DeepScience

The Complexity of Logarithmic Space Bounded Counting Classes

(Second Edition)

T. C. Vijayaraghavan

The Complexity of Logarithmic Space Bounded Counting Classes
(Second Edition)

T. C. Vijayaraghavan

DeepScience

Published, marketed, and distributed by:

Deep Science Publishing, 2026
USA | UK | India | Turkey
Reg. No. MH-33-0658412
www.deepscienceresearch.com
editor@deepscienceresearch.com
WhatsApp: +91 7977171947

ISBN: 978-93-7185-303-3

E-ISBN: 978-93-7185-918-9

https://doi.org/10.70593/978-93-7185-918-9

Citation: Vijayaraghavan, T. C. (2026). *The Complexity of Logarithmic Space Bounded Counting Classes (Second Edition)*. Deep Science Publishing. https://doi.org/10.70593/978-93-7185-918-9

2010 *Mathematics Subject Classification. Primary:* 68Q10, 68Q15

Key words and phrases. Structural Complexity, Computational Complexity, Counting classes, Logarithmic space bounded counting classes, Theory of Computation

ABSTRACT. In this monograph, we study complexity classes that are defined using $O(\log n)$-space bounded non-deterministic Turing machines. We prove salient results of Computational Complexity in this topic such as the Immerman-Szelepcsenyi Theorem, the Isolating Lemma, theorems of Meena Mahajan and V. Vinay on the determinant and many consequences of these very important results. The manuscript is intended to be a comprehensive textbook on the topic of The Complexity of Logarithmic Space Bounded Counting Classes.

Contents

Preface to the Second Edition

The second edition of this monograph was brought about due to a pