A. Thus, $c_A \in span\text{-}L[logn]$ and by the assumption, $c_A \in \#L$. That is, there is a nondeterministic, log-space Turing machine which has one accepting computation on inputs $x \in A$, and none on $x \notin A$. Thus $A \in UL$.

Conversely, let $UL = FewL$ and let $f = span_M$ be a member of $span\text{-}L[logn]$. Then the set $A = \{(x, y) : |y| \leqslant c \cdot log(|x|)$ and M outputs y on input $x\}$ is in $FewL$: An NL machine N simulates M on input x and checks that M's output is equal to y. Since M has at most $poly(|x|)$ different outputs for some polynomial p, the machine N has polynomial number of accepting paths. Hence $A \in UL$ by the assumption. That is, there is a nondeterministic log-space machine M' for the set A: If M' accepts an input (x, y) then it has a unique accepting computation. Define the nondeterministic log-space machine M'' which on input x guesses y and simulates M' on (x, y). Then we have $f = acc_{M''}$, which shows that $f \in \#L$. ■

Definition 3.3.2 *(i) A* 1-1-*TM is a Turing machine with two one-way input tapes.*
(ii) Let 1-1-*L*, 1-1-*NL be the classes of polynomially length-bounded two-argument predicates* $\mathcal{R}$ *that are accepted by deterministic and nondeterministic logspace* 1-1-*Turing machines respectively.*
(iii) $\# \cdot 1L$ *(or* $\#\Sigma_0 1L$*) is the class of functions* f *such that, for some two-argument predicate* $\mathcal{R} \in$ 1-1-L *and some polynomial* p*, for every string* x *it holds that* $f(x) = |\{y : p(|x|) = |y|$ *and* $\mathcal{R}(x, y)\}|$.
(iv) $\# \cdot 1NL$ *(or* $\#\Sigma_1 1L$*) is the class of functions* f *such that, for some two-argument predicate* $\mathcal{R} \in$ 1-1-NL *and some polynomial* p*, for every string* x *it holds that* $f(x) = |\{y : p(|x|) = |y|$ *and* $\mathcal{R}(x, y)\}|$.
(v) The classes $\#\Sigma_k 1L$ *and* $\#\Pi_k 1L$ *are defined analogously.*

Theorem 3.2.1 holds for the corresponding one-way classes as well.

Proposition 3.3.2 *(i)* $\#1L = \# \cdot 1L$.
(ii) span-$1L = \# \cdot 1NL$.

Comment 3.3.1 *It is easy to observe that the* $\#P$*-complete (under "Cook" reductions) problem* $\#3DNF$ *is in the class* $\#\Sigma_1 1L$*. Given a* $3DNF$ *formula* F *and an assignment of* F*, a nondeterministic* 1-1-*Turing machine guesses a clause* C *of* F*, writes on its work tape the three literals occurring in* C*, and reads the truth value of these variables given on the second input. Obviously, the two inputs can be read from left to right. Thus,* $\#3DNF \in$ *span*-$1L$.

Based on the definitions of $\#\Pi_k L$ and $\#\Pi_k 1L$ classes, for $k \geqslant 1$, and Lemma 2.2.1, we can prove the following proposition.

Proposition 3.3.3 $\#\Pi_1 1L = \#\Pi_1 L = \#P$.

Proof. $\#\Pi_1 1L \subseteq \#\Pi_1 L \subseteq \#P$: A nondeterministic polynomial time bounded Turing machine can guess the second input of a 2-1- Turing machine and simulate the universal branches deterministically. Since the computation of each branch is log-space bounded, the total computation is polynomial time bounded.

$\#P \subseteq \#\Pi_1 1L$: Let the relation $R_{3SAT} = \{(\phi, b) : \phi \in 3CNF$ and b is a satisfying assignment for $\phi\}$. It holds that $R_{3SAT} \in$ 1-1-$\Pi_1 1L$: A universal 1-1-Turing machine guesses universally a clause of ϕ, copies the literals of this clause on its working tape and reading the second input b one-way checks that the clause is satisfiable. Thus, $\#3SAT \in \#\Pi_1 1L$. Also there exists a parsimonious $1L$ reduction from every problem in $\#P$ to $\#3SAT$. ■

Regarding the class *span-L*, we can define the following subclasses:
span-L_p: $\{f : f = span_M$ for some NL transducer M having at most polynomial number of different valid outputs$\}$.
span-$L[determ]$: $\{f : f = span_M$ for some NL transducer M that writes its output deterministically$\}$.

span-L[*logn*]: $\{f : f = span_M$ for some NL transducer M that writes at most logarithmic number of bits on its output tape$\}$.

It holds that *span-L*[*determ*] $\subseteq$ *span-L*[*logn*] $\subseteq$ *span-L*$_p$.

- *span-L*[*determ*] $\subseteq$ *span-L*[*logn*]: Let $f \in$ *span-L*[*determ*] and M the transducer witnessing f. The output w_c printed along one path of M is completely determined by the first configuration c on the path in which a symbol of the output occurs. Since there are only polynomially many different configurations in the length n of the input, each one of them can be mapped to a different string of length $\mathcal{O}(log(n))$. However we should map two configurations c and c' to the same string if $w_c = w_{c'}$. A transducer M' simulates M and when M reaches the configuration c_w in which the first symbol of the output w is printed, M' outputs the string (of logarithmic length) mapped to c_w. Clearly, $f \in$ *span-L*[*logn*].

- *span-L*[*logn*] $\subseteq$ *span-L*$_p$: Since the outputs are of logarithmic length, there are at most polynomial many such outputs.

The inverse inclusion *span-L*$_p$ $\subseteq$ *span-L*[*determ*] can be proved via the class FL^{NL}[*logn*]. In [1] was shown that *span-L*[*determ*] $= FL^{NL}$[*logn*]. We show here that *span-L*$_p$ $= FL^{NL}$[*logn*].

Lemma 3.3.1 *Given numbers* $k_1, k_2, ..., k_t$*, where* $\forall i\ k_i \leqslant 2^n$ *and* $t \leqslant p(n)$ *and* $k_i \neq k_j$ *for all* $i \neq j$ *, there exists a prime number* $q < p^3(n) \cdot n^2$ *such that for all* i *and* j *and* $i \neq j$*,* k_i *mod* $q \neq k_j$ *mod* q*.*

Theorem 3.3.2 *span-L*$_p$ $= FL^{NL}$[*logn*]*.*

Proof. *span-L*$_p$ $\subseteq FL^{NL}$[*logn*]: Let M be an NL transducer having $p_1(|x|)$ valid outputs on input x for some polynomial p_1, and let f be the $span_M$ function. We show the construction of an FL^{NL} machine which compute the same function f and its output is logarithmic in the length of x. There exists a polynomial p_2 such that each one of M's valid outputs on input x has a length bounded by $p_2(|x|)$. So, each output can be represented by $p_2(|x|)$ bits. By Lemma 3.3.1 there exists a prime $q < p_2^3(|x|) \cdot p_1^2(|x|)$ such that the integer representations of all the valid outputs $y_1, ..., y_{p_1(|x|)}$ taken *mod* q are all distinct, i.e y_i *mod* $q = w_i$ and all w_i are distinct. The w_i can be represented by $\mathcal{O}(log|x|)$ bits. To find q, queries are sent to an oracle A:
$A = \{(M, x, q)$: on input x, there are at least two accepting paths of M whose outputs are distinct but when taken *mod* q, have the same value$\}$.
A lies in NL: The corresponding NL machine guesses two paths and computes the *mod* q of their outputs in parallel and verifies whether the mods are equal and the paths are accepting. The outputs of the paths are stored bit by bit on the work tape of the machine.
Now, the machine asks if w represents a valid output for all $w < q$. The queries are made to an oracle B:

$B = \{(M, x, w) : w$ represents a valid output of M on input $x\}$.
B lies in NL: The corresponding NL machine guesses a path and verifies that the mod of its output is w.
Every time a new representation w of a valid output is found, the machine increments a counter by one (since the number of the different outputs is bounded by a polynomial, a counter can be maintained in logspace). The value of the counter at the end gives the value of $f = span_M$, and it is logarithmic in $|x|$.

$FL^{NL}[logn] \subseteq$ *span*-L_p: Let f be in $FL^{NL}[logn]$. Since NL is closed under complementation, a deterministic log-space transducer M with oracle $A \in NL$ computing f can be simulated by an NL transducer M' which uses subroutines for A and $\bar{A}$ to solve the oracle queries: For any oracle query w, the answer "yes" or "no" is guessed and the subroutine for A or $\bar{A}$ is started respectively. If the corresponding subroutine ends answering "yes" (which means that the guessed oracle answer made by M' was correct) the computation is continued, otherwise M' halts rejecting. Since the length of w can be polynomial in the length of x, w is produced bit by bit. If some bit of w is necessary to be produced again, M' computes w from the beginning (this computation is deterministic). Every output of M' is equal to $f(x)$. We can construct a new transducer N, which on input x simulates M' and for every output y of M' guesses a $z \leqslant y$ and outputs z. Since neither y nor z can be stored on the work tapes, N first guesses $|y|$ and then outputs a value z of length $|y| - 1$ or a value z of length $|y|$ smaller or equal to y. This is done comparing the symbols of y produced by M' to the symbols of z. It is clear that $span_N(x) = f(x)$. Hence $f \in$ *span*-$L[logn]$. ■

Corollary 3.3.2 *span*-$L[determ]$ = *span*-$L[logn]$ = *span*-L_p.

3.4 The counting class TotP and its relationship with span-L

In [38], Pagourtzis and Zachos introduced the classes $\#PE$ and $TotP$. The first one is the class of "*hard-to-count-easy-to-decide*" problems and the second one is the class of "counting" all the computation paths of a polynomial-time nondeterministic machine.

More formally, for each function $f : \{0,1\}^* \to \mathbb{N}$ we define a related language $L_f = \{x \mid f(x) > 0\}$. Note that for function problems this language represents a natural decision version of the problem.

Definition 3.4.1 *Let the function* $f : \{0,1\}^* \to \mathbb{N}$, *then* $f \in \#PE$ *if* $f \in \#P$ *and* $L_f \in P$.

Definition 3.4.2 *For a machine* M, *let* tot_M *denote the function from* $\{0,1\}^*$ *to* $\mathbb{N}$ *such that* $tot_M(x) = (\#paths\ of\ M\ on\ input\ x) - 1$. *Define*
$TotP = \{f : f = tot_M$ *for some polynomial-time nondeterministic Turing machine* $M\}$.

Some $\#PE$ problems are in $TotP$. The key property of these problems is the possibility to reduce them to other instances of the same problem. The definition of self-reducibility is given below:

Definition 3.4.3 *A function $f : \Sigma^* \to \mathbb{N}$ is called poly-time self-reducible if there exist polynomials r and q and polynomial time computable functions $h : \Sigma^* \times \mathbb{N} \to \Sigma^*$,*
$g : \Sigma^* \times \mathbb{N} \to \mathbb{N}$ and $t : \Sigma^* \to \mathbb{N}$ such that for all $x \in \Sigma^*$:
(a) $f(x) = t(x) + \sum_{i=0}^{r(|x|)} g(x,i) f(h(x,i))$, that is, f can be processed recursively by reducing x to $h(x,i)$, $0 \leqslant i \leqslant r(|x|)$, and
(b) the recursion terminates after at most polynomial depth, that is, $f\big(h(...h(h(x,i_1),i_2)...,i_{q(|x|)})\big)$ can be computed in polynomial time.

Let $\#PE_{SR}$ be the class of self-reducible functions of $\#PE$. In [38] it was also proved the following important theorem.

Theorem 3.4.1 *$TotP$ is exactly the closure under polynomial-time functional many-one reductions ($\leqslant_m^p$) of $\#PE_{SR}$.*

Proposition 3.4.1 *The following problems belong to $TotP$. They are also TotP complete with respect to metric reducibility (Cook reducibility).*
1. $\#DNF$: given a DNF Boolean formula, count the number of its satisfying assignments.
2. $\#MONOTONE$-2-SAT: given a CNF formula with exactly two positive literals per clause, count the number of its satisfying assignments.
3. $\#NON$-$CLIQUES$: given a graph G and a number k, count the number of size-k subgraphs of G that are not cliques.
4. $\#NON$-IS: given a graph G and a number k, count the number of size-k subgraphs of G that are not independent sets.
5. $\#NON$-$NEGATIVE$-$PERMANENT$: given a $n \times n$ matrix A with nonnegative integer entries, compute its permanent.
6. $\#NFA$: given an NFA M and a string x, count the number of strings $\leqslant x$ that are accepted by M.

Using Theorem 3.4.1 we can prove the connection between $TotP$ and *span-L*.

Theorem 3.4.2 *span-L $\subseteq TotP$.*

Proof. Let $f \in$ *span-L* via an NL transducer M ($f \equiv span_M$). It suffices to show that f is poly-time many-one reducible to a $\#PE_{SR}$ function.

It holds that $L_f \in NL \subseteq P$: We can convert M to a logspace acceptor M': M' accepts whenever M gives an output and therefore has at least one accepting path iff M has at least one output. In Theorem 2.3.2, we showed that *span-L* $\subseteq \#P$. These two facts mean that $f \in \#PE$.

We observe that the number of different outputs of M is equal to the number of different outputs of M starting with "0" plus the number of different outputs of M starting with "1". Define the NL transducer N, which on input (x, y) generates the outputs of M on input x that have y as a prefix. The *span-L* function of N is self-reducible: $span_N(x,y) = span_N(x,y0) + span_N(x,y1) + g(x,y)$, where

$$g(x,y) := \begin{cases} 1 & \text{if } y \text{ is an output of N on input x} \\ 0 & \text{otherwise.} \end{cases}$$

Similarly to $span_M$, $span_N$ belongs to $\#PE$. Moreover, $span_M(x) = span_N(x,\epsilon)$, which means that $span_M \leqslant_m^p span_N \in \#PE_{SR}$. ■

Similarly to *span-L*, $TotP$ and $\#P$ share the same complete functions with respect to metric reducibility.

Proposition 3.4.2 $P^{span-L} = P^{TotP} = P^{\#P}$.

3.5 The log-space analog of the class TotP

We introduce now the counting class $TotL$. Analogously to $\#PE$ and $TotP$, [38], we have the following definitions.

Definition 3.5.1 *Define the class* $\#LE = \{f \mid f \in \#L \text{ and } L_f \in L\}$,
where $L_f = \{x \mid f(x) > 0\}$.

Definition 3.5.2 *For a machine M, let tot_M denote the function from $\{0,1\}^*$ to $\mathbb{N}$ such that $tot_M(x) = (\#paths\ of\ M\ on\ input\ x) - 1$. Define*
$TotL = \{f \mid f = tot_M$ *for some log-space nondeterministic Turing machine* $M\}$.

But, which is the relationship between these two classes and $\#L$? Do the statements about $\#PE$, $TotP$ and $\#P$ hold when we refer to the corresponding log-space counting classes? Below, we answer this question in the affirmative.

Proposition 3.5.1 *(i)* $\#L \subseteq TotL - FL$.
(ii) $FL \subseteq TotL \subseteq \#LE \subseteq \#L$.

Proof. (i) Let $f \in \#L$ via an NL machine M. We can convert M so that we have an NL machine M' with a full binary computation tree and then we double the accepting paths of M' obtaining the NL machine N with $tot_N(x) = tot_{M'}(x) + acc_{M'}(x)$, for every x. Clearly, $f(x) = acc_M(x) = acc_{M'}(x) = tot_N(x) - tot_{M'}(x) = tot_N(x) - (2^{p(|x|)} - 1)$, where $p(|x|)$ is the length that has every computation path of M''s full binary tree. Observe that $2^{p(|x|)}$ can be computed in logarithmic space using $log\big(p(|x|)\big)$ bits to store $p(|x|)$.

(ii) Let $f \in FL$. Define the NL machine M which computes $f(x)$ bit by bit. When a bit is computed, M branches in such a way that the number of its computation paths

are equal to $f(x) + 1$ ($f(x)$ paths and one dummy path). For this branching, M needs two states:
q_0, which allows M to nondeterministically choose between two actions and
q_1, which forces M to deterministically continue if the current bit is "0" and allows M to nondeterministically choose between two actions if the current bit is "1". Hence,
$tot_M(x) = \big(f(x) + 1\big) - 1 = f(x)$, for every x. On the other hand, if $TotL \subseteq FL$, then by (i) of this proposition, we would also have $\#L \subseteq FL$ and we would be able to compute $f_{\#path}$ in deterministic logarithmic space. This would mean that $REACHABILITY \in L$ and $L = NL$.

For the second inclusion, let $f \in TotL$ with $f(x) = (\#paths\ of\ M\ on\ input\ x) - 1$, for some M. The leftmost path of the machine M can be distinguished (nondeterministic choices can be lexicographically ordered). So, we define the modified machine M' with all the paths of M accepting, except for the leftmost path. For every input x, it holds that $f(x) = tot_M(x) = (\#paths\ of\ M\ on\ input\ x) - 1 = \#accepting\ paths\ of\ M'\ on\ input\ x = acc_{M'}(x)$. It remains to show that $tot_M(x) > 0$ can be checked in deterministic logarithmic space. Equivalently, that $(\#paths\ of\ M) > 1$. For this, it suffices to simulate M until it branches: If it does not, then $tot_M(x) = 0$, otherwise, $tot_M(x) > 0$. We conclude that $f \in \#L$ and $L_f \in L$. To see that $\#LE \nsubseteq TotL$, unless $L = NL$, consider the problem $f_{\#path+1}$: for any input directed graph return the number of paths from node 1 to node n plus one. This problem is in $\#LE$ since it is in $\#L$ and $f_{\#path+1} > 0$, for every input x. If $f_{\#path+1} \in TotL$, then there is an NL machine N such that $f_{\#path+1}(x) = (\#paths\ of\ N\ on\ input\ x) - 1 \Leftrightarrow f_{\#path}(x) = (\#paths\ of\ N\ on\ input\ x) - 2$. The $REACHABILITY$ is equal to solving $f_{\#path}(x) > 0$ for any input x or equivalent to $\big((\#paths\ of\ N\ on\ input\ x) - 2\big) > 0$, which can be solved in deterministic logarithmic space. ■

A 2004 result by Omer Reingold shows that $USTCON$, the problem of whether there exists a path between two vertices in a given undirected graph (undirected reachability), is in L (showing that $L = SL$, since $USTCON$ is SL-complete) [40].

Similarly to $\#Path$, the counting version of $USTCON$ belongs to $\#L$:

$\#USTCON$:
Input: An undirected graph $G = (V, E)$ with vertex set $V = \{1, 2, ..., n\}$.
Output: The number of different paths of length at most n from vertex 1 to vertex n.

Hence, the counting problem $\#USTCON$ belongs to the class $\#LE$. Does $\#USTCON$ belong to $TotL$?

When we refer to counting in log-space, what kind of self-reducibility would make sense? Could we have a result similar to Theorem 3.4.1, i.e. a result that associates $TotL$ with $\#LE$? There is no easy answer to this question. However, restricted versions of $\#USTCON$ are in $TotL$.

In the next definition $N(u)$ is the neighbourhood of a vertex u in a graph G, i.e. $N(u) = \{v \in V : E(u,v)\}$.

$\#UASTCON_{log}$:
Input: $(G, 1, n)$, where $G = (V, E)$ is an undirected acyclic graph with vertex set $V = \{1, 2, ..., n\}$ such that $|N(u)| = \mathcal{O}(logn)$ for every vertex $u \in \{1, ..., n\}$.
Output: The number of different paths from vertex 1 to vertex n.

$\#USTCON_{log}$:
Input: $(G, 1, n)$, where $G = (V, E)$ is an undirected graph with vertex set $V = \{1, 2, ..., n\}$ such that $|N(u)| = \mathcal{O}(logn)$ for every vertex $u \in \{1, ..., n\}$.
Output: The number of different paths from vertex 1 to vertex n.

Proposition 3.5.2 *(i)* $\#UASTCON_{log} \in TotL$.

Proof. Let $(G, 1, n)$ be an input of $\#UASTCON_{log}$. The properties mentioned in the definition of the problem can be checked in log-space. We can design a nondeterministic log-space algorithm with a computation tree with exactly $\#UASTCON_{log}(G) + 1$ leaves. A description of the algorithm follows.

```
begin
if $\#UASTCON_{log}(G,1,n) = 0$ then stop
else nondeterministically choose between

    • stop
    • call $GenTree_{\#UASTCON_{log}}(G, 1, \emptyset)$
end
```

The first nondeterministic choice creates a dummy additional path, whereas the second calls the following procedure.

```
procedure $GenTree_{\#UASTCON_{log}}(G, u, S)$
begin
if $u = n$ then stop
else
for each $v \in N(u) \setminus S$
$Previous(v) \leftarrow \{u\}$;
check whether $\#UASTCON_{log}(G \setminus Previous(v), v, n) > 0$;
nondeterministically choose between
spawn a nondeterministic branch for some $v \in N(u) \setminus S$ provided $\#UASTCON_{log}(G \setminus Previous(v), v, n) > 0$
```

```
call $GenTree_{\#UASTCON_{log}}(G, v, Previous(v))$;
end
```

The procedure $GenTree_{\#UASTCON_{log}}$ first checks whether it just discovered a path from vertex 1 to vertex n, by checking if $u = n$. In this case it stops. Otherwise it computes and stores all the neighbouring vertices to u that are contained in a path from 1 to vertex n. This can be done in deterministic logarithmic space because $\#UASTCON_{log} \in \#LE$, and $N(u) = \mathcal{O}(logn)$. Moreover, this computation exclude the vertices that are connected to n but are not in a path from 1 to n by searching the subgraph $G \setminus u$. The acyclicity of the graph guarantees the correct result. ■

3.6 Arithmetizing classes around NC^1 and L

The title of this chapter is inspired by the paper [36] by Limaye, Mahajan and Raghavendra Rao. Knowing that $NC^1 \subseteq L$, we would like to define an arithmetization of NC^1, the class $\#NC^1$, and study the relationship between this class and $\#L$.

3.6.1 Barrington's Theorem

Some definitions that we need are the following. The first one is about the class NC^1 and we have already seen this.

Definition 3.6.1 *We say that the language L is decided by $\{C_n\}_{n\in\mathbb{N}}$, if for every $x \in \{0,1\}^n$, it holds that $C_n(x) = 1$ iff $x \in L$.*
A language L is in NC^1 if L can be decided by a family of circuits $\{C_n\}_{n\in\mathbb{N}}$, where C_n has poly(n) size, depth $\mathcal{O}(logn)$ and fan-in 2, for every n.

Buss [10] showed that the evaluation of Boolean formulae is a complete problem for NC^1. The class NC^1 contains functions which are computed by very fast parallel algorithms, such as the sum or product of 2 integers of n bits each, the sum of n integers of n bits each, integer or Boolean matrix multiplication, sorting n integers of n bits each and parity of n bits. It is the smallest class of the ten surveyed by Cook [12].

Definition 3.6.2 *A* ***branching program*** *is a finite directed acyclic graph which accepts some subset of $\{0,1\}^n$. Each node (except for the sink nodes) is labelled with an integer i, $1 \leqslant i \leqslant n$, and has two outgoing arrows labelled 0 and 1. This pair of edges corresponds to querying the i^{th} bit x_i of the input, and making a transition along one outgoing edge or the other depending on the value of x_i. There is a single source node, s, corresponding to the start state, and a sink node , t, corresponding to accepting state. An input x is accepted iff it*

induces a chain of transitions from s to t, and the set of such inputs is the language accepted by the program.
A branching program is ***oblivious*** *if the nodes can be partitioned into levels $V_1, ..., V_l$ and a level V_{l+1}, such that the nodes in V_{l+1} are the sink nodes, nodes in each level V_j, with $j \leqslant l$, have outgoing edges only to nodes in the next level V_{j+1}, and all nodes in a given level V_j query the same bit x_{i_j} of the input.*
Such a program is said to have ***length*** *l, and* ***width*** *k if each level has at most k nodes.*

Definition 3.6.3 *k-$BWBP$ is the class of languages recognized by polynomial length branching programs of width k, and $BWBP = \bigcup_k$ k-$BWBP$.*

It holds that, a language L is in *L/poly* if and only if it can be recognized by branching programs of polynomial size [2]. The class *L/poly* is a non-uniform logarithmic space class, analogous to the non-uniform polynomial time class P/poly. Formally, for a formal language L to belong to *L/poly*, there must exist an advice function f that maps an integer n to a string of length polynomial in n, and a Turing machine M with two read-only input tapes and one read-write tape of size logarithmic in the input size, such that an input x of length n belongs to L if and only if the machine M accepts the input $(x, f(n))$.

In 1989, Barrington showed that any language recognized by an NC^1 circuit can be recognized by a width-5 polynomial-size branching program.

Theorem 3.6.1 *(Barrington [5]) If a boolean function can be computed by a DeMorgan formula of a polynomial size, then it can also be computed by an oblivious width-5 branching program of polynomial length.*

Comment 3.6.1 *(i) A DeMorgan formula is a rooted binary tree in which each leaf is labelled by a literal of the set $\{x_1, ..., x_n, \overline{x_1}, ..., \overline{x_n}\}$ or a constant from $\{0, 1\}$ and each internal node is labelled by $\wedge$ ("and") or $\vee$ ("or"). Every DeMorgan formula computes in a natural way a boolean function from $\{0,1\}^n$ to $\{0,1\}$. The size of a DeMorgan formula is defined to be the number of leaves in it.*

(ii) Theorem 3.6.1 means that $NC^1 \subseteq BWBP$. In [5] the inverse inclusion $BWBP \subseteq NC^1$ was also proved. The classes NC^1 and $BWBP$ coincide and they equal the class 5-$BWBP$.

As stated in Lemma 1 of [36], BWBP coincides with BP-NFA, where BP-NFA is the class of all languages recognized by uniform polynomial length branching programs over a nondeterministic automaton. In more detail, a projection $P(x)$ of the input x is computed by a deterministic branching program of polynomial size and then $P(x)$ is given as input to a nondeterministic automaton M. The input is accepted iff M accepts $P(x)$.

Definition 3.6.4 *[11] A n-projection over Δ is a finite sequence of pairs (i, f), where $1 \leqslant i \leqslant n$ and f is a function from Σ to Δ, such pairs being called instructions. The length of this sequence is denoted S_n , and its j^{th} instruction is denoted $(B_n(j), E_n(j))$. A projection* *over Δ is a family $P = \{P_n\}_{n\in\mathbb{N}}$ of n-projections over Δ. We can consider P as a tuple $P = (\Sigma, \Delta, S, B, E)$, where S is a function from $\mathbb{N}$ to $\mathbb{N}$, B is a function from $\mathbb{N}\times\mathbb{N}$ to $\mathbb{N}$ and E is a function from $\mathbb{N}\times\mathbb{N}$ into $\Delta^\Sigma = \{f \mid \Sigma \to \Delta\}$. We write $P(x)$ for the concatenation of the $(E_{|x|}(j))(x_{B_{|x|}}(j))$ for $1 \leqslant j \leqslant S_{|x|}$.*

Thus, as x ranges over strings of length n, each bit of $P(x)$ depends on at most one bit of x.

In section 3.5.3, we are going to use nondeterministic branching programs in order to define counting for the class $BWBP$. Nondeterministic branching programs have the capability to branch nondeterministically without consuming an input bit.

Definition 3.6.5 *A nondeterministic branching program on the variable set $\{x_1, ..., x_n\}$ is a directed, acyclic graph with the same structure as a (usual) branching program, but which may additionally contain unlabelled nodes with two unlabelled outgoing edges. These nodes are called nondeterministic nodes (guessing nodes).*

Such a graph represents a Boolean function $f : \{0,1\}^n \to \{0,1\}$ in the following way. Let an input $x = (x_1, x_2, ..., x_n) \in \{0,1\}^n$ be given. We call an edge activated for x if it leaves a nondeterministic node or if it leaves a labelled node which is labelled by x_i. We define $f(x) = 1$ iff there is a path consisting of activated edges from the source to the accepting sink node (such a path is called accepting path).

3.6.2 Circuit definitions of decision and counting classes

Before defining the corresponding counting classes $\#NC^1$ and $\#BWBP$, we refer to the boolean circuit characterizations of NP and NL, given in [52]. By "arithmetizing their boolean circuits", we obtain the counting classes $\#P$ and $\#L$.

Theorem 3.6.2 *(i) NL = Uniform Skew Circuit SIZE($n^{O(1)}$).*
(ii) NP = Uniform Skew Circuit DEPTH($n^{O(1)}$).
(iii) NP = Uniform Circuit DEPTH,DEGREE($n^{O(1)}$; $n^{O(1)}$).

To arithmetize a boolean circuit, we propagate all NOT gates to the input level, convert OR gates to PLUS (+) gates, and convert AND gates to MULT ($\times$) gates. As a result, we obtain monotone arithmetic circuits which map naturally from the binary strings $\{0,1\}^*$ to the natural numbers.

Let B be a Boolean circuit of size s, depth d and degree D. Then there exists an arithmetic circuit A of size s, depth d and degree D, such that B has p accepting subtrees on an input x on which it evaluates to one if and only if A has value p on input x. Given a Boolean circuit B, the arithmetic circuit A is obtained in the way we mentioned in the previous paragraph, by replacing all the OR (AND) gates of B by PLUS (respectively, MULT) gates.

Definition 3.6.6 *The notion of an **accepting subtree** of a Boolean circuit given an input on which it evaluates to one is analogous to the notion of accepting subtrees of machines.*

Let B be a Boolean circuit, and let T(B) be its tree equivalent: The tree-equivalent of a graph is obtained by replicating vertices whose outdegree is greater than one until the resulting graph is a tree. Let x be an input on which B evaluates to one. An accepting subtree H of the circuit B on input x is a subtree of T(B) defined as follows:

- *H includes the output gate,*
- *for any AND gate v included in H, all the immediate predecessors of v in T(B) are included as its immediate predecessors in H,*
- *for any OR gate v included in H, exactly one immediate predecessor of v in T(B) is included as its only immediate predecessor in H, and*
- *any input vertex of T(B) included in H has value one as determined by the input x.*

It is easy to verify the fact that the circuit B evaluates to one given the input x if and only if there is an accepting subtree of T(B) on input x.

Definition 3.6.7 *A function* $f : \{0,1\}^* \to \mathbb{N}$ *is in* $\#Uniform\ Circuit\ SIZE, DEPTH, DEGREE(s(n); d(n); D(n))$ *if and only if there exists a uniform family* $\{G_n\}_{n\in\mathbb{N}}$ *of Boolean circuits of size* $\mathcal{O}(s(n))$*, depth* $\mathcal{O}(d(n))$ *and degree* $\mathcal{O}(D(n))$ *such that for all strings x of length n, $f(x)$ is the number of accepting subtrees of G_n on input x.*

The next lemma comes naturally if we consider our observations above.

Lemma 3.6.1 *For* $s(n)$, $D(n) = \Omega(n)$,
$\#Uniform\ Circuit\ SIZE, DEPTH, DEGREE(s^{\mathcal{O}(1)}(n); d(n); D(n)) =$
$Uniform\ Monotone\ Arithmetic\ Circuit\ SIZE, DEPTH, DEGREE(s^{\mathcal{O}(1)}(n); d(n); D(n))$.

The theorem below establishes the relationship between the number of accepting paths in nondeterministic Turing machines and the number of accepting subtrees of Boolean circuits.

Theorem 3.6.3 *For* $T(n) = \Omega(n)$,
$\#NTIME(T^{\mathcal{O}(1)}(n)) = \#Uniform\ Circuit\ DEPTH, DEGREE(T^{\mathcal{O}(1)}(n); T^{\mathcal{O}(1)}(n)) = Uniform\ Monotone\ Arithmetic\ Circuit\ DEPTH, DEGREE(T^{\mathcal{O}(1)}(n); T^{\mathcal{O}(1)}(n))$.

For the special case of $\#P$, it holds that $\#P = Uniform\ Monotone\ Arithmetic\ Circuit\ DEPTH, DEGREE(n^{\mathcal{O}(1)}; n^{\mathcal{O}(1)})$.

Similarly, arithmetizing uniform boolean scew circuits of polynomial size, we obtain $\#L$.

3.6.3 The classes #NC^1 and #BWBP

At this point, it has become clear that $\#NC^1$ can be defined in two ways.

Definition 3.6.8 *A function $f : \{0,1\}^n \to \mathbb{N}$ is in $\#NC^1$ if it can be computed by an arithmetic circuit of polynomial size, $\mathcal{O}(logn)$ depth and bounded fan-in.*

Equivalently, a function $f : \{0,1\}^n \to \mathbb{N}$ is in $\#NC^1$ if there exist a boolean NC^1 circuit C, such that $f(x)$ is the number of accepting subtrees (or proof trees) of C for input x [11].

Evaluating arithmetic formulae over $\mathbb{N}$ is complete for $\#NC^1$.

The class $\#BWBP$ can also be defined in two ways.

Definition 3.6.9 *A function $f : \{0,1\}^n \to \mathbb{N}$ is in $\#BWBP$ if there exist a nondeterministic branching program P of bounded width and polynomial length, such that $f(x)$ is equal to the number of accepting paths of P on input x.*

The following problem is $\#BWBP$-complete: Given a sequence of constant dimension matrices over the natural numbers $0, 1$, compute a specific entry in the matrix product.

The class $\#BWBP$ can equivalently be defined by considering ordinary (i.e., deterministic) branching programs which output a sequence of elements from a given alphabet: This sequence is then given as input to a nondeterministic finite automaton, and we count accepting paths in that NFA. Thus, let us define $\#BP\text{-}NFA$, the class of functions that “count” the number of accepting paths in a nondeterministic finite-state automaton when run on the projection of the input x, $P(x)$. Then $\#BWBP = \#BP\text{-}NFA$ [36].

As presented in Lemma 2 of [36], $\#BWBP = \#BP\text{-}NFA \subseteq \#NC^1 \subseteq FL$, where the last inclusion was proved in [13]. But it remains an open question if the classes $\#BWBP$ and $\#NC^1$ are equal. There has even been some speculation in the community that these two classes may really be different, since the techniques used to prove Barrington’s theorem do not seem to help in this setting.

For any counting class $\#C$, we can define the associated “Gap” class $GapC$. A function f is in $GapC$ iff there are two functions f_1, f_2 in $\#C$, such that $f = f_1 - f_2$. Thus, the class $GapNC^1$ can be characterized as the difference of two functions in $\#NC^1$. We also obtain the class $GapNC^1$ if we augment the arithmetic circuits of $\#NC^1$ by allowing the constant -1. Evaluating arithmetic formulae over $\mathbb{Z}$ is complete for $GapNC^1$. The authors of [11] did show that $GapNC^1$ is equal to the class of functions that are the difference of two $\#BWBP$ functions, i.e. $GapNC^1 = GapBWBP$.

Theorem 3.6.4 *(i) $\#BWBP \subseteq \#NC^1$.*
(ii) $GapNC^1 = GapBWBP$.

An important open question regarding arithmetic NC^1 is whether $GapNC^1$ (and hence $\#NC^1$) is equal to boolean NC^1. It is observed that $\#NC^1$ is at least as powerful as boolean NC^1. A hint that they might be the same class of functions is provided by the following theorem, which states that the class $\#NC^1$ is almost in NC^1.

Complexity Classes	Complete Problems
L	Graph Acyclicity, Tree Isomorphism
$NC^1 = BWBP$	Regular Sets, Boolean Formula Evaluation
$\#L$	$\#DFA$, $\#Path$
$GapL$	Determinant of Integer Matrices
$\#NC^1$	Counting accepting paths in VPA
$\#BWBP$	Counting accepting paths in NFA
$GapNC^1 = GapBWBP$	Evaluating Arithmetic Formula over $\mathbb{Z}$

Table 3.1: Complexity Classes and some of their complete problems

Theorem 3.6.5 *[29] Let $f \in GapNC^1$. Then f is computed by a family of Boolean circuits having bounded fan-in, polynomial size, and depth $\mathcal{O}(lognlog^*n)$, where log^*n is the number of times the logarithm function must be iteratively applied before the result is less than or equal to 1.*

Krebs, Limaye, and Mahajan prove a complete problem for $\#NC^1$ in [32] and relate the difference of $\#NC^1$ and $\#BWBP$ to the difference of their complete problems.

More formally, the following is proved in [11]: There is a fixed NFA N, such that any function f in $\#BWBP$ can be reduced to counting accepting paths of N. In particular, $f(x)$ equals the number of accepting paths of N on a word $g(x)$ that is a projection of x and is of size polynomial in the length of x.

There had been no similar characterization of $\#NC^1$ before [32] was published. The main result of [32] is this: There is a fixed VPA (visibly pushdown automaton) V, such that any function f in $\#NC^1$ can be reduced via projections to counting accepting paths of V. Moreover, any $\#VPA$ function ("counting" the number of accepting paths in any VPA) can be computed in $\#NC^1$.

Thus, the difference (if any) between $\#BWBP$ and $\#NC^1$, which is known to vanish with one subtraction, is captured exactly by the extension of NFA to VPA.

Visibly pushdown automata (VPA) are pushdown automata (PDA) with certain restrictions on their transition functions. There are no ϵ moves. The input alphabet is partitioned into call, return and internal letters. On a call letter, the PDA must push a symbol onto its stack, on a return letter it must pop a symbol, and on an internal move it cannot access the stack at all. While this is a severe restriction, it still allows VPA to accept non-regular languages (the simplest example is $a^n b^n$). At the same time, VPA are less powerful than all PDA; they cannot even check if a string has an equal number of a's and b's. In fact, due to the visible nature of the stack, membership testing for VPA is significantly easier than for general PDA; it is known to be in Boolean NC^1. In other words, as far as membership testing is concerned, VPA are no harder than NFA.

Chapter 4

Problems that characterize #L and span-L

4.1 #L and the determinant

One of the most important early results of complexity theory is the theorem of [54] showing that the complexity of computing the permanent of an integer matrix is characterized by the complexity class $\#P$. It is perhaps surprising that well over a decade passed before it was discovered that an equally-close connection exists between the complexity of computing the determinant of a matrix and the class $\#L$.

There are several efficient algorithms for computing the determinant of a matrix. If two matrices of order n can be multiplied in time $M(n)$, where $M(n) \geqslant n^{\alpha}$ for some $\alpha > 2$, then the determinant can be computed in time $\mathcal{O}(M(n))$ [8]. This means, for example, that an $\mathcal{O}(n^{2.376})$ algorithm exists based on the Coppersmith-Winograd algorithm for matrix multiplication.

Although stated in different ways, the results of [55], [56], [48] and [15] show the following fact:

Theorem 4.1.1 *[4] A function f is in GapL iff f is logspace many-one reducible to the determinant, i.e.* $Closure_{\leqslant^{l}_{m}}(INTDET) = GapL$.

We describe here the proof of Toda in [48], but not in detail.

In [12], Cook defined the class DET of functions that are NC^1-reducible to the integer determinant, $DET = Closure_{\leqslant_{NC^1}}(INTDET)$, and exhibited some linear algebraic problems that are complete for the class. For NC^1 reducibility we give the definition 4.1.3 below.

Toda studied the relationship between the integer determinant and some counting graph theoretic problems related to $\#L$. More precisely speaking, he showed that the integer determinant is interreducible to the problems of counting the followings: the number of paths between two nodes of a given acyclic directed graph, the number of shortest paths between two nodes of an undirected graph or a directed graph, the number of rooted spanning trees of a given directed graph, and the number of Eulerian cycles of a given Eulerian directed graph.

Theorem 4.1.2 *$DET = Closure_{\leqslant_{NC^1}}(INTDET) = Closure_{\leqslant_{NC^1}}(\#L)$. More precisely, $DET = Closure_{\leqslant_{AC^0}}(\#L)$.*

Note that it is unknown if $\#L$ is closed under AC^0 OR NC^1 reductions.

Thus, Toda proved that $\#L$ is computationally equivalent to DET under NC^1 -reducibility and hence that all complete functions for $\#L$ are also complete for DET. Toda's results may also be contrasted to the results by Valiant [49]. A similarity and a difference between the determinant and the permanent are translated into those between $\#P$-complete functions and the problems above. To state the contrasts on the above four problems: The problems of counting (1) the number of simple paths between two nodes of a directed graph not necessarily acyclic, (2) the number of all simple paths not necessarily shortest between two nodes of an undirected graph, (3) the number of all rooted trees not necessarily spanning of a directed graph, and (4) the number of all Hamiltonian paths in a directed graph are $\#P$-complete.

The results of [48] are partitioned into three groups depending on the reducibilities used. The first group includes some problems of which the integer determinant problem is a p-projection. The second group includes three counting problems to which the integer determinant problem is equivalent with respect to the polynomial-size and constant-depth truth-table reducibility. The third one includes a counting problem to which the integer determinant problem is equivalent with respect to P-uniform NC^1-reducibility.

Definition 4.1.1 *[43] Let the problems $F = \{f_n\}_{n\in\mathbb{N}}$, $G = \{g_n\}_{n\in\mathbb{N}}$, where f_n, g_n are boolean functions from $\{0,1\}^n$ to $\{0,1\}^m$. Then F is projection reducible to G, $F \leqslant_{proj} G$, if there is a function $p(n)$ bounded above by a polynomial in n and for each $f_n \in F$ a mapping $\sigma_n : \{y_1, ...y_{p(n)}\} \to \{x_1, \overline{x_1}, x_2, \overline{x_2}, ..., 0, 1\}$ such that $f_n(x_1, ..., x_n) = g_{p(n)}\big(\sigma_n(y_1), ..., \sigma_n(y_{p(n)})\big)$.*

Definition 4.1.2 *[14] Let the problems $F = \{f_n\}_{n\in\mathbb{N}}$, $G = \{g_n\}_{n\in\mathbb{N}}$, where f_n, g_n are boolean functions from $\{0,1\}^n$ to $\{0,1\}^m$. Then F is constant depth truth table reducible to G, $F \leqslant_{cd-tt} G$, if there is a polynomial $p(n)$ and a constant c such that each f_n is computed by a circuit of depth $\leqslant c$ and size $\leqslant p(n)$ containing "black boxes" which compute members g_j of G or their negations $\overline{g_j}$ with $j \leqslant p(n)$, where the size and depth of black boxes are counted as unity, and such that there is no path in the circuit from an output of one black box to an input of another black box.*

For NC^1-reducibility, one possible definition is to say that a set A is many-one $\leqslant_{NC^1}$-reducible to a set B if there is an NC^1 computable function f such that for all x, $x \in A$ iff $f(x) \in B$. However, here we are interested not just in sets, but in computing functions and solving problems, so the "Turing" (or "Cook") version of reducibility is most useful.

Definition 4.1.3 *[12] Let the problems $F = \{f_n\}_{n\in\mathbb{N}}$, $G = \{g_n\}_{n\in\mathbb{N}}$, where f_n, g_n are boolean functions from $\{0,1\}^n$ to $\{0,1\}^m$. Then F is (P-uniform) NC^1 reducible to G, $F \leqslant_{NC^1} G$, if there is a (P-uniform) family $\{C_n\}_{n\in\mathbb{N}}$ of NC^1 circuits for solving F, containing oracle nodes for G. An oracle node for G is a node with some sequence $< y_l, ..., y_j >$ of input edges and a sequence $< z_1, ..., z_l >$ of output edges whose values satisfy $g_j(y_l, ..., y_j) = (z_1, ..., z_l)$. For the purpose of defining depth in C_n, this oracle node counts as depth $log(j + l)$.*

Comment 4.1.1 *The notions of "black boxes" for G and oracle nodes for G in the definitions 4.1.2 and 4.1.3 respectively, describe the same thing.*

Here are the definitions of the problems mentioned in [48].

$INTDET$:
Input: A $n \times n$ matrix A of n-bit integer entries.
Output: The determinant of A.

$ZERO\text{-}ONE\ DET$: A restricted version of $INTDET$ where all entries of the matrix are 0 or 1.

$MATPOW_{\{-1,0,1\}}$:
Input: A $n \times n$ integer matrix A of entries either -1, 0, or 1.
Output: The $(1, n)^{th}$ entry of $\sum_{i=1}^{n} A^i$.

$N^{th}POWER_{-1,0,1}$:
Input: A $n \times n$ integer matrix of entries either -1, 0, or 1.
Output: The $(1, n)^{th}$ entry of the n^{th} power of the matrix.

$DIRECTED\ PATH\ DIFFERENCE$:
Input: Two monotone directed graphs G and H with n nodes (where a directed graph is called monotone if it contains no edge (i, j) such that $j \leqslant i$).
Output: $f_{\#path}(G) - f_{\#path}(H)$.

$DIRECTED\ PATH$:
Input: A monotone directed graph G with n nodes.
Output: The number of paths in G from the first node to the n^{th} node.

$SHORTEST\ PATH$:
Input: An undirected graph G with n nodes.
Output: The number of shortest paths in G between the first node and the n^{th} node (where we define the length of a path as the number of edges on the path).

$ROOTED\ SPANNING\ TREE$:
Input: A directed graph G.

Output: The number of rooted spanning trees of G (where a rooted spanning tree is a directed spanning tree such that every vertex except the root has in-degree 1, while the root has in-degree 0).

DIRECTED EULERIAN PATH:
Input: A directed graph G.
Output: The number of Eulerian paths in G.

In the first place, Toda proves the following sequence of $\leqslant_{proj}$-reductions:

$INTDET \leqslant_{proj} MATPOW_{\{-1,0,1\}} \leqslant_{proj} N^{th}POWER_{\{-1,0,1\}} \leqslant_{proj}$ $DIRECTED\ PATH\ DIFFERENCE \leqslant_{proj} ZERO\text{-}ONE\ DET$.

This fact means that the problems above are equivalent under projection reductions.

The second important theorem of Toda states that $INTDET$ is $\leqslant_{cd-tt}$-reducible to each of the following problems: $DIRECTED\ PATH$, $SHORTEST\ PATH$, $ROOTED\ SPANNING\ TREE$ and vice versa.

At the end Toda shows that $DIRECTED\ EULERIAN\ PATH \leqslant_{cd-tt} INTDET$ and $INTDET \leqslant_{NC^1} DIRECTED\ EULERIAN\ PATH$.

The proof of the first result gives a proof of the Theorem 4.1.1. The rest of Toda's theorems relate the complexity of the determinant to the complexity of known algebra or graph problems.

4.2 #DNF: A problem in span-L with FPRAS

Although DNF, the decision version of the problem, belongs to L, it is known that $\#DNF$ is $\#P$-complete with respect to log-space metric reductions.

Theorem 4.2.1 *(i) $DNF \in L$.*
(ii) $\#DNF$ is complete for $\#P$ with respect to log-space metric reductions.

Proof. (i) Let a DNF formula F with variables $x_1, ..., x_n$. A deterministic log-space machine M reads F's clauses from the first to the last one until it finds a clause such that for all $i \in \{0, ..., n\}$, it does not contain both x_i and $\overline{x_i}$. If M finds a clause with this property, it halts accepting, otherwise M halts rejecting. Checking the above property can be done in logarithmic space.

(ii) In section 2.2 we defined the problem $\#3SAT$. Here, we need the function $\#SAT$ which on input a boolean formula F in CNF outputs the number of its satisfying assignments, $\#F$. We reduce $\#SAT$ to $\#DNF$. Consider a CNF formula F. Let $G = \overline{F}$. Note that G can be easily expressed in DNF with the same size as F. Now any assignment that satisfies F does not satisfy G and vice versa. Thus we have $\#F = 2^n - \#G$, where n is the number of the variables in F. Thus, solving $\#G$ gives an answer to $\#F$. It is clear that we described a log-space metric reduction. ■

According to the results of section 2.3, $\#DNF$ is also *span-L* - complete with respect to log-space metric reductions. It is unlikely that $\#DNF$ is *span-L* complete under log-space functional many-one reductions. This would mean that the decision versions of all *span-L* problems could be decided in L.

It is also well known that $\#DNF$ admits an FPRAS [33]. An interesting question would be "Is there any *span-L* - complete problem with respect to log-space functional many-one reductions which admits an FPRAS?". If this question is answered in the affirmative then all the problems that belong to *span-L* have an FPRAS.

In the rest of this section we present the FPRAS for $\#DNF$.

Definition 4.2.1 *A randomised algorithm is called an unbiased estimator for a function f if on input n, the expected value of the output is $f(n)$.*

Definition 4.2.2 *Consider an input F of $\#DNF$ and the problem of approximating $\#F$. An ϵ, δ approximation algorithm for the $\#DNF$ problem is a Monte-Carlo algorithm which on every input formula F, $\epsilon > 0$, $\delta > 0$, outputs a number $\tilde{Y}$ such that* $Pr[(1-\epsilon)\#F \leqslant \tilde{Y} \leqslant (1+\epsilon)\#F] \geqslant 1-\delta$.

Our goal is a fully polynomial randomized approximation scheme, known as an FPRAS. Given an input $\#F$, an error parameter $\epsilon > 0$ and a confidence parameter $0 < \delta < 1$, our goal is to compute $\tilde{Y}$ as in Definition 4.2.2 in time polynomial in $|F|$, ϵ^{-1} and $log(1/\delta)$.

The first algorithms can be thought as variants of the following abstract Monte-Carlo algorithm. We have a finite set U of known size and an efficient method of randomly choosing elements from U such that each $u \in U$ is equally likely to be chosen. We have a function f defined on U such that $f(u)$ is either 0 or 1 at each u and we have an efficient method for computing $f(u)$ given u. Our goal is to estimate $|G|$ where G is the subset of U at which the function f takes on the value 1. One trial of the Monte-Carlo algorithm consists of randomly choosing $u \in U$, computing $f(u)$, and $Y = |U| \cdot f(u)$. The Monte-Carlo algorithm consists of running N trials, where Y_i is the value of Y from the *i-th* trial. The output of the algorithm is $\tilde{Y} = \sum_{i=1}^{N} \frac{Y_i}{N}$.

For each of the previous results it holds that $E[Y_i] = Pr\big(f(u)=1\big) \cdot |U| + Pr\big(f(u)=0\big) \cdot 0 = \frac{|G|}{|U|} \cdot |U| = |G|$ and $E[\tilde{Y}] = |G|$. Thus, we have an unbiased estimator for our problem. The number of times we have to repeat this procedure in order to have an ϵ, δ approximation algorithm is given by the following theorem.

Theorem 4.2.2 *(Zero-One Estimator) Let $\mu = \frac{|G|}{|U|}$ and $\epsilon \leqslant 2$. If $N \geqslant (1/\mu)\cdot(4\cdot ln(2/\delta)/\epsilon^2)$ then the Monte-Carlo algorithm described above is an ϵ, δ approximation algorithm.*

To obtain an FPRAS for the DNF counting problem it is sufficient to design an algorithm that has the following properties:

1. In polynomial time we can compute $|U|$ and we can randomly choose members from U with the uniform probability distribution.
2. The function f is computable in polynomial time.
3. We can compute in polynomial time an upper bound on $B = (1/\mu) = |U|/|G|$ such that this value is polynomial in the length of F.

If we can achieve these goals, then the polynomial time ϵ, δ approximation algorithm consists of computing B and running $N = B \cdot (4 \cdot ln(2/\delta)/\epsilon^2)$ trials.

A first idea that doesn't work: Let U be the set of all possible truth assignments for the n variables. Let f be the function which evaluates to 1 on the set of truth assignments which satisfy F. Naturally, G is the set of truth assignments which satisfy F and the quantity we want to estimate is $|G|$. It is easy to verify that the first and second properties are true with these definitions of U and f, but $|U|/|G|$ might be exponentially large in the size of the formula. This algorithm is an exponential time algorithm for the $\#DNF$ problem!

We present an ϵ, δ approximation algorithm for the DNF counting problem based on the coverage algorithm for the union of sets presented in Section 3 of [33]. The running time of this algorithm is $\mathcal{O}(nm^2 \cdot ln(1/\delta)/\epsilon^2)$. This can be achieved by "decreasing" the size of the universe U.

Let D_i be the set of truth assignments which satisfy clause C_i. Let $D = \bigcup_{i=1}^{m} D_i$. It is easy to verify that $\#F = |D|$. Let U be the disjoint union of the D_i, i.e. $U = D_1 \uplus ... \uplus D_m$. An element in U is represented by a pair (s, i), where $1 \leqslant i \leqslant m$ and $s \in D_i$. Then U contains only the satisfying assignments. However, if an assignment s satisfies k clauses, U contains k copies of s. Thus $|U| = \sum_{i=1}^{m} |D_i| \geqslant |D|$.

For each $s \in D$, define the coverage set of s to be $cov(s) = \{(s, i) : (s, i) \in U\}$. The coverage sets define a partition on U, where the size of each coverage set is at most m (each satisfying assignment satisfies at most m clauses) and the number of coverage sets is exactly $|D|$ (exactly one coverage set corresponds to exactly one satisfying assignment). Each coverage set contains information about which clauses become true by s.

We define $f(s, i) = 1$ if i is the smallest index such that $s \in D_i$ and $f(s, i) = 0$ otherwise. Since, for each $s \in D$, f takes on the value 1 for exactly one element of $cov(s)$, the size of the subset of U for which f takes on the value 1 is equal to $|D|$. This definition means that $f(s, i)$ takes on the value 1 if C_i is the first clause satisfied by s.

To implement the algorithm, we need to define the sets D_i so as to have the following properties:

- For all $i \in \{1, 2, ..., m\}$, $|D_i|$ can be easily computed. Obviously, the number of satisfying assignments of the clause C_i is equal to $|D_i| = 2^{n - \#variables\ in\ C_i}$.

- For all $i \in \{1, 2, ..., m\}$, we can randomly choose an element $s \in D_i$ such that the probability of choosing each such s is $1/|D_i|$. This is done by setting the truth values for the variables which appear in C_i to satisfy clause C_i and then choosing the truth values for the variables that do not appear in C_i randomly to be "true" or "false" each with probability $\frac{1}{2}$.

- Given any $s \in D$ and any $i \in \{1, 2, ..., m\}$, it is easy to decide whether or not $s \in D_i$. Simply by determining the truth values of the variables of the clause C_i according to s one can check if this clause is satisfied or not.

After all, the above procedures can be done in polynomial time.

A trial proceeds as described in the beginning: Choose (s, i) uniformly at random from U and set $Y = |U| \cdot f(s, i)$. In more detail:

1st *step*: Randomly choose $i \in \{1, 2, ..., m\}$ such that i is chosen with probability $|D_i|/|U|$.

2nd *step*: Randomly choose $s \in D_i$ such that s is chosen with probability $1/|D_i|$.

(Note. Steps 1 and 2 randomly choose $(s, i) \in U$ with probability $1/|U|$.)

3rd *step*: Compute $f(s, i)$.

4th *step*: $Y = f(s, i) \cdot |U|$.

The 2nd step was discussed as one of the properties that the sets D_i have.

The 1st step can also be done in polynomial time. Let A be an integer array of length $m + 1$ such that $A_0 = 0$ and $A_i = \sum_{j=1}^{i} |D_i|$ (then $A_m = |U|$). Choose uniformly at random a number r between 1 and $|U|$ and use binary search to find the entry in A such that $A_{i-1} < r \leqslant A_i$. The entry A_i and the index i are chosen with probability $\frac{A_i - A_{i-1}}{|U|} = \frac{|D_i|}{|U|}$.

Finally, the computation of $f(s, i)$ require polynomial time. Let $j = min\{l : s \in D_l\}$. The value of j is computed by indexing sequentially through the clauses and checking whether s satisfies C_l. Then $f(s, i) = 1$ iff $i = j$.

The total time of the preprocessing step of the algorithm (computing all the $|D_i|$ and the array A) is $\mathcal{O}(mn)$. The total time for steps 1 and 2 is $\mathcal{O}(n)$ and for step 3 is $\mathcal{O}(mn)$. The advantage we have with the coverage algorithm over the naive algorithm is that $\frac{1}{\mu} = \frac{|U|}{|D|} \leqslant \frac{m \cdot |D|}{|D|} = m$, where the last inequality holds because each element of D appears at most m times in U (a satisfying assignment s satisfies at most m clauses). Thus applying the Zero-One Estimator Theorem, $N = m \cdot 4 \cdot ln(2/\delta)/\epsilon^2$ trials suffice for an ϵ, δ approximation algorithm. The total time for preprocessing plus all trials of the algorithm is $\mathcal{O}(nm^2 \cdot ln(1/\delta)/\epsilon^2)$.

4.3 The complexity of the span-L - complete problem #NFA

In this section we will study the complexity of a version of the problem $\#NFA$, which we defined in the section 2.1.

$\#NFA_m$:
Input: An encoding of a NFA M and an integer m in unary.
Output: The number of words of length exactly m accepted by M, i.e. $|L(M) \cap \Sigma^m|$.

This problem has a variety of applications in computational biology. In addition, there has been considerable theoretical interest in the complexity of counting and random generation.

We have already seen that $\#NFA$ is *#P-complete* and it is easily shown that $\#NFA_m$ is *#P-complete*. The results we present are based on the work of Jerrum, Valiant and Vazirani [30] and the work of Karp and Luby [34] and they can be found in [35].

Theorem 4.3.1 *$\#NFA_m$ is $\#P$-complete with respect to log-space metric reductions.*

Proof. There is a simple reduction from $\#DNF$ to $\#NFA_m$ (and it is a reduction from $\#DNF$ to $\#NFA$ at the same time).

Let F be a boolean formula in disjunctive normal form. We construct an NFA whose language is exactly the satisfying assignments of F. Let $x_1, ..., x_m$ and $t_1, ..., t_l$ be the variables and the terms respectively of F. For each t_i we construct an NFA M_i. This machine will consist of a chain of $m+1$ states, $s_{0i}, ..., s_{mi}$, where the state s_{0i} is the start state and s_{mi} the accepting state of this automaton. The edge $< s_{j_i}, s_{j+1_i} >$ is labelled 1 if x_{j+1} occurs in t_i, 0 if $\overline{x}_{j+1}$ occurs in t_i and 0, 1 otherwise. Clearly, M_i accepts exactly those strings corresponding to truth assignments that satisfy term t_i. Now let M be the NFA with a start state s and an ϵ-transition to the start state s_{0i} for each $1 \leqslant i \leqslant l$. The final states are exactly the final states of each M_i. Thus, M accepts exactly the strings corresponding to satisfying assignments of F and $\#NFA_m(M, m) = \#DFA(F)$. ■

Comment 4.3.1 *(i) The witnesses of an NP language can be modelled as satisfying assignments of a CNF formula and the non-witnesses as the satisfying assignments of a DNF formula. Because of the construction in the previous proof, the non-witnesses of an NP language can be modelled as a regular language in which the accepting NFA makes all of its nondeterministic moves in the first step.*

(ii) The $n^{\mathcal{O}(logn)}$ -ras given in [35] is a generalization of the FPRAS for $\#DNF$ described in Section 4.2.

Similarly to the Definition 4.2.1 of the previous section we give the following definition of a $g(n)$-ras.

Definition 4.3.1 *A $g(n)$-randomised approximation scheme for a non-negative real-valued function f is a probabilistic algorithm which, on input x and $\epsilon > 0$ and $\delta < 1$ computes $\tilde{f}(x)$ where $Pr[f(x) \cdot (1+\epsilon)^{-1} \leqslant \tilde{f}(x) \leqslant f(x) \cdot (1+\epsilon)] \geqslant 1-\delta$.*
Further, the algorithm runs in time $\mathcal{O}(g(n))$ where n is the maximum of $|x|$, ϵ^{-1} and $log\delta^{-1}$.

Under a reasonable assumption of self-reducibility on an NP set, it is known that the problem of randomised approximate counting is polynomial time equivalent to the problem of almost uniform generation. Let R be any polynomial-time computable binary relation. For any x, the counting problem is to count, or approximately count the number of y such that $(x, y) \in R$. The associated generation problem is, on input x, to generate uniformly at random, an element of the set $\{y \mid (x, y) \in R\}$.

Definition 4.3.2 *Let R be a polynomial-time computable binary relation. For any x, let $\phi(x) = \{y : (x, y) \in R\}$. A $g(n)$-almost uniform generator for the relation R is a probabilistic algorithm, A, which, on input x, $\epsilon > 0$ outputs a $y \in \phi(x)$ such that*
$|\phi(x)|^{-1}(1+\epsilon)^{-1} \leqslant Pr(A \text{ outputs } y) \leqslant |\phi(x)|^{-1}(1+\epsilon)$.
Also, A runs in time $\mathcal{O}(g(n))$ where n is the maximum of $|x|$ and $log\epsilon^{-1}$.

Jerrum, Valiant and Vazirani showed in [30] the following important fact: If R is a self-reducible polynomial-time computable relation, then there exists a polynomial-time randomised approximation scheme if and only if there exists a polynomial-time almost uniform generator. We give a $n^{\mathcal{O}(logn)}$-ras for $\#NFA_m$ and a $n^{\mathcal{O}(logn)}$- almost uniform generator for $R = \{(M, x) : M \text{ is an } NFA,\ x \in L(M) \text{ and } |x| = m\}$ based on this fact.

A first attempt should consist of finding an unbiased estimator for $\#NFA_m$ with the hope its standard deviation to be small and hence the estimator could be used to obtain a polynomial time approximation to $\#NFA_m$.

In the first place we describe one of the two unbiased estimators of [35].

Given an NFA (without ϵ transitions) and a number m in unary:

1. Count the number of accepting paths of length m, $A(m)$.
2. Uniformly generate an accepting path of length m in the NFA. Let the string labelling the path be s.
3. Count the number of accepting paths labelled by s, $A(m, s)$.
4. Output the estimate $A(m)/A(m, s)$ for $\#NFA_m$.

In step 2 the probability that a string s is generated is $A(m, s)/A(m)$ and when s is generated the output is $A(m)/A(m, s)$. Thus, the expected value of the output is $\sum_s[A(m)/A(m, s) \cdot A(m, s)/A(m)]$ which is the number of the accepted words of length m.

Comment 4.3.2 *(i) The number of accepting paths of length m of an NFA can be computed with dynamic programming in the same way the number of accepting paths of a DFA is computed: If $f(q,i) = the\ number\ of\ paths\ of\ length\ i\ from\ q_0\ to\ q,\ then\ f(q,i+1) = \sum_p f(p,i)$ where the sum is taken over all $p \in Q$ and all $a \in \Sigma$ such that $\delta(p,a) = q$. Obviously, $f(q_{acc}, m)$ is the number of accepting paths of length m. Of course this number overcounts the number of accepted words of an NFA. With slight modification, this dynamic programming algorithm can be used to count the number of accepting paths of a particular string s of an NFA: If $f(q,i) = the\ number\ of\ paths\ labelled\ by\ the\ first\ i\ symbols\ of\ s,$ then $f(q,i+1) = \sum_p f(p,i)$ where the sum is taken over the $p \in Q$ such that $\delta(p,a) = q$ and a is the $i+1^{th}$ symbol of s. The value $f(q_{acc}, m)$ is the number of accepting paths of the string s. Here we suppose that the NFA has one starting state q_0 and one accepting state q_{acc}.*

(ii) Unfortunately the standard deviation of this experiment can be rather large.

We present the algorithms for approximate counting and almost uniform generation of [35]. The input to these algorithms is some $< M, 1^m >$, where M is a m-level NFA with binary alphabet and no ϵ-transitions.

Definition 4.3.3 *A m-level NFA is an NFA in which the states can be partitioned into $m+1$ levels with the following properties:*

- *There is exactly one state at level 0 and it is the starting state.*
- *There is exactly one state at level m and it is the accepting state.*
- *All transitions are from a node in level i to a node in level $i+1$ for $i \in \{0, ..., m-1\}$.*
- *For any state, p, the accepting state is reachable from p and p is reachable from the starting state.*

Lemma 4.3.1 *For any NFA M and integer m, there exists an NFA M' such that M' is a m-level NFA with no ϵ-transitions, a binary alphabet and such that $|L(M')| = |L(M) \cap \Sigma^m|$. Further, the size of M' is polynomial in the size of M and m.*

Comment 4.3.3 *If the NFA M of the previous lemma has $n = |Q|$ states then each level of the NFA M' has at most n states.*

Let M be a m-level NFA with at most n states per level and let $L = L(M)$. We write $\Delta(q,x)$ to denote the set of states reachable from q on string x. Let $q_1, ..., q_j$ be the states at level $\lfloor \frac{m}{2} \rfloor$ in M. For each state q of them define L_q as the strings in L that have accepting paths in M through state q.

Let $L_{q,P}$ be the prefixes of L_q of length $\lfloor \frac{m}{2} \rfloor$, i.e. $L_{q,P} = \{x : q \in \Delta(q_{start}, x)\}$ and $M_{q,P}$ the $\lfloor \frac{m}{2} \rfloor$-level NFA that consists of all the states from which q is reachable. Similarly, let $L_{q,S}$ be the suffixes of L_q of length $\lceil \frac{m}{2} \rceil$ and $M_{q,S}$ the corresponding NFA.

Assume that we know $|L_{q,P}|$ and $|L_{q,S}|$ exactly and that we can generate strings uniformly at random from these languages. Then $|L_q| = |L_{q,P}| \cdot |L_{q,S}|$. To generate an element of L_q, we generate an element x from $L_{q,P}$ and y from $|L_{q,S}$ and output xy. Then xy is uniformly drawn from L_q.

Now we want to compute $|L| = |\bigcup_{i=1}^{j} L_{q_i}|$, where the sets L_{q_i} are not necessarily disjoint. For $i = 1, ..., j$, let p_i be the probability that a random string of L_{q_i} is not in $\bigcup_{k=1}^{i-1} L_{q_k}$. Then we compute the size of L as follows: $|L| = \sum_{i=1}^{j} p_i \cdot |L_{q_i}|$.

The algorithms described above, are given in detail here:

The algorithm to generate a random string in L is as follows:

1. Randomly choose a state q_i from level $\lfloor \frac{m}{2} \rfloor$ with probability $\frac{|L_{q_i}|}{\sum_{l=1}^{j} |L_{q_l}|}$.
2. Randomly generate a string x from L_{q_i}. If x has paths through k different states at level $\lfloor \frac{m}{2} \rfloor$, output x with probability $\frac{1}{k}$.
3. If in step 2 we fail to output x repeat steps 1 and 2 until a string is output.

The algorithm to generate a random string in L is as follows:

1. For each state q_i in level $\lfloor \frac{m}{2} \rfloor$ approximately count L_{q_i}.
2. For each state q_i in level $\lfloor \frac{m}{2} \rfloor$ uniformly generate k strings independently from l_{q_i}.
3. For each i estimate the probability p_i as the fraction of the k strings generated from L_{q_i} that are not in $\sum_{l=1}^{i-1} |L_{q_l}|$.
4. Output $\sum_i p_i \cdot |L_{q_i}|$.

The running time analysis leads to the bound $n^{\mathcal{O}(logm)}$ for approximately counting the number of strings accepted by a m-level NFA with n states per level.

In [42] was proved the following theorem.

Theorem 4.3.2 *Let $R \subseteq \Sigma^* \times \Sigma^*$ be self-reducible. If there exists a polynomially time-bounded randomised approximate counter for R within ratio $1 + \mathcal{O}(n^{\alpha})$ for some $\alpha \in \mathbb{R}$, then there exists a fully polynomial randomised approximation scheme for #R.*

Approximate counting is robust with respect to polynomial time computation for self-reducible counting functions.

For the $\#NFA_m$ problem we have the following result which is slightly different and it does not require the self-reducibility property.

Theorem 4.3.3 *If $\#NFA_m$ can be approximated within a factor of 2^{n^δ} in polynomial time for some constant $0 \leqslant \delta < 1$, then $\#NFA_m$ can be approximated within a factor of $1+\epsilon$ for any constant $\epsilon > 0$ in polynomial time.*

The above theorem can be used to bootstrap the approximate counting algorithm so that achieves an approximation factor of $(1+\epsilon)$ for any ϵ that is inverse polynomial.

For further analysis we refer the reader to [35].

Chapter 5

Descriptive Complexity of counting functions

Descriptive complexity is an attempt to understand computation from the point of view of logic. It characterizes complexity classes by the type of logic needed to express the languages in them. This connection between complexity and the logic of finite structures allows results to be transferred easily from one area to the other and provides additional evidence that the main complexity classes are somehow "natural" and not tied to the specific abstract machines used to define them.

Specifically, each logical system produces a set of queries expressible in it. The queries when restricted to finite structures correspond to the computational problems of traditional complexity theory.

The first main result of descriptive complexity was Fagin's theorem, shown by Ronald Fagin in 1974. It established that NP is precisely the set of languages expressible by sentences of existential second-order logic [25] [58].

5.1 Preliminary definitions of first-order logic

We first review basic notions of first-order logic.

Definition 5.1.1 *A vocabulary $\tau =< R_1^{\alpha_1}, ..., R_r^{\alpha_r}, c_1, ..., c_s, f_1^{\beta_1}, ..., f_t^{\beta_t} >$ is a tuple of relation symbols, constant symbols and function symbols. R_i is a relation symbol of arity α_i and f_j is a function symbol of arity β_j.*

A structure with vocabulary τ is a tuple $\mathcal{A} =< |\mathcal{A}|, R_1^{\mathcal{A}}, ..., R_r^{\mathcal{A}}, c_1^{\mathcal{A}}, ..., c_s^{\mathcal{A}}, f_1^{\mathcal{A}}, ..., f_t^{\mathcal{A}} >$ whose universe is the nonempty set $|\mathcal{A}|$. For each relation symbol R_i of arity α_i in τ, $\mathcal{A}$ has a relation $R_i^{\mathcal{A}}$ of arity α_i defined on $|\mathcal{A}|$, i.e. $R_i^{\mathcal{A}} \subseteq |\mathcal{A}|^{\alpha_i}$. For each constant symbol $c_j \in \tau$, $\mathcal{A}$ has a specified element of its universe $c_j^{\mathcal{A}} \in |\mathcal{A}|$. For each function symbol $f_i \in \tau$, $f_i^{\mathcal{A}}$ is a total function from $|\mathcal{A}|^{\beta_i}$ to $|\mathcal{A}|$. A vocabulary without function symbols is called a relational vocabulary. The notation $||\mathcal{A}||$ denotes the cardinality of the universe of $\mathcal{A}$.

Example 5.1.1 *(i) The tuple $\tau_g =< E^2, s, t >$ is the vocabulary of graphs with specified source and terminal nodes. The symbol E is a relational symbol of arity 2 for expressing "edge relation" and s, t are constant symbols for the source and terminal respectively.*

(ii) The tuple $\sigma =<\leqslant^2, S^1>$ is the vocabulary of boolean strings. The $\leqslant$ represents the usual ordering on natural numbers and the relation S represents the positions where a binary string has one. Consider the binary string $w = 01101$. We can code w as the structure $\mathcal{A}_w =< \{0, 1, ..., 4\}, \leqslant, \{1, 2, 4\} >$ of vocabulary σ.

Comment 5.1.1 *In the history of mathematical logic most interest has concentrated on infinite structures. Yet, the objects computers have and hold are always finite. To study computation we need a theory of finite structures. That's why we concentrate on finite structures. We define $STRUC[\tau]$ to be the set of finite structures of vocabulary τ.*

Other symbols of first-order logic are: • the variables $x, y, z, ...$ • $\wedge$ ("and") • $\vee$ ("or") • $\neg$ ("not") • $\rightarrow$ (implication) • the symbols , () (comma and parentheses) • $\exists$ ("there exists") and $\forall$ ("for all").

Definition 5.1.2 *For any vocabulary τ, define the first-order language $\mathcal{L}(\tau)$ to be the set of formulas built up from the relation, constant and function symbols of τ, the logical relation symbol $=$, the boolean connectives $\wedge$, $\vee$, $\rightarrow$, $\neg$, variables: $VAR = \{x, y, z, ...\}$ and quantifiers $\exists$ and $\forall$.*

We say that an occurrence of a variable x in ϕ is bound if it lies within the scope of a quantifier $(\exists x)$ or $(\forall x)$, otherwise it is free. We write $\phi(x_1, ..., x_n)$ to denote that the free variables of the formula ϕ belong to $\{x_1, ..., x_n\}$.

A formula without free variables is called a sentence.

Comment 5.1.2 *For the inductive definitions of terms, atomic formulas and well formed formulas one could consult [18].*

We write $\mathcal{A} \models \phi$ to mean that $\mathcal{A}$ satisfies ϕ, i.e., that ϕ is true in $\mathcal{A}$. Since ϕ may contain some free variables, we will let an interpretation into $\mathcal{A}$ be a map $i : V \rightarrow |\mathcal{A}|$ where V is some finite subset of VAR. For convenience, for every constant symbol $c \in \tau$ and any interpretation i for $\mathcal{A}$, we let $i(c) = c^{\mathcal{A}}$. If τ has function symbols, then the definition of i extends to all terms via the recurrence, $i(f_j(t_1, ..., t_{r_j})) = f_j^{\mathcal{A}}(i(t_1), ..., i(t_{r_j}))$.

We have an inductive definition for what it means for a sentence ϕ to be true in a structure $\mathcal{A}$.

Definition 5.1.3 *(Definition of Truth)*

Let $\mathcal{A} \in STRUCT[\tau]$ be a structure, and let i be an interpretation into $\mathcal{A}$ whose domain includes all the relevant free variables. We inductively define whether a formula $\phi \in \mathcal{L}(\tau)$ is true in $(\mathcal{A}, i)$:

- $(\mathcal{A}, i) \models t_1 = t_2 \Leftrightarrow i(t_1) = i(t_2)$

- $(\mathcal{A}, i) \models R_j(t_1, ..., t_{r_j}) \Leftrightarrow < i(t_1), ..., i(t_{r_j}) > \in R_j^{\mathcal{A}}$
- $(\mathcal{A}, i) \models \neg\phi \Leftrightarrow$ *it is not the case that* $(\mathcal{A}, i) \models \phi$
- $(\mathcal{A}, i) \models \phi \wedge \psi \Leftrightarrow (\mathcal{A}, i) \models \phi$ *and* $(\mathcal{A}, i) \models \psi$
- $(\mathcal{A}, i) \models (\exists x)\phi \Leftrightarrow$ *(there exists* $\alpha \in |\mathcal{A}|)(\mathcal{A}, i, \alpha/x) \models \phi$*, where*

$$(i, \alpha/x)(y) := \begin{cases} i(y) & \text{if } y \neq x, \\ \alpha & \text{if } y = x. \end{cases}$$

We write $\mathcal{A} \models \phi$ *to mean that* $(\mathcal{A}, \emptyset) \models \phi$*.*

Comment 5.1.3 *The truth of sentences containing* $\vee$*,* $\rightarrow$ *and* $\forall$ *is defined naturally following the previous definition.*

[18] is strongly recommended for a good revision of the basic first-order logic notions.

5.2 Fagin's Theorem

Second-order logic consists of first-order logic plus new relation variables over which we may quantify. For example, the formula $(\forall R^r)\phi$ means that for all choices of relation R of arity r, ϕ holds.

Definition 5.2.1 *Let* $\mathbf{K}$ *be a class of finite structures over* $\mathcal{L}(\tau)$ *which is closed under isomorphism. Then* $\mathbf{K}$ *is (first-order) definable if it is of the form* $Mod(\phi) = \{\mathcal{A} : \mathcal{A}$ *is an* $\mathcal{L}(\tau)$ *structure and* $\mathcal{A} \models \phi\}$ *for some first-order sentence* ϕ*. Let FO be the set of (first-order) definable classes.*
Similarly, we say that $\mathbf{K}$ *is definable by a second-order sentence if* $\mathbf{K} = Mod(\phi) = \{\mathcal{A} : \mathcal{A}$ *is an* $\mathcal{L}(\tau)$ *structure and* $\mathcal{A} \models \phi\}$ *for some second-order sentence* ϕ*. Let SO be the set of second-order definable classes.*
Finally SO$\exists$ *(resp. SO*$\forall$*) is the class of existential (resp. universal) second-order definable classes, i.e. the sentence of the above definition can only contain existential (resp. universal) quantification over relational symbols.*

Example 5.2.1 *(i) Examples of definable classes are the class of all graphs and the class of all groups. A property of structures is said to be expressible in first order logic (or some other logic) if it determines a definable class. For example the property of a graph being complete is expressible in first order logic by the sentence* $\forall x \forall y < x, y > \in E$*.*

(ii) A natural way to encode a formula ϕ *in CNF is via the structure* $\mathcal{A}_\phi = (A, P, N)$*, where* A *is a set of clauses and variables, the relation* $P(c, v)$ *means that variable* v *occurs positively in clause* c *and* $N(c, v)$ *means that* v *occurs negatively in* c*.*
SAT is the set of boolean formulae in CNF that admit a satisfying assignment. This property

is expressible in $SO\exists$ as follows:
$\Phi_{SAT} = (\exists S)(\forall x)(\exists y)((P(x,y) \wedge S(y)) \vee ((N(x,y) \wedge \neg S(y)))$.
Φ_{SAT} asserts that there exists a set S of variables - the variables that should be assigned true - that is a satisfying assignment of the input formula.

Definition 5.2.2 *A query is any mapping $I : STRUC[\sigma] \rightarrow STRUC[\tau]$ that is polynomially bounded. A boolean query is a map $I_b : STRUC[\sigma] \rightarrow \{0,1\}$. A boolean query may also be thought of as a subset of $STRUC[\sigma]$ - the set of structures $\mathcal{A}$ for which $I_b(\mathcal{A}) = 1$.*

Everything that a Turing machine does may be thought of as a query from binary strings to binary strings. In order to make Descriptive complexity rich and flexible it is useful to consider other vocabularies. To relate structures over other vocabularies to Turing machine complexity, we fix a scheme that encodes the structures of some vocabulary τ as boolean strings. To do this, for each τ, we define an encoding $bin_\tau : STRUC[\tau] \rightarrow STRUC[\tau_s]$, where $\tau_s =< S^1 >$ is the vocabulary of boolean strings. The details of the encoding are not important, but it is useful to know that for each τ, bin_τ (and its inverse) are first-order queries. This means that bin_τ is given by a tuple of formulas from $\mathcal{L}(\sigma)$ (and its inverse is given by a tuple of formulas from $\mathcal{L}(\tau)$). However, the encoding $bin_\tau(\mathcal{A})$ does presuppose an ordering on the universe. There is no way to code a structure as a string without an ordering. Since a structure determines its vocabulary, in the sequel we usually write $bin(\mathcal{A})$ instead of $bin_\tau(\mathcal{A})$.

Definition 5.2.3 *Let a Turing machine M, a vocabulary τ and $\mathbf{Q} \subseteq STRUC[\tau]$. We say that M computes $\mathbf{Q}$ if for every $\mathcal{A} \in STRUC[\tau]$, M on input $bin_\tau(\mathcal{A})$ halts and accepts if and only if $\mathcal{A} \in \mathbf{Q}$.*

Proposition 5.2.1 *Let $\mathbf{L}$ be an isomorphism closed class of finite structures over a vocabulary τ.*
If $\mathbf{L}$ is definable by a first-order sentence, then it belongs to the complexity class L, i.e. $FO \subseteq L$.

Theorem 5.2.1 *(Fagin [20])*
Let $\mathbf{L}$ be an isomorphism closed class of finite structures over a vocabulary τ.
$\mathbf{L}$ is NP computable if and only if it is definable by an existential second-order sentence, i.e. $SO\exists = NP$.

Proof. $SO\exists \subseteq NP$: Let $\Phi = \exists R_1^{\alpha_1} ... \exists R_k^{\alpha_k} \phi$ be a second-order existential sentence and let τ be the vocabulary of Φ. Our task is to build an NP machine N such that for all $\mathcal{A} \in STRUC[\tau]$, $\mathcal{A} \models \Phi \iff$ *N accepts on input $bin(\mathcal{A})$.*

Let $\mathcal{A}$ be an input structure to N and let $n = \|\mathcal{A}\|$. What N does is to nondeterministically choose relations $R_1, ..., R_k$, subsets of $|\mathcal{A}|^{\alpha_1}, ..., |\mathcal{A}|^{\alpha_k}$ (this can be done by nondeterministically writing down binary string of length $n^{\alpha_1}, ..., n^{\alpha_k}$). After this polynomial number of steps, we have an expanded structure $\mathcal{A}' =< \mathcal{A}, R_1, R_2, ..., R_k >$. N should accept iff $\mathcal{A}' \models \phi$. By Proposition 5.2.1, we can test if $\mathcal{A}' \models \phi$ in logspace, so certainly in NP. Notice that N accepts A iff there is some choice of relations R_1 through R_k such that $< \mathcal{A}, R_1, R_2, ..., R_k > \models \phi$.

$NP \subseteq SO\exists$: Let M be a nondeterministic Turing machine that uses time $n^k - 1$ for inputs $bin(\mathcal{A})$ with $n = \|\mathcal{A}\|$. We write a second-order sentence, $\Phi = (\exists C_1^{2k} ... C_t^{2k} \Delta^k)\phi$ that says, "There exists an accepting computation $C_1^{2k}, ..., C_t^{2k}, \Delta$ of M". We suppose that M has one work tape and we describe how to code M's computation. Define a $(n^k - 1) \times (n^k - 1)$ matrix $\overline{C}$ of n^{2k} tape cells. For each pair $(\overline{s}, \overline{t})$, $\overline{C}(\overline{s}, \overline{t})$ codes the tape symbol σ that appears in cell $\overline{s}$ of M's work tape at time $\overline{t}$ if the cursor doesn't scan this cell. If the cursor scans the cell $\overline{s}$ at time $\overline{t}$, then $\overline{C}(\overline{s}, \overline{t})$ codes the pair $< q, \sigma >$ consisting of M's state q at time $\overline{t}$ and tape symbol σ. Let $\Gamma = \{\gamma_0, ..., \gamma_t\} = (Q \times \Sigma) \cup \Sigma$ be a listing of the possible contents of a cell of the matrix $\overline{C}$. We will let C_i, $0 \leqslant i \leqslant t$, be relation variables, one for each γ_i. The intuitive meaning of $C_i(\overline{s}, \overline{t})$ is that computation cell $\overline{s}$ at time $\overline{t}$ contains symbol γ_i. For coding $\overline{s}$ and $\overline{t}$ $(0 \leqslant \overline{s}, \overline{t} \leqslant n^k - 1)$ we use k-tuples $\overline{s} = s_1 ... s_k$, $\overline{t} = t_1 ... t_k$, where each s_i, t_i ranges over the universe of $\mathcal{A}$, i.e. is between 0 and $n - 1$. Thus, each C_i is of arity $2k$.

The k-ary relation Δ encodes M's nondeterministic moves. Intuitively, $\Delta(\overline{t})$ is true, if step $t+1$ of the computation makes choice "1" and it is false if step $t+1$ of the computation makes choice "0". Note that these choices can be determined from $C_0, ..., C_t$, but the formula Φ is simplified when we explicitly quantify Δ.

Now the sentence $\phi(C_1, ..., C_t, \Delta)$ consists of four parts: $\phi \equiv \alpha \wedge \beta \wedge \eta \wedge \zeta$. We analyse what each of these parts expresses. The first two sentences can be easily written explicitly. We explain how one can write η and ζ.

- Sentence α asserts that row 0 of the computation correctly codes input $bin(\mathcal{A})$.

- Sentence β says that it is never the case that $C_i(\overline{s}, \overline{t})$ and $C_j(\overline{s}, \overline{t})$ both hold, for $i \neq j$.

- Sentence η says that for all i, row $i+1$ of $\overline{C}$ follows from row i via move $\Delta(\overline{t})$ of M. Precisely, η asserts that the contents of tape cell $(\overline{s}, \overline{t+1})$ follows from the contents of cells $(\overline{s-1}, \overline{t+1})$, $(\overline{s}, \overline{t})$ and $(\overline{s+1}, \overline{t})$ via the move $\Delta(\overline{t})$ of M. Let $< a_{-1}, a_0, a_1, \delta > \rightarrow^M b$ mean that the triple of cell contents a_{-1},a_0, a_1 lead to cell content b via move δ of M. Then, the following sentence encodes this information for the cells of $\overline{C}$ that are not in the first and last column:
  $\eta_1 \equiv (\forall \overline{t}, \overline{t} \neq \overline{max})(\forall \overline{s}, \overline{0} < \overline{s} < \overline{max}) \bigwedge_{<a_{-1}, a_0, a_1, \delta> \rightarrow^M b} \Big(\neg^\delta \Delta(\overline{t}) \vee \neg C_{a_{-1}}(\overline{s-1}, \overline{t}) \vee \neg C_{a_0}(\overline{s}, \overline{t}) \vee \neg C_{a_1}(\overline{s+1}, \overline{t}) \vee \neg C_b(\overline{s}, \overline{t+1}) \Big)$,
  where $\neg^\delta$ is $\neg$ if δ is the nondeterministic choice "0" and is the empty symbol when δ is "1". If η_0 and η_2 are the sentences that encode the information for $\overline{s} = \overline{0}$ and $\overline{s} = \overline{max}$ respectively, then $\eta \equiv \eta_0 \wedge \eta_1 \wedge \eta_2$.

- Finally, ζ asserts that the last row of the computation includes the accept state. We may assume that when M accepts it clears its tape, moves all the way to the left, writes 1 at the first cell and enters a unique accept state q_{acc}. Then M has these tape contents until time $\overline{max} = \overline{n^k - 1}$. Let γ_f, for some $f \in \{0, ..., t\}$, correspond to the pair $< q_{acc}, 1 >$. Then $\zeta = C_f(\overline{0}, \overline{max})$.

Clearly $\mathcal{A} \models \Phi \iff$ *M accepts on input bin*($\mathcal{A}$), as Φ describes the steps of M's computation. The structure $\mathcal{A}$ has to include a relation symbol $\leqslant$ corresponding to a total order on the universe $|A|$. If such an ordering does not exist then $\Phi \equiv \exists X \exists C_1 ... C_t \Delta(X$ *is a total order*$) \wedge \phi$, where "*X is a total order*" can be expressed by a first order sentence. ■

5.3 Logical characterization of #P

Our purpose is to give the logical characterization for the class $\#P$ [44] and comment on the expressiveness and feasibility of syntax-restricted subclasses obtained in this setting of logical definability.

Motivated by Fagin's characterization of NP, Saluha et al. examine the class of problems with an associated counting function f definable using first-order formulae $\phi(\mathbf{z}, \mathbf{T})$ as follows: $f(\mathcal{A}) = |\{< \mathbf{T}, \mathbf{z} >: \mathcal{A} \models \phi(\mathbf{z}, \mathbf{T})\}|$, where $\mathcal{A}$ is an ordered finite structure, $\mathbf{T}$ is a sequence of predicate variables (relation symbols) and $\mathbf{z}$ is a sequence of first-order variables. This framework captures exactly the class $\#P$. We emphasize that the instance space of a counting problem is a set of ordered finite structures over a certain vocabulary τ, i.e. the structures that include a binary relation which is always interpreted as a total order on the elements of their universe.

Definition 5.3.1 *A counting problem is a tuple* $\mathcal{L} = (\mathcal{I}_\mathcal{L}, \mathcal{F}_\mathcal{L}, f_\mathcal{L})$, *where* $\mathcal{I}_\mathcal{L}$ *is the set of input instances,* $\mathcal{F}_\mathcal{L}(I)$ *is a set of feasible solutions for the input* $I \in \mathcal{I}_\mathcal{L}$ *and* $f_\mathcal{L}(I) = |\mathcal{F}_\mathcal{L}(I)|$ *is the counting function corresponding to the problem.*

Definition 5.3.2 *Let* $\mathcal{L}$ *be a counting problem with finite structures* $\mathcal{A}$ *over vocabulary* σ *as instances. The relation* $\leqslant$ *is interpreted as a total order on the elements of* $|A|$. *Let* $\mathbf{T} = (T_1, ..., T_r)$, $r \geqslant 0$, *be a sequence of predicate symbols and let* $\mathbf{z} = (z_1, ..., z_m)$, $m \geqslant 0$, *be a sequence of first-order variables, such that* $m + r > 0$. *We say* $\mathcal{L}$ *is in the class* $\#\mathcal{FO}$ *if there is a first-order formula* $\phi(\mathbf{z}, \mathbf{T})$ *with predicate symbols from* $\sigma \cup \mathbf{T}$ *and free first-order variables from* $\mathbf{z}$, *such that:* $f_\mathcal{L}(\mathcal{A}) = |\{< \mathbf{T}, \mathbf{z} >: \mathcal{A} \models \phi(\mathbf{z}, \mathbf{T})\}|$.

We define subclasses $\#\Pi_n$, $\#\Sigma_n$, $n \geqslant 0$ *analogously using* Π_n, Σ_n *respectively, instead of arbitrary first-order formulae. A* Π_n *(resp.* Σ_n*) formula is a formula which is of the form* $\forall x_1 \exists x_2 \forall x_3 ... Q x_n \psi$ *(resp.* $\exists x_1 \forall x_2 \exists x_3 ... Q x_n \psi$*) when it is written in the quantifier prenex normal form, where obviously* $Q = \exists$ *(resp.* $Q = \forall$*), if n is even and* $Q = \forall$ *(resp.* $Q = \exists$*), if n is odd.*

For a counting problem in $\#\mathcal{FO}$, *the associated decision problem is the following: Given an input structure* $\mathcal{A}$, *is there an interpretation of* $< \mathbf{T}, \mathbf{z} >$ *on the structure* $\mathcal{A}$ *such that* $\mathcal{A} \models \phi(\mathbf{T}, \mathbf{z})$*?*

Theorem 5.3.1 *The class* $\#P$ *coincides with the class* $\#\mathcal{FO}$.
Precisely, $\#P = \#\Pi_2$. *Hence,* $\#P = \#\Pi_2 = \#\Pi_n = \#\Sigma_n$, $n > 2$.

Proof. $\#\mathcal{FO} \subseteq \#P$: As in the proof of Fagin's Theorem, the easy direction is proved using the fact that $FO \subseteq L$.

$\#P \subseteq \#\mathcal{FO}$: Assume that $\mathcal{L}$ is a problem in the class $\#P$ and the instances of $\mathcal{L}$ are finite structures $\mathcal{A}$ over σ, which includes the relation symbol $\leqslant$. Let $\mathcal{A}$ be such a structure and M be the nondeterministic machine such that the number of accepting paths of the machine on input $bin(\mathcal{A})$ is given by $f_{\mathcal{L}}(\mathcal{A})$. Hence, to check if $f_{\mathcal{L}}(\mathcal{A})$ is nonzero is in NP. By Fagin's theorem, there is a first-order sentence $\phi(\mathbf{T})$ with relation symbols from $\sigma \cup \mathbf{T}$ such that $f_{\mathcal{L}}(\mathcal{A}) \neq 0$ if and only if $\mathcal{A} \models (\exists \mathbf{T})\phi(\mathbf{T})$.

The formula ϕ can be chosen to be a Π_2 formula involving the binary relation $\leqslant$. In the proof of Fagin's theorem, the description of the sentences α, β and η have only $\forall$ quantifiers over first-order variables. In addition, we need to define a binary relation S representing the "successor" relation, and two unary relations for the max and min. When we use these relations in ϕ, an alternation of quantifiers occurs. Finally, the formula Φ of Fagin's Theorem can be written in quantifier prenex normal form as follows $\Phi \equiv \exists \mathbf{T}\phi(\mathbf{T}) \equiv \exists \mathbf{T}\forall \mathbf{x}\exists \mathbf{y}\psi(\mathbf{T})$.

Further, the formula ϕ is such that, every accepting computation of the NP machine M on input $bin(\mathcal{A})$ corresponds to a unique value of the sequence $\mathbf{T}$ which satisfies $\phi(\mathbf{T})$. The number of accepting computations of the NP machine is equal to $|\{<\mathbf{T}>: \mathcal{A} \models \phi(\mathbf{T})\}|$.

Therefore $\#P = \#\Pi_2 = \#\mathcal{FO}$. ■

Comment 5.3.1 *(i) Although the relation $\leqslant$ is not needed in the proof of Fagin's theorem, in the above proof the structure $\mathcal{A}$ includes such a relation. In the absence of a total order, one can define a binary relation $\leqslant$ and assert, as a subformula of ϕ, that $\leqslant$ represents a total order on the universe. However, any total order will suffice in satisfying the formula. There are $\|\mathcal{A}\|!$ possible binary relations which are total orders and hence the number of distinct assignments to $\mathbf{T}$ which satisfy $\phi(\mathbf{T})$ is $\|\mathcal{A}\|!$ times the number of accepting paths in the corresponding NP machine. This problem disappears when the structures are ordered, i.e. when they include a relation which represents a total order.*

(ii) Instead of counting only satisfying predicate variables $\mathbf{T}$, we count assignments to both second-order and first-order variables $<\mathbf{T}, \mathbf{z}>$. The reason for doing this is that there are functions in $\#P$, which are more naturally expressible by counting assignments of just first-order variables or a combination of first-order and second-order variables.

In [9] it was shown that $\#\Pi_2[\preceq] = \#\Pi_1[SUCC] = \#\Pi_1[+]$. At first we make clear what we mean by writing $\#\Pi_k[op]$ and then give a sketch proof for these equations.

Definition 5.3.3 *Let op be one of the following predicates: $\preceq, SUCC, +$. Let σ and τ be vocabularies that do not contain op and $\leqslant$. Let $(\mathcal{A}, \leqslant^{\mathcal{A}})$ be an ordered σ-structure and let $op^{\leqslant^{\mathcal{A}}}$ be the total ordering ($\preceq$) or the successor relation ($SUCC$) or the plus relation ($+$) that induces $\leqslant^{\mathcal{A}}$.*

For $k \in \mathbb{N}$ we define the class $\#\Sigma_k[op]$ ($\#\Pi_k[op]$) to be the class of functions f such that there exists a Σ_k-formula (Π_k-formula respectively) ϕ over (σ, τ, op) and $f(\mathcal{A}, \leqslant^{\mathcal{A}}) = |\{\mathbf{T} : (\mathcal{A}, \mathbf{T}, op^{\mathcal{A}}) \models \phi\}|$ for all ordered structures $(\mathcal{A}, \leqslant^{\mathcal{A}})$.

Note that given a linear ordering $\leqslant^{\mathcal{A}}$ the relations $\preceq^{\mathcal{A}}$ ($\equiv\leqslant^{\mathcal{A}}$), $SUCC^{\mathcal{A}}$ and $+^{\mathcal{A}}$ compatible to $\leqslant^{\mathcal{A}}$ are uniquely determined. We also assume that the universe of the structure is of the form $\{0, ..., n-1\}$.

The definition of $\#\Sigma_k[op]$ ($\#\Pi_k[op]$) is a generalization of the definition given in [44]: $\#\Sigma_k = \#\Sigma_k[\preceq]$ ($\#\Pi_k = \#\Pi_k[\preceq]$). Moreover, we prefer to count only relation symbols at this point, since these proofs are theoretical and they do not consider any examples.

Theorem 5.3.2 *(i)* $\#\Pi_1[SUCC] = \#\Pi_1[+]$.
(ii) $\#\Pi_2[\preceq] = \#\Pi_1[SUCC]$.

Sketch proof (i) $\#\Pi_1[SUCC] \subseteq \#\Pi_1[+]$: Let $f \in \#\Pi_1[SUCC]$. That is, there exist σ, τ such that for all ordered σ-structures $(\mathcal{A}, \leqslant^{\mathcal{A}})$: $f(\mathcal{A}, \leqslant^{\mathcal{A}}) = |\{\mathbf{T} : (\mathcal{A}, \mathbf{T}, SUCC^{\mathcal{A}}) \models \phi\}|$ for some Π_1 formula ϕ. A Π_1 formula ϕ_+ can be defined such that

$$(\mathcal{A}, 0, n-1, \preceq^{\mathcal{A}}, +^{\mathcal{A}}, \mathbf{T}, SUCC^{\mathcal{A}}) \models (\phi \wedge \phi_+) \text{ iff } (\mathcal{A}, \mathbf{T}, SUCC^{\mathcal{A}}) \models \phi.$$

In other words a plus relation can be defined using the successor relation.
Let $\tau' = (min, max, \preceq, SUCC, \tau)$, then there exists a τ'-expansion $\mathbf{T}'$ for all ordered σ-structures $\mathcal{A}$ such that $f(\mathcal{A}, \leqslant^{\mathcal{A}}) = |\mathbf{T}' : (\mathcal{A}, \mathbf{T}', +^{\mathcal{A}}) \models (\phi \wedge \phi_+)\}|$. Thus, $f \in \#\Pi_1[+]$.
$\#\Pi_1[+] \subseteq \#\Pi_1[SUCC]$: Similar as above, we use a Π_1 formula ϕ_{SUCC} to define a successor relation using the plus relation.

(ii) $\#\Pi_1[SUCC] \subseteq \#\Pi_2[\preceq]$: This direction is easy since we can use the formula ϕ_{SUCC} to define a successor relation using a given linear order. Thus, all counting functions that are definable via Π_1 formulae that use a successor relation are definable also by Π_2 formulae using a linear ordering.
$\#\Pi_2[\preceq] \subseteq \#\Pi_1[SUCC]$: This direction is much more difficult to prove. For each Π_2 formula ϕ over $(\sigma, \tau, \preceq)$ we should construct a Π_1 formula $\hat{\phi}$ over $(\sigma, \hat{\tau}, SUCC)$, such that for all ordered σ-structures $(\mathcal{A}, \leqslant^{\mathcal{A}})$ it holds that

$$|\{\mathbf{T}^{\mathcal{A}} : (\mathcal{A}, \mathbf{T}^{\mathcal{A}}, \leqslant^{\mathcal{A}} \models \phi\}| = |\{\hat{\mathbf{T}}^{\mathcal{A}} : (\mathcal{A}, \hat{\mathbf{T}}^{\mathcal{A}}, SUCC^{\mathcal{A}}) \models \hat{\phi}\}|.$$

The proof can be found in [9]. ■

Corollary 5.3.1 $\#P = \#\Pi_2[\preceq] = \#\Pi_1[+] = \#\Pi_1[SUCC]$.

5.4 Logical Hierarchy in #P

In this section we restrict ourselves to the classes $\#\Sigma_k[\preceq]$, $\#\Pi_k[\preceq]$ or just $\#\Sigma_k$, and $\#\Pi_k$ as defined in [44].

Descriptive complexity gives some insight into subclasses of $\#P$. A fundamental question of counting complexity concerns the computational feasibility of counting problems. By computationally feasible we mean either polynomial time computable or approximable by a polynomial time randomised algorithm. Can we distinguish the “easy” from the “hard” counting problems? Which subclasses of $\#P$ can help us to classify counting problems?

In [44] and [16] the computational feasibility of problems belonging to $\#P$ is related to logically definable classes.

We begin the presentation of these attempts from the study of the classes $\#\Sigma_0 = \#\Pi_0$, $\#\Sigma_1$, $\#\Pi_1$, $\#\Sigma_2$ and $\#\Pi_2$ which are obtained by restricting the quantifier complexity of the first-order formula.

Theorem 5.4.1 *The problems in $\#\Pi_2$ form a linear hierarchy with five distinct levels:*
$\#\Sigma_0 = \#\Pi_0 \subsetneq \#\Sigma_1 \subsetneq \#\Pi_1 \subsetneq \#\Sigma_2 \subsetneq \#\Pi_2 = \#P$.

Proof. All the containments of the theorem are trivial except for the containment $\#\Sigma_1 \subseteq \#\Pi_1$. Let $\mathcal{L}$ be a counting problem in $\#\Sigma_1$ with $f_{\mathcal{L}}(\mathcal{A}) = |\{<\mathbf{T},\mathbf{z}>: \mathcal{A} \models (\exists\mathbf{x})\psi(\mathbf{x},\mathbf{z},\mathbf{T})\}|$, where $\psi(\mathbf{x},\mathbf{z},\mathbf{T})$ is a quantifier-free formula. Instead of counting the tuples $<\mathbf{T},\mathbf{z}>$, we can count the tuples $<\mathbf{T},(\mathbf{z},\mathbf{x}^*)>$, where $\mathbf{x}^*$ is the lexicographically smallest $\mathbf{x}$, such that $\mathcal{A} \models \psi(\mathbf{x},\mathbf{z},\mathbf{T})$. Let $\theta(\mathbf{x}^*,\mathbf{x},\leqslant)$ be the quantifier-free formula that expresses the fact that $\mathbf{x}^*$ is lexicographically smaller than $\mathbf{x}$ under $\leqslant$. Then,
$f_{\mathcal{L}}(\mathcal{A}) = |\{<\mathbf{T},(\mathbf{z},\mathbf{x}^*)>: \mathcal{A} \models \psi(\mathbf{x}^*,\mathbf{z},\mathbf{T}) \wedge (\forall\mathbf{x})\psi(\mathbf{x},\mathbf{z},\mathbf{T}) \rightarrow \theta(\mathbf{x}^*,\mathbf{x},\leqslant)\}|$.
Hence, $\mathcal{L} \in \#\Pi_1$ and $\#\Sigma_1 \subseteq \#\Pi_1$.

The strictness of the containments is due to the following facts:

- $\#3DNF$ is in the class $\#\Sigma_1$ but not in the class $\#\Sigma_0$.
- $\#3SAT$ is in the class $\#\Pi_1$ but not in the class $\#\Sigma_1$.
- $\#DNF$ is in the class $\#\Sigma_2$ but not in the class $\#\Pi_1$.
- $\#HAMILTONIAN$ is in the class $\#\Pi_2$ but not in the class $\#\Sigma_2$.

The proofs of the above propositions are model theoretic and are based on smart remarks. They can be found in [44]. ■

The next two theorems will bring us one step closer to our objective. They assert that every problem in $\#\Sigma_0$ is polynomial time computable and that every problem in $\#\Sigma_1$ is approximable in polynomial time by a randomised algorithm.

Theorem 5.4.2 *Every counting problem in $\#\Sigma_0$ is computable in deterministic polynomial time.*

Proof. Let $\mathcal{L}$ be a counting problem in $\#\Sigma_0$. Then $f_{\mathcal{L}}(\mathcal{A}) = |\{<\mathbf{T},\mathbf{z}>: \mathcal{A} \models \psi_{\mathcal{L}}(\mathbf{z},\mathbf{T})\}|$, where $\psi_{\mathcal{L}}$ is a quantifier-free formula, $\mathcal{A}$ is a finite ordered structure over a vocabulary σ, $\mathbf{T}$ is a sequence $(T_1,...,T_r)$ of predicate variables of arities $a_1,...,a_r$, respectively, and $\mathbf{z}$ is a sequence $(z_1,...,z_m)$ of first-order variables.

To compute $f_{\mathcal{L}}(\mathcal{A})$, we count for each $\mathbf{z}^* \in |\mathcal{A}|^m$, the number of assignments to $\mathbf{T}$ so that $\mathcal{A} \models \psi_{\mathcal{L}}(\mathbf{z}^*,\mathbf{T})$. The number of such $\mathbf{z}^*$ is $\|\mathcal{A}\|^m$. So, it suffices to show that for every $\mathbf{z}^* \in |\mathcal{A}|^m$ we can compute $f_{\mathcal{L}}^{\mathbf{z}^*}(\mathcal{A}) = |\{\mathbf{T} : \mathcal{A} \models \psi_{\mathcal{L}}(\mathbf{z}^*,\mathbf{T})\}|$ in polynomial time. Then, $f_{\mathcal{L}}(\mathcal{A}) = \sum_{\mathbf{z}^* \in |\mathcal{A}|^m} f_{\mathcal{L}}^{\mathbf{z}^*}(\mathcal{A})$.

For every $\mathbf{z}^* \in |\mathcal{A}|^m$ consider the formula $\psi_{\mathcal{L}}(\mathbf{z}^*,\mathbf{T})$. This formula can be viewed as a propositional formula with variables of the form $T_i(\mathbf{y_i})$ where $\mathbf{y_i}$ are tuples of arity a_i,

$i = 1, ..., r$. The total number of such variables is $\sum_{i=1}^{r} \|\mathcal{A}\|^{a_i}$. Let $c(\mathbf{z}^*)$ denote the number of a_i-tuples $\mathbf{y_i}$, for $i = 1, ..., r$, such that $T_i(\mathbf{y_i})$ or $\neg T_i(\mathbf{y_i})$ appear in $\psi_{\mathcal{L}}(\mathbf{z}^*, \mathbf{T})$. Obviously the number of variables of the form $T_i(\mathbf{y_i})$, $i = 1, ..., r$, that do not appear in the formula is $(\sum_{i=1}^{r} \|\mathcal{A}\|^{a_i}) - c(\mathbf{z}^*)$. Also, $c(\mathbf{z}^*)$ does not depend on the size of the structure $\mathcal{A}$. Hence, we can find all the truth assignments to the variables that occur in this propositional formula in constant time. Let $s(\mathbf{z}^*)$ be the number of such satisfying assignments. It is easily seen that $f_{\mathcal{L}}^{\mathbf{z}^*}(\mathcal{A}) = s(\mathbf{z}^*) \cdot 2^{(\sum_{i=1}^{r} \|\mathcal{A}\|^{a_i}) - c(\mathbf{z}^*)}$, which is computed in polynomial time. ■

Comment 5.4.1 *It is unlikely that every problem in the next higher class in the hierarchy, i.e. $\#\Sigma_1$, is computable in polynomial time, because $\#\Sigma_1$ has #P-complete problems, e.g. $\#3DNF$.*

Theorem 5.4.3 *Every counting function in $\#\Sigma_1$ has an FPRAS.*

The existence of a FPRAS for every problem in $\#\Sigma_1$ is proved in two steps:

1. Every problem in $\#\Sigma_1$ is reducible to a restricted version of $\#DNF$, the problem $\#k \cdot logDNF$.
2. The problem $\#k \cdot logDNF$ has an FPRAS.

For the first step, we need the appropriate notion of reduction and the definition of the problem $\#k \cdot logDNF$.

Definition 5.4.1 *Given counting problems $\mathcal{L}$, $\mathcal{R}$, we say $\mathcal{L}$ is (polynomial time) product reducible to $\mathcal{R}$, $\mathcal{L} \leqslant_{pr} \mathcal{R}$, if there are polynomial time computable functions $g : \mathcal{I}_{\mathcal{L}} \to \mathcal{I}_{\mathcal{R}}$, $h : \mathbb{N} \to \mathbb{N}$, such that for every finite structure $\mathcal{A}$ which is an input to $\mathcal{L}$ the value of the counting function is given by $f_{\mathcal{L}}(\mathcal{A}) = f_{\mathcal{R}}(g(\mathcal{A})) \times h(|\mathcal{A}|)$. If h is the constant 1 function, the reduction is said to be parsimonious.*

Proposition 5.4.1 *(i) Given counting problems $\mathcal{L}$, $\mathcal{R}$, if $\mathcal{L} \leqslant_{pr} \mathcal{R}$ and $\mathcal{R}$ is computable in polynomial time, then $\mathcal{L}$ is computable in polynomial time.*

(ii) Given counting problems $\mathcal{L}$, $\mathcal{R}$, if $\mathcal{L} \leqslant_{pr} \mathcal{R}$ and $\mathcal{R}$ has a polynomial time randomised (ϵ, δ) approximation algorithm, then $\mathcal{L}$ also has a polynomial time randomised (ϵ, δ) approximation algorithm.

Proof. Follows easily from the definition of the product reduction. ■

Definition 5.4.2 *For any positive integer k, the counting problem $\#k \cdot logDNF$ ($\#k \cdot logCNF$) is the problem of counting the number of satisfying assignments to a propositional formula in disjunctive normal form (respectively, conjunctive normal form) in which the number of literals in each disjunct (respectively, conjunct) is at most $klogn$, where n is the number of propositional variables in the formula.*

Lemma 5.4.1 *For every counting problem $\mathcal{L} \in \#\Sigma_1$, there is a positive constant k so that $\mathcal{L} \leqslant_{pr} \#k \cdot logDNF$.*

Proof. Let $\mathcal{L} \in \#\Sigma_1$ and $f_{\mathcal{L}}(\mathcal{A}) = |\{< \mathbf{T}, \mathbf{z} >: \mathcal{A} \models (\exists \mathbf{y})\psi(\mathbf{y}, \mathbf{z}, \mathbf{T})\}|$, where $\psi(\mathbf{y}, \mathbf{z}, \mathbf{T})$ is a quantifier-free formula in DNF form, with at most t literals per disjunct, $\mathbf{T}$ is a sequence $(T_1, ..., T_r)$ of second-order predicate variables whose arities are $a_1, ..., a_r$ respectively, $\mathbf{y}$ and $\mathbf{z}$ are sequences $(y_1, ..., y_p)$ and $(z_1, ..., z_m)$ respectively, of first-order variables. Let $\{y_1, ..., y_{\|\mathcal{A}\|^p}\}$ be the set $|\mathcal{A}|^p$ and $\{z_0, ..., z_{\|\mathcal{A}\|^m-1}\}$ be the set $|\mathcal{A}|^m$.

For every $z_i \in |\mathcal{A}|^m$, we write the formula $(\exists \mathbf{y})\psi(\mathbf{y}, \mathbf{z}, \mathbf{T})$ as a disjunction $\bigvee_{j=1}^{\|\mathcal{A}\|^p} \psi(\mathbf{y_j}, \mathbf{z_i}, \mathbf{T})$. By replacing in this new formula every subformula that is satisfied by $\mathcal{A}$ by the value TRUE, and every subformula that is not satisfied by $\mathcal{A}$ by the value FALSE, we obtain a propositional formula $\psi'(\mathbf{z_i}, \mathbf{T})$ in DNF form with variables $T_q(\mathbf{w_q})$, $\mathbf{w_q} \in |\mathcal{A}|^{a_q}$ and $q = 1, ..., r$. This means that each literal of this formula is of the form $T_q(\mathbf{w_q})$ or $\neg T_q(\mathbf{w_q})$ for some $q \in \{1, ..., r\}$ (or it has already the value TRUE or FALSE). Let $c(\mathcal{A})$ be the number of variables $T_q(\mathbf{w_q})$, $\mathbf{w_q} \in |\mathcal{A}|^{a_q}$ and $q = 1, ..., r$, that do not appear in any formula $\psi'(\mathbf{z_i}, \mathbf{T})$, $i = 1, ..., m$.

At the end, the *DNF* formula we construct has to accept as satisfying assignments all the pairs $< \mathbf{T}, \mathbf{z} >$ that satisfy ψ in $\mathcal{A}$. Thus, we need $|\mathcal{A}|^m$ new conjuncts, one for each z_i: We define l new propositional variables $x_1, ..., x_l$, where $2^{l-1} < |\mathcal{A}|^m \leqslant 2^l$. For every $\mathbf{s} \in \{0,1\}^l$, let $x(\mathbf{s})$ represent the conjunction of these l variables so that for $1 \leqslant i \leqslant l$, x_i appears complemented if and only if the i^{th} component of $\mathbf{s}$ is 0. For example, if $l = 4$ and $\mathbf{s} = 0101$, then $x(\mathbf{s}) = \neg x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4$. We interpret $\mathbf{s}$ as the binary representation of an integer between 0 and $2^l - 1$.

Consider now the propositional formula
$\theta_{\mathcal{A}} \equiv^{def} [\psi'(\mathbf{z_0}, \mathbf{T}) \wedge x(0)] \vee [\psi'(\mathbf{z_1}, \mathbf{T}) \wedge x(1)] \vee ... \vee [\psi'(\mathbf{z_{\|\mathcal{A}\|^m-1}}, \mathbf{T}) \wedge x(\|\mathcal{A}\|^m - 1)]$.

(1) This formula can be written in DNF form and its variables are among $x_1, ..., x_l$ and $T_q(\mathbf{w_q})$, $q = 1, ...r$.

(2) Each disjunct contains at most $t + l$ literals, for some constant t. All the variables occurring in $\theta_{\mathcal{A}}$ are at most $n = \sum_{i=1}^{r} \|\mathcal{A}\|^{a_i} + l$, and $2^{l-1} < |\mathcal{A}|^m \leqslant 2^l$, $l = \mathcal{O}(logn)$. Hence $\theta_{\mathcal{A}}$ is a $k \cdot logDNF$ formula for suitable k depending on the size of $\psi(\mathbf{y}, \mathbf{z}, \mathbf{T})$.

(3) The truth value of $c(\mathcal{A})$ variables do not affect the truth of $\theta_{\mathcal{A}}$. So all the possible truth assignments on these variables can generate pairs $< \mathbf{T}, \mathbf{z} >$ that satisfy the initial formula ψ in $\mathcal{A}$. As a result, $f_{\mathcal{L}}(\mathcal{A}) = 2^{c(\mathcal{A})} \times$ (*the number of satisfying assignments of* $\theta_{\mathcal{A}}$).

(4) We can construct $\theta_{\mathcal{A}}$ in polynomial time. ■

Comment 5.4.2 *The decision version of a problem $\mathcal{L} \in \#\Sigma_1$ is : Given a finite structure $\mathcal{A}$, is $f_{\mathcal{L}}(\mathcal{A}) \neq 0$? Equivalently, $\mathcal{A} \models (\exists \mathbf{T})(\exists \mathbf{z})\phi(\mathbf{z}, \mathbf{T})$?, where ϕ is an existential formula. There exists a first-order existential formula ϕ' such that $\mathcal{A} \models (\exists \mathbf{T})(\exists \mathbf{z})\phi(\mathbf{z}, \mathbf{T})$ if and only if $\mathcal{A} \models \phi'$. It is known that the problem of model checking for a first-order formula is in the uniform version of the class AC^0. Hence, the complexity of the decision version of every $\#\Sigma_1$ problem is very low.*

Lemma 5.4.2 *For every k, there is a FPRAS for the $\#k \cdot logDNF$ problem.*

The $\#k \cdot logDNF$ problem is a special case of the $\#DNF$ problem which has a FPRAS as we showed in section 4.2. In fact, it suffices to show that every problem in $\#\Sigma_1$ is product reducible to $\#DNF$ in Lemma 5.4.1.

Theorem 5.4.3 follows immediately from Proposition 5.4.1(ii), Lemma 5.4.1 and Lemma 5.4.2.

Comment 5.4.3 *(i) The class $\#\Sigma_1$ has a lot of interesting problems like, $\#MONOCHROMATIC$-$kCLIQUE$-$PARTITIONS$, $\#NON$-$VERTEX$-$COVERS$, $\#NON$-$CLIQUES$.*

(ii) It is unlikely that every problem in the next higher class in the hierarchy, i.e. $\#\Pi_1$ has an FPRAS. Such a result would imply that $NP = RP$, since $\#3SAT$ is in the class $\#\Pi_1$ and $3SAT$ is NP-complete.

A natural question to ask in this context is: Given an arbitrary first order formula $\phi(\mathbf{z}, \mathbf{T})$, is the counting problem defined using this formula polynomial time computable or is approximable by a polynomial time (ϵ, δ) randomised algorithm?

Let σ be a vocabulary and let $\phi(\mathbf{z}, \mathbf{T})$ be a first-order formula with predicate symbols from $\sigma \cup \mathcal{T}$. We say a counting problem $\mathcal{L}_\phi$ with instances finite structures over σ is defined by the formula $\phi(\mathbf{z}, \mathbf{T})$, if the counting function $f_\mathcal{L}$ is defined using ϕ: $f_\mathcal{L}(\mathcal{A}) = |\{< \mathbf{T}, \mathbf{z} >: \mathcal{A} \models \phi(\mathbf{z}, \mathbf{T})\}|$. We also write f_ϕ for this function.

Theorem 5.4.4 *Let σ be a vocabulary with a unary predicate $\{C\}$ and three binary predicate symbols $\{E, P, N\}$.*

(i) Assuming $NP \neq RP$, the following is an undecidable problem: Given a first-order formula $\phi(\mathbf{z}, \mathbf{T})$ over $\sigma \cup \mathcal{T}$, does the counting problem $\mathcal{L}_\phi$ have a polynomial time (ϵ, δ) randomised approximation algorithm, for some constants $\epsilon, \delta > 0$?

(ii) Similarly, assuming $P \neq P^{\#P}$, the following is an undecidable problem: Given a first-order formula $\phi(\mathbf{z}, \mathbf{T})$ over $\sigma \cup \mathcal{T}$, is the counting problem $\mathcal{L}_\phi$ polynomial time computable?

The above results do not contradict the fact that there is an effective syntax for the class of counting problems that are computable in polynomial time or for the class of all approximable counting functions.

We would like to see whether the classes $\#\Pi_1$ and $\#\Sigma_2$ also capture some aspect of the computational complexity of the counting problems in them. One intuitive reason for the difficulty in answering this question is that $\#\Pi_1$ has counting functions which are complete for $\#P$ under parsimonious reductions, e.g., $\#3SAT$.

An approach to study the computational properties of subclasses of $\#P$ is to restrict the quantifier-free part of the first-order formula. At the first place we isolate a syntactic subclass of $\#\Sigma_2$.

Definition 5.4.3 *Let σ be a vocabulary and let $\mathcal{L}$ be a counting problem, with finite ordered structures $\mathcal{A}$ over σ as instances. We say that $\mathcal{L}$ is in the class $\#R\Sigma_2$ if there is a quantifier-free first-order formula $\psi_{\mathcal{L}}(\mathbf{z},\mathbf{T})$ over $\sigma \cup \mathbf{T}$, such that $f_{\mathcal{L}}(\mathcal{A}) = |\{<\mathbf{T},\mathbf{z}>: \mathcal{A} \models \exists x \forall y \psi_{\mathcal{L}}(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{T})\}|$, where $\psi_{\mathcal{L}}(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{T})$ is a quantifier-free formula, and when $\psi_{\mathcal{L}}$ is expressed in an equivalent formula in conjunctive normal form, each conjunct has at most one occurrence of a predicate symbol from $\mathbf{T}$.*

Lemma 5.4.3 *$\#DNF$ is complete for $\#R\Sigma_2$ under product reductions.*

Proof. To show that $\#DNF$ is in $\#R\Sigma_2$, we use the vocabulary $\{D,P,N\}$, with a unary relation D and two binary P,N, to encode a DNF formula ϕ as a finite structure $\mathcal{A}_\phi$. The structure $\mathcal{A}_\phi$ has universe $|\mathcal{A}_\phi| = D \cup V$, where V is the set of variables and D is the set of disjuncts of ϕ. The predicate $D(d)$ expresses the fact that d is a disjunct, whereas $P(d,u)$ ($N(d,u)$) expresses that the disjunct d contains variable u positively (negatively). Let T denote the set of variables assigned true value in a satisfying assignment to the instance ϕ. Under this encoding, $f_{\#DNF}(\mathcal{A}_\phi) = |\{<T>: \mathcal{A}_\phi \models (\exists d)(\forall u) D(d) \wedge (P(d,u) \rightarrow T(u)) \wedge (N(d,u) \rightarrow \neg T(u))\}|$. Hence, $\#DNF \in \#R\Sigma_2$, since the quantifier-free part $D(d) \wedge (P(d,u) \rightarrow T(u)) \wedge (N(d,u) \rightarrow \neg T(u))$ is in conjunctive normal form and T appears at most once in each conjunct.

To show that $\#DNF$ is hard for the class $\#R\Sigma_2$, consider a counting function $\mathcal{L}$ in $\#R\Sigma_2$ which is expressible as $f_{\mathcal{L}}(\mathcal{A}) = |\{<\mathbf{T},\mathbf{z}>: \mathcal{A} \models (\exists \mathbf{x})(\forall \mathbf{y}) \psi_{\mathcal{L}}(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{T})\}|$, where $\psi_{\mathcal{L}}(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{T})$ is a quantifier-free formula in CNF and each conjunct has at most one occurrence of a predicate symbol $\mathbf{T}$.

Let $\mathbf{x} = (x_1,...,x_{p_1})$ and $\mathbf{y} = (y_1,...,y_{q_1})$ for some $p_1, q_1 > 0$. Let $\|\mathcal{A}\|^{p_1} = p$ and $\|\mathcal{A}\|^{q_1} = q$, $|\mathcal{A}|^p = \{\mathbf{x}_1,...,\mathbf{x}_p\}$ and $|\mathcal{A}|^q = \{\mathbf{y}_1,...,\mathbf{y}_q\}$. Then, $\mathcal{A} \models (\exists \mathbf{x})(\forall \mathbf{y}) \psi_{\mathcal{L}}(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{T})$ if and only if $\mathcal{A} \models \bigvee_{i=1}^{p} \bigwedge_{j=1}^{q} \psi_{\mathcal{L}}(\mathbf{x}_i,\mathbf{y}_j,\mathbf{z},\mathbf{T})$ if and only if $\mathcal{A} \models \bigvee_{i=1}^{p} \bigwedge_{j=1}^{q} \psi_{i,j}(\mathbf{z},\mathbf{T})$, where $\psi_{i,j}(\mathbf{z},\mathbf{T})$ is obtained from $\psi_{\mathcal{L}}(\mathbf{x}_i,\mathbf{y}_j,\mathbf{z},\mathbf{T})$ by replacing every subformula that is true in $\mathcal{A}$ by the logical value TRUE, every subformula that is false in $\mathcal{A}$ by FALSE, and then by deleting the values FALSE and all the conjuncts that contain the value TRUE. One can check that the resulting formula, say $\theta_{\mathcal{A}}$, is in DNF form with propositional variables of the form $T_i(\mathbf{w}_i)$, $\mathbf{w}_i \in |\mathcal{A}|^{a_i}$. This uses the fact that $\psi_{\mathcal{L}}(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{T})$ has at most one of any predicate from the sequence $\mathbf{T}$ per conjunct.

Let $c_{\mathcal{A}}$ be the number of propositional variables which do not appear in $\theta_{\mathcal{A}}$. It can be easily verified that $f_{\mathcal{L}}(\mathcal{A}) = c_{\mathcal{A}} \times (\textit{the number of satisfying assignments of } \theta_{\mathcal{A}})$. Finally, the above reduction can be computed in polynomial time. ■

Theorem 5.4.5 *Every counting problem in $\#R\Sigma_2$ has an FPRAS. Moreover, the decision version of every problem in $\#R\Sigma_2$ is in P.*

Example 5.4.1 *Examples of problems in $\#R\Sigma_2$ are $\#NON{-}HITTING{-}SETS$, $\#NON{-}DOMINATING - SETS$ and $\#NON - EDGE - DOMINATING - SETS$.*

Another approach to obtain classes of counting problems with feasible computational properties is to consider the closure of the classes $\#\Sigma_0$ and $\#\Sigma_1$ under product reductions. Then $\#P$ form an hierarchy with three levels.

Definition 5.4.4 *Given a class $\mathcal{C}$ of counting problems, the closure class $P_{pr}(\mathcal{C})$ is the class of counting problems which are product reducible to some problem in $\mathcal{C}$.*

In particular, we are interested in the closure classes $P_{pr}(\#\Sigma_0)$, $P_{pr}(\#\Sigma_1)$ and $P_{pr}(\#\Pi_1)$.

Theorem 5.4.6 *(i) Every counting problem in the class $P_{pr}(\#\Sigma_0)$ is computable in polynomial time. In fact, $P_{pr}(\#\Sigma_0)$ is exactly the class of polynomially computable counting functions.*

(ii) Every counting problem in the class $P_{pr}(\#\Sigma_1)$ has an FPRAS. The decision version of every problem in $P_{pr}(\#\Sigma_1)$ is in P.

Theorem 5.4.7 *(i) The classes $P_{pr}(\#\Sigma_0)$, $P_{pr}(\#\Sigma_1)$ and $P_{pr}(\#\Pi_1)$ form a hierarchy with three levels: $P_{pr}(\#\Sigma_0) \subseteq P_{pr}(\#\Sigma_1) \subseteq P_{pr}(\#\Pi_1) = \#P$.*

(ii) $P_{pr}(\#\Sigma_0) = P_{pr}(\#\Sigma_1)$ if and only if $P = P^{\#P}$.

(iii) If $P_{pr}(\#\Sigma_1) = P_{pr}(\#\Pi_1)$, then $NP = RP$.

5.5 Descriptive Complexity of #BIS and its relatives

In recent years there has been great interest in classifying the approximation complexity of counting problems.

The steady progress in determining the complexity of counting graph homomorphisms contributed to the study of approximation counting complexity. Counting graph homomorphisms is equivalent to counting colourings (with specific properties) of graphs. These problems have applications in statistical physics.

In such problems, known as H-colouring problems, the graph H is fixed and we consider the problem of counting the number of homomorphisms from an input graph G to the fixed graph H (H-colourings of G). In 1990 Hell and Nešetřil [23] showed that there is a complexity dichotomy between graphs H for which the corresponding decision problem is in P, and those for which it is NP-complete. In 2000 Dyer and Greenhill [17] considered the problem of counting H-colourings. They were able to completely characterize the graphs H for which this problem is $\#P$-complete. They defined a trivial connected component of H to be one that is a complete graph with all loops present or a complete bipartite graph with no loops present, and showed that counting H-colourings is $\#P$-complete if H has a nontrivial component and that it is in P otherwise.

According to a very recent result [21], if H is any fixed graph without trivial components, then the problem is as hard as $\#BIS$ with respect to approximation-preserving reducibility. This means that if H is any fixed graph without trivial components, the problem of approximately counting H-colourings is as hard as approximately counting independent sets in bipartite graphs.

In fact $\#BIS$ is a well known problem. In 2003 Dyer et al. [16] turned their attention to the relative complexity of approximate counting problems, introducing the idea of approximation preserving reductions. They showed that there appear to be three distinct classes of problems relative to such reductions: those that can be approximated in polynomial time, those that are as hard to approximate as $\#SAT$, and a logically defined intermediate class of problems that are equivalent in approximation complexity to approximately counting independent sets in a bipartite graph, i.e. $\#BIS$.

We present useful definitions for these results.

Definition 5.5.1 *Suppose $f, g : \Sigma^* \to \mathbb{N}$ are functions whose complexity (of approximation) we want to compare. An approximation-preserving reduction from f to g is a probabilistic oracle Turing machine M that takes as input a pair $(x, \epsilon) \in \Sigma^* \times (0, 1)$, and satisfies the following three conditions: (i) every oracle call made by M is of the form (w, δ), where $w \in \Sigma^*$ is an instance of g, and $0 < \delta < 1$ is an error bound satisfying $\delta^{-1} \leqslant poly(|x|, \epsilon^{-1})$, (ii) the TM M meets the specification for being a RAS for f whenever the oracle meets the specification for being a RAS for g, and (iii) the run-time of M is polynomial in $|x|$ and ϵ^{-1}.*

If an approximation-preserving reduction from f to g exists we write $f \leqslant_{AP} g$ and say that f is AP-reducible to g. If $f \leqslant_{AP} g$ and $g \leqslant_{AP} f$, then we say that f and g are AP-interreducible and write $f \equiv_{AP} g$.

$\#BIS$:
Input: A bipartite graph B.
Output: The number of independent sets in B.

A very few non-trivial combinatorial structures may be counted exactly using a polynomial time deterministic algorithm. The two key examples are spanning trees in a graph and perfect matchings in a planar graph. Both of these algorithms rely on a reduction to a determinant, which can be computed in polynomial time by Gaussian elimination.

There are some counting problems that admit an FPRAS despite being complete in $\#P$ with respect to Turing reducibility. Two representative examples are $\#DNF$ and $\#MATCH$.

$\#MATCH$:
Input: A graph G.
Output: The number of matchings (of all sizes) in G.

On the other hand, the problem $\#SAT$ is $\#P$-complete under parsimonious reductions. Since a parsimonious reduction is a very special case of an approximation-preserving reduction, all problems in $\#P$ are AP-reducible to $\#SAT$. Thus, $\#SAT$ is complete for $\#P$ with respect to AP-reducibility. Zuckerman [57] has shown that $\#SAT$ cannot have an FPRAS unless $NP = RP$. The same is obviously true of any problem in $\#P$ to which $\#SAT$ is AP-reducible. Dyer et al. proved that the counting versions of NP-complete problems are complete for $\#P$ with respect to AP-reducibility. In other words, $\#SAT$ is AP-reducible to these problems. For example, $\#SAT \leqslant_{AP} \#LARGEIS$ and hence $\#SAT \equiv_{AP} \#LARGEIS$.

$\#LARGEIS$:
Input: A positive integer m and a graph G in which every independent set has size at most m.
Output: The number of size-m independent sets in G.

It is known that the decision version of determining whether or not G has a size-m independent set is NP-complete. Additionally, a problem may be complete for $\#P$ with respect to AP-reducibility even though its associated decision problem is polynomial time solvable, as it is the case with $\#IS \equiv_{AP} \#SAT$.

We do not have any evidence that the restriction of $\#IS$ to bipartite graphs is complete for $\#P$ with respect to AP-reducibility. The fact that $\#BIS$ is interreducible with a number of other problems not known to be complete or to admit an FPRAS makes this problem and its relatives interesting. The following list provides examples of problems AP-interreducible with $\#BIS$:

$\#P_4\text{-}COL$:
Input: A graph G.
Output: The number of P_4-colourings of G, where P_4 is the path of length 3.

$\#DOWNSETS$:
Input: A partially ordered set $(X, \leqslant)$.
Output: The number of downsets in $(X, \leqslant)$.

$\#1P1NSAT$:
Input: A boolean formula ϕ in CNF, such that every clause has at most one unnegated literal and at most one negated literal.
Output: The number of satisfying assignments to ϕ.

$\#BEACHCONFIGS$:
Input: A graph G.
Output: The number of "Beach configurations" in G, i.e. P_4^*-colourings of G, where P_4^* denotes the path of length 3 with loops on all three vertices.

All the above problems lie in a syntactically restricted subclass $\#RH\Pi_1 \subseteq \#\Pi_1$. Furthermore, they characterize $\#RH\Pi_1$ in the sense of being complete for $\#RH\Pi_1$ with respect to AP-reducibility. We say that a problem is in $\#RH\Pi_1$ if it can be expressed in the form $f(\mathcal{A}) = |\{(\mathbf{T}, \mathbf{z}) : \mathcal{A} \models \forall \mathbf{y} \psi(\mathbf{y}, \mathbf{z}, \mathbf{T})\}|$, where ψ is an unquantified CNF formula in which each clause has at most one occurence of an unnegated relation symbol from $\mathbf{T}$ and at most one negated relation symbol from $\mathbf{T}$. The rationale behind the naming of the class $\#RH\Pi_1$ is as follows: "Π_1" indicates that only universal quantification is allowed, and "RH" indicates that the unquantified subformula ψ is in "restricted Horn" form. Note that the restriction on clauses of ψ applies only to terms involving symbols from $\mathbf{T}$. For example:

$f_{DS}(\mathcal{A}) = |\{(D) : \mathcal{A} \models \forall x \forall y \big(D(x) \wedge y \leqslant x \to D(y)\big)\}|$, where we represent an instance of $\#DOWNSETS$ as a structure $\mathcal{A} = (|\mathcal{A}|, \leqslant)$, where $\leqslant$ is a binary relation assumed to be a partial order. The downset is represented by a unary relation D.

$f_{1P1NSAT}(\mathcal{A}) = |\{(T) : \mathcal{A} \models \forall x \forall y \big(T(x) \wedge x \sim y \to T(y)\big)\}|$, where we represent an instance of $\#1P1NSAT$ as a structure $\mathcal{A} = (|\mathcal{A}|, \sim)$, where $\sim$ is a binary relation and $x \sim y$ represents that the variables x, y exist in the same clause, x negated and y unnegated. The idea here is that $\neg x \vee y$ is equivalent to $x \to y$. Note also that if the input formula contains clauses with one literal we can eliminate them, as they do not add more truth assignments to the formula. The truth assignment is represented by a unary relation T.

$f_{BIS}(\mathcal{A}) = |\{(I) : \mathcal{A} \models \forall x \forall y \big(L(x) \wedge x \sim y \wedge I(x)\big) \to I(y)\}|$, where we represent an instance of $\#BIS$ as a structure $\mathcal{A} = (|\mathcal{A}|, L, \sim)$, where L is the set of "left" vertices and $\sim$ is a binary relation assumed to represent adjacency. An independent set includes the left vertices for which the unary relation I is "true" and the right vertices for which I is "false". Counting all the relation symbols which satisfy the above formula is equivalent to counting all the independent sets of a bipartite graph.

$f_{P_4^*}(\mathcal{A}) = |\{(C_1, C_2, C_3) : \mathcal{A} \models \forall x \forall y \big(C_1(x) \to C_2(x)\big) \wedge \big(C_2(x) \to C_3(x)\big) \wedge \big(C_1(x) \wedge x \sim y \to C_2(y)\big) \wedge \big(C_2(x) \wedge x \sim y \to C_3(y)\big)\}|$, where C_j is "true" for a vertex iff its colour is in $\{c_1, c_2, ..., c_j\}$. Obviously a vertex belongs to none of the relations C_1, C_2, C_3 iff its colour is c_4.

Theorem 5.5.1 *$\#1P1NSAT$ is complete for $\#RH\Pi_1$ under parsimonious reducibility.*

Proof. Consider a problem in $\#RH\Pi_1$, $f(\mathcal{A}) = |\{(\mathbf{T}, \mathbf{z}) : \mathcal{A} \models \forall \mathbf{y} \psi(\mathbf{y}, \mathbf{z}, \mathbf{T})\}|$, and suppose $\mathbf{T} = (T_0, ..., T_{r-1})$, $\mathbf{y} = (y_0, ..., y_{l-1})$ and $z = (z_0, ..., z_{m-1})$, where T_i are relations of arity a_i and y_j, z_k are first-order variables. Let $L = \|\mathcal{A}\|^l$ and $M = \|\mathcal{A}\|^m$ and $(\eta_0, ..., \eta_{L-1})$ and $(\zeta_0, ..., \zeta_{M-1})$ be enumerations of $|\mathcal{A}|^l$ and $|\mathcal{A}|^m$ respectively. Then, $\mathcal{A} \models \forall \mathbf{y} \psi(\mathbf{y}, \mathbf{z}, \mathbf{T})$ iff $\mathcal{A} \models \bigwedge_{q=0}^{L-1} \psi(\eta_q, \mathbf{z}, \mathbf{T})$.

If we replace $\mathbf{z}$ by some ζ_s in $\bigwedge_{q=0}^{L-1} \psi(\eta_q, \mathbf{z}, \mathbf{T})$, we obtain $\bigwedge_{q=0}^{L-1} \psi(\eta_q, \zeta_s, \mathbf{T})$. It suffices to count the relations $\mathbf{T}$ such that $\mathcal{A} \models \bigwedge_{q=0}^{L-1} \psi(\eta_q, \zeta_s, \mathbf{T})$, for each $s = 0, ..., M-1$, and sum these values, i.e. $f(\mathcal{A}) = \sum_{s=0}^{M-1} \big|\big\{\mathbf{T} : \mathcal{A} \models \bigwedge_{q=0}^{L-1} \psi(\eta_q, \zeta_s, \mathbf{T})\big\}\big|$.

Now, $\psi_{q,s}(\mathbf{T})$ is obtained from $\psi(\eta_q, \zeta_s, \mathbf{T})$ by replacing every subformula that is true (resp. false) in $\mathcal{A}$ by TRUE (resp. FALSE) and eliminating the clauses containing TRUE-literals (resp. eliminating the FALSE-literals from their clauses). The formula $\bigwedge_{q=0}^{L-1} \psi_{q,s}(\mathbf{T})$ is in CNF form with propositional variables $T_i(w_i)$, where $w_i \in |\mathcal{A}|^{a_i}$, and there is at most one occurrence of an unnegated propositional variable in each clause and at most one of a negated variable. To obtain a precise correspondence we must add, in each instance, trivial clauses $T_i(w_i) \to T_i(w_i)$ for every propositional variable $T_i(w_i)$ not already occurring in

$\bigwedge_{q=0}^{L-1} \psi_{q,s}(\mathbf{T})$, otherwise the number of $\mathbf{T}$ will be underestimated by a factor 2^u, where u is the number of unrepresented variables $T_i(w_i)$.

The value $f(\mathcal{A})$ is the sum of the numbers of satisfying assignments to M(polynomially many) instances of $\#1P1NSAT$. Thus, the above procedure provides an AP-reduction to $\#1P1NSAT$. In the rest of the proof, we modify this reduction in order to have a parsimonious reduction from a problem in $\#RH\Pi_1$ to $\#1P1NSAT$.

First, we distinguish the variables in the above set of instances of $\#1P1NSAT$ as $T_i{}^s(w_i)$, $s = 0, 1, ..., M-1$. We distinguish the instances as $\Psi^s = \bigwedge_{q=0}^{L-1} \psi_{q,s}(\mathbf{T})$, $s = 0, 1, ..., M-1$. We assume that each Ψ^s does not contain any one-literal clause, since the the truth setting of any such literal is forced. Let $v_1, ..., v_{M-1}$ be new propositional variables. Let $\Phi^s = \bigwedge_{i=0}^{r-1} \wedge_{w_i \in |\mathcal{A}|^{a_i}} (T_i{}^s(w_i) \rightarrow v_{s+1})$, $s = 0, 1, ..., M-2$, and $\Xi^s = \bigwedge_{i=0}^{r-1} \bigwedge_{w_i \in |\mathcal{A}|^{a_i}} (v_s \rightarrow T_i{}^s(w_i))$, $s = 1, ..., M-1$.

The formula $\phi = \bigwedge_{s=0}^{M-1} \Psi^s \wedge \bigwedge_{s=0}^{M-2} \Phi^s \wedge \bigwedge_{s=1}^{M-1} \Xi^s$ has $f(\mathcal{A})$ satisfying assignments. To see this, note that if, for a given s and for some i, $T_i{}^s(w_i)$ is assigned TRUE, then $T_i{}^p(w_j)$ must be assigned TRUE for every $p > s$ and for every j because of the formulae Φ^s and Ξ^s. There can only be one s such that the variables $T_i{}^s(w_i)$, $i = 0, ..., r-1$, receive both truth assignments. This is the unique s such that $v_s = FALSE$ and $v_{s+1} = TRUE$. And this is the case that we count the pairs $(\mathbf{T}, \zeta_s)$ satisfying the initial formula: When some s is fixed, ϕ is satisfied if and only if Ψ^s is satisfied. Since the satisfying assignments are disjoint for different s, we get the number of pairs $(\mathbf{T}, \mathbf{z})$ satisfying the initial formula. ■

Corollary 5.5.1 *The problems $\#P_4 - COL$, $\#DOWNSETS$, $\#1P1NSAT$, $\#BEACHCONFIGS$ are all complete for $\#RH\Pi_1$ with respect to AP-reducibility.*

Corollary 5.5.1 gives information about the class $\#RH\Pi_1$. It is likely to be a strict subset of $\#\Pi_1$. If $\#RH\Pi_1 = \#\Pi_1$ then $\#IS \leqslant_{AP} \#BIS$ and $\#BIS =_{AP} \#SAT$, since $\#IS \in \#\Pi_1$ and $\#IS =_{AP} \#SAT$. Moreover, $\#RH\Pi_1$ is not a subset of $\#\Sigma_1$. In particular, $\#1P1NSAT \in \#RH\Pi_1 \setminus \#\Sigma_1$ in the same way that $\#3SAT \in \#\Pi_1 \setminus \#\Sigma_1$. Obviously, all the problems in $\#\Sigma_1$ are AP-reducible to complete problems in $\#RH\Pi_1$, but it is not known whether $\#\Sigma_1 \subset \#RH\Pi_1$.

5.5.1 #BIS and the Approximation Complexity Hierarchy

The question of the approximation complexity of $\#BIS$ remains open. But the conjecture that lies between "FPRASable" problems and $\#SAT$ was strengthened by Bordewich's result [7]. Under the assumption that $NP \neq RP$, there are counting problems which neither admit an FPRAS nor $\#SAT$ is AP-reducible to them, i.e. are of intermediate approximation complexity between "FPRASable" and $\#SAT$.

The proof of this proposition is based on the proof of Ladner's theorem [37]. The main result is given below.

Proposition 5.5.1 *If $NP \neq RP$ then there are an infinite number of problems $\pi_{A_1}, \pi_{A_2}, ...$ in $\#P$ such that*

(i) for all i, π_{A_i} does not have an FPRAS,
(ii) for all i, $\#SAT$ is not AP-reducible to π_{A_i}, and
(iii) for all i, j such that $i < j$, we have $\pi_{A_j} <_{AP} \pi_{A_i}$.

Assuming $NP \neq RP$, there are infinitely many complexity levels between efficient approximability and the inapproximability of $\#SAT$. Moreover, if $\#BIS$ is genuinely in the middle ground, then there are problems that do not admit an FPRAS, are not equivalent in approximation complexity to $\#BIS$ and are not AP-interreducible with $\#SAT$, thus also occupy the middle ground.

5.6 Descriptive Complexity and Log-space Counting Classes

We have already shown that every problem in $\#\Sigma_0$ can be computed in deterministic polynomial time, and every problem in $\#\Sigma_1$ has an FPRAS. In this section, we relate these two logically defined classes to Turing machine based counting classes.

First, we describe an encoding e of ordered σ-structures into binary strings. Let $(\mathcal{A}, \leqslant)$ and $n = \|\mathcal{A}\|$. Then $e(\mathcal{A}, \leqslant)$ starts with 1^n0. We use the enumeration induced by $\leqslant$ to encode the constants and the relations in binary strings. For example, let R be a k-ary relation, then in the encoding of R the bit at position $\sum_{i=1}^{k} x_i \cdot n^{i-1}$ is 1 iff $(x_1, x_2, ...x_k) \in R$. Also, we encode a τ-expansion $\mathbf{T}$ of $\mathcal{A}$ using the enumeration induced by $\leqslant$. For all functions $f \in \#\mathcal{FO}$ let $e(f)$ be the function that for all $x \in \{0,1\}^*$, $e(f)(x) = f(\mathcal{A}, \leqslant)$ if $x = e(\mathcal{A}, \leqslant)$ and $f(x) = 0$ otherwise. The function $e(f)$ is well-defined since our encoding is one-to-one. For any class of first-order counting functions $\mathcal{C} \subseteq \#\mathcal{FO}$ let $e(\mathcal{C})$ be the class of functions $e(f) : \{0,1\}^* \to \mathbb{N}$ for $f \in \mathcal{C}$. The encoding e can be handled by a Turing machine working in logarithmic space.

In section 3.2 we defined the classes $\# \cdot 1L$ (or $\#\Sigma_0 1L$), $\# \cdot 1NL$ (or $\#\Sigma_1 1L$), $\#\Sigma_k L$, and $\#\Pi_k L$ for $k \geqslant 1$. These classes consist of functions that “count” the second argument of binary relations in 1-1-$\Sigma_0 L$, 1-1-$\Sigma_1 L$, 1-1-$\Sigma_k L$, and 1-1-$\Pi_k L$ respectively.

Theorem 5.6.1 *[9] It holds that:*
(i) $e(\#\Sigma_0) \subseteq \#\Sigma_0 1L = \#1L$.
(ii) $e(\#\Sigma_1) \subseteq \#\Sigma_1 1L = \#span\text{-}1L$.
(iii) $e(\#\Pi_1) \subseteq \#\Pi_1 1L$.

Proof. (i) Let $e(f) \in e(\#\Sigma_0)$, and let ϕ be the Σ_0 formula with relation symbols from $\sigma \cup \mathbf{T}$, such that $f(\mathcal{A}, \leqslant) = |\{\mathbf{T} : (\mathcal{A}, \mathbf{T}, \leqslant) \models \phi\}|$ for all ordered σ-structures $(\mathcal{A}, \leqslant)$. We define the binary relation R_ϕ such that $(e(\mathcal{A}, \leqslant), e(\mathbf{T}, \leqslant)) \in R_\phi$ iff $(\mathcal{A}, \mathbf{T}, \leqslant) \models \phi$. Since our encoding is one-to-one, we have $e(f)(x) = |y : (x, y) \in R_\phi\}|$ for every $x \in \{0,1\}^*$. It is now sufficient to show that $R_\phi \in$ 1-1-$\Sigma_0 L$. We describe a log-space bounded 1-1-Σ_0-Turing machine M_ϕ that accepts R_ϕ. Since ϕ is a Σ_0 formula and has no variables, M_ϕ has to read a constant number of bits in order to evaluate ϕ. The machine makes a sweep over the two inputs and copies on its work tape the constants and the members of relations that needs for the evaluation of ϕ. Then M_ϕ checks deterministically that the representation is correct and that ϕ holds.

(ii) We define the relation R_ϕ exactly as in (i). In this case the formula ϕ has existentially quantified variables. A log-space bounded 1-1-Σ_1-Turing machine M_ϕ after reading the inputs, keeps the elements needed, guesses assignments to the variables and evaluates ϕ as above.

(iii) M_ϕ works similar to (ii), but instead of existentially guessing the variable assignment, it uses universal branches. This can be done by a 1-1-Π_1-Turing machine. ■

In the above proof, it was more convenient to count only relations $\mathbf{T}$ satisfying a formula instead of counting relations and first-order variables. As we noted below Theorem 5.3.1 we are allowed to do that.

We show that the above inclusions are strict. We define the following problems.

#DEG_1_NGB:
Input: A graph G.
Output: The number of nodes of G that have a neighbour of degree 1.

#DIST2:
Input: A graph G.
Output: The number of pairs of nodes of G for which the shortest path in the graph is of length two.

Since $f_{\#DEG\_1\_NGB}(\mathcal{A}) = |\{z : \mathcal{A} \models \exists x\big(zEx \wedge \forall y(y \neq x \rightarrow \neg(zEy))\big)\}|$, and $f_{\#DIST2}(\mathcal{A}) = |\{(z_1, z_2) : \mathcal{A} \models \exists x\big(z_1Ex \wedge xEz_2 \wedge \neg(z_1Ez_2)\big)\}|$, it holds that $\#DEG\_1\_NGB \in \#\Sigma_2$, and $\#DIST2 \in \#\Sigma_1$. It can be proved that $\#DEG\_1\_NGB \notin \#\Sigma_1$, and $\#DIST2 \notin \#\Sigma_0$ using model theoretic arguments similar to those in [44].

- $\#DEG\_1\_NGB \in \textit{span}\text{-}1L \setminus \#\Sigma_1$: Given a graph G as first input and a node v with a neighbour of degree 1 as second input, a nondeterministic 1-1- Turing machine guesses a neighbour u of the node v and checks if u has degree 1 reading the first input from left to right. This means that $\#DEG\_1\_NGB \in \#\Sigma_1 1L = \textit{span}\text{-}1L$.

- $\#DIST2 \in \#1L \setminus \#\Sigma_0$: Given a graph G as first input and a path (u, v, w) of length two as second input, a deterministic 1-1-Turing machine reads the second input and copies the path on its work tape. Then it reads the first input one-way and checks that there are the edges (u, v), (v, w), and that there is no edge (u, w) in the graph. So, $\#DIST2 \in \#\Sigma_0 1L = \#1L$.

Chapter 6

Conclusions

For further study and research many problems are open.

First we should examine all the possible results that the class *span*-L could give. An open problem is the existence of a polynomial-time approximation algorithm for $\#NFA$. Alternatively, some problem already owning an FPRAS could be *span*-L complete under $\leqslant_m^l$. Concerning descriptive complexity, the class *span*-L could be shown to coincide with some subclass of $\#P$. A decision problem for NFA is expressed in MSO (Monadic Second Order Logic).

Moreover, research around $\#BIS$ is strong. A trichotomy for approximately counting H-colorings has recently showed up (Galanis, Goldberg, Jerrum 2016). It would be useful to find other problems belonging to $\#RH\Pi_1$.

Other classes considered in this thesis should be enriched. Classes like $TotP$, *span*-L, $\#L$ give an insight in feasible computation and in the consequences of space and time restrictions. For example a complete problem for $TotP$ under parsimonious reductions would give information about the real computational difficulty of this class.

Bibliography

[1] C. Àlvarez and B. Jenner, A very hard log-space counting class. Theoretical Computer Science 107 (1993), 3-30.

[2] C. Alan, The recognition problem for the set of perfect squares. Proceedings of the 7th Annual IEEE Symposium on Switching and Automata Theory (1966), 78-87.

[3] F. Ablayev, C. Moore and C. Pollett, Quantum and stochastic branching programs of bounded width. Proc. 29th Intl. Colloquium on Automata, Languages and Programming Lecture Notes in Computer Science, Springer-Verlag (2002), 343-354.

[4] E. Allender and M. Ogihara, Relationships among PL, #L, and the Determininant. In 9th Annual Structure in Complexity Theory Conference (1994), 267-279.

[5] D. Barrington, Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. JCSS, 38(1) (1989), 150-164.

[6] J.L. Balcázar, R.V. Book and U. Shönning, The polynomial-time hierarchy and sparse oracles. J. ACM 33 (1986), 603-617.

[7] M. Bordewich, On the Approximation Complexity Hierarchy. Approximation and Online Algorithms, 8th International Workshop, WAOA 2010, Liverpool, UK, 37-46.

[8] J. R. Bunch and J. E. Hopcroft, Triangular Factorization and Inversion by Fast Matrix Multiplication. Mathematics of Computation 28 (125)(1974), 231-236.

[9] H. Burtschick, Comparing Counting Classes for Logspace, One-way Logspace, and First-order. MFCS, 139-148, (1995).

[10] S. R. Buss, The boolean formula value problem is in ALOGTIME. 19^{th} ACM STOC Symp. (1987), 123-131.

[11] H. Caussinus, P. McKenzie, D. Thérien and H. Vollmer, Nondeterministic NC^1 computation. JCSS 57 (1998), 200-212.

[12] S. A. Cook, A Taxonomy of Problems with Fast Parallel Algorithms. Information and Control 64 (1985), 2-22.

[13] A. Chiu, G. Davida and B. Litow, Division in logspace-uniform NC^1. RAIRO Theoretical Informatics and Applications 35 (2001), 259-276.

[14] A. K. Chandra, L. Stockmeyer and U. Vishkin, Constant Depth Reducibility. SIAM J. Computing 13 (1984), 423-439.

[15] C. Damm, $DET = L^{\#L}$?. Informatik-Preprint 8, Fachbereich Informatik der Humboldt Universität zu Berlin (1991).

[16] M. Dyer, L. A. Goldberg, C Greenhill and M. Jerrum, On the relative complexity of approximate counting problems. Algorithmica 38 (3) (2003), 471-500.

[17] M. Dyer and C. Greenhill, The complexity of counting graph homomorphisms. Random Structures and Algorithms 17 (2000), 260-289.

[18] H. B. Enderton, A Mathematical Introduction to Logic. Academic Press (2000).

[19] S. Fenner, L. Fortnow and S. Kurtz, Gap-definable counting classes. Journal of Computer and System Sciences 48(1) (1994), 116-148.

[20] R. Fagin, Generalized first-order spectra and polynomial time recognizeble sets. Complexity of Computations, SIAM-AMS Proc. 7 (1974), 43-73.

[21] A. Galanis, L. A. Goldberg and M. Jerrum, Approximately Counting H-Colourings is $\#BIS$-Hard. ICALP 1 (2015), 529-541.

[22] S. Grollmann, A. L. Selman, Complexity measures for public-key crypto-systems. 25th FOCS (1984), 495-503.

[23] P. Hell and J. Nešetřil, On the complexity of H-coloring. Journal of Combinatorial Theory, Series B 48 (1990), 92-110.

[24] L. A. Hemaspaandra, H. Vollmer, The Satanic Notations: Counting classes beyond #P and Other Definitional Adventures. SIGACT News, Volume 26 Issue 1 (1995), 2-13.

[25] N. Immerman, Descriptive Complexity. Springer, Graduate Texts in Computer Science (1999).

[26] N. Immerman, Nondeterministic space is closed under complement. SIAM J. Comput. 17 (1988), 935-938.

[27] N.D. Jones, E. Lien and W.T. Laaser, New problems complete for nondeterministic log-space. Math. Systems Theory 10 (1976), 1-17.

[28] S. Jukna, Boolean Function Complexity - Advances and Frontiers. Algorithms and combinatorics 27, Springer (2012).

[29] H. Jung, Depth efficient transformations of arithmetic into Boolean circuits. In Proc. FCT 199 in Lecture Notes in Computer Science, Springer (1985), 167-173.

[30] M. R. Jerrum, L. G. Valiant and V.V. Vazirani, Random Generation of Combinatorial Structures from a Uniform Distribution. Theoretical Computer Science 43 (1986), 169-188.

[31] J. Köbler, U. Shönning and J. Torán, On counting and approximation. Acta Inform. 26 (1989), 363-379.

[32] A. Krebs, N. Limaye and M. Mahajan, Counting paths in VPA is complete for $\#NC^1$. In Proceedings of COCOON 2010, volume 6196 of LNCS, Springer (2010), 44-53.

[33] R. M. Karp, M. Luby and N. Madras, Monte-Carlo Approximation Algorithms for Enumeration Problems. J. Algorithms 10 (3) (1989), 429-448.

[34] R. M. Karp and M. Luby, Monte-Carlo Algorithms for Enumeration and Reliability Problems. Proc. 24^{th} IEEE FOCS (1983), 56-64.

[35] S. Kannan, Z. Sweedyk, S. Mahaney, Counting and Random Generation of Strings in Regular Languages. SODA (1995), 551-557.

[36] N. Limaye, M. Mahajan and B. V. Raghavendra Rao, Arithmetizing Classes Around NC^1 and L. Theory Comput. Syst. 46(3) (2010), 499-522.

[37] R. E. Ladner, On the Structure of Polynomial Time Reducibility. Journal of Combinatorial Theory, Series B 48, 92-110 (1990).

[38] A. Pagourtzis and S. Zachos, The Complexity of Counting Functions with Easy Decision Version. MFCS (2006), 741-752.

[39] C. H. Papadimitriou, Computational Complexity. Addison-Wesley (1994).

[40] O. Reingold, Undirected ST-connectivity in log-space. STOC'05: Proceedings of the 37th Annual ACM Symposium on Theory of Computing. ACM, New York, 376-385.

[41] K. Reinhardt and E. Allender, Making Nondeterminism Unambiguous. In 38th IEEE Symposium on Foundations of Computer Science (FOCS) (1997), 244-253.

[42] A.J. Sinclair and M.R. Jerrum, Approximate counting, uniform generation and rapidly mixing Markov chains. Information and Computation, 82 (1989), 93-133.

[43] S. Skyum and L. G. Valiant, A complexity theory based on Boolean Algebra. Proc. 22^{nd} IEEE Symposium on Foundations of Computer Science (1981), 244-253.

[44] S. Saluja, K. V. Subrahmanyam and M. Thakur, Descriptive Complexity of $\#P$ functions. Proc. 7th IEEE Symposium on Structure in Complexity Theory (1992).

[45] R. Szelepcsényi, The method of forced enumeration for nondeterministic automata, Acta Inform. 26 (1979), 279-284.

[46] S. Toda, On the computational power of PP and $\oplus P$. Proc. 30th IEEEE Symp. on the Foundations of Computer Science (1989), 514-519.

[47] S. Toda, Computational complexity of counting complexity classes. PhD Thesis, Tokyo Institute of Technology, Department of Computer Science, Tokyo, Japan (1991).

[48] S. Toda, Counting problems computationally equivalent to computing the determinant. Technical Report CSIM 91-07, Dept. Comp. Sci. and Inf. Math., Univ. of Electro- Communications, Tokyo (1991).

[49] L.G. Valiant, The complexity of enumeration and reliability problems. SIAM J. Comput. 8 (1979), 410-421.

[50] L.G. Valiant, The complexity of computing the permanent. Theoretical Computer Science 8 (1979), 189-201.

[51] H. Vollmer, Komplexitätsklassen von Functionen. PhD thesis, Universität Würzburg, Institut für Informatik, Würzburg, Germany (1994).

[52] H. Venkateswaran, Circuit definitions of non-deterministic complexity classes. SIAM Journal on Computing 21 (1992), 655-670.

[53] H. Vollmer, Introduction to Circuit Complexity: A Uniform Approach. Springer-Verlag New York Inc. (1999).

[54] L. Valiant, The complexity of computing the Permanent. Theoretical Computer Science 8 (1979), 189-201.

[55] V. Vinay, Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits. Proc. 6th IEEE Structure in Complexity Theory Conference (1991), 270-284.

[56] L. Valiant, Why is Boolean complexity theory difficult? Boolean Function Complexity, London Mathematical Society Lecture Notes Series 169, Cambridge University Press (1992).

[57] D. Zuckerman, On unapproximable versions of NP-complete problems. SIAM Journal on Computing 25 (1996), 1293-1304.

[58] Descriptive Complexity Theory. In Wikipedia, `https://en.wikipedia.org/wiki/Descriptive_complexity_theory`.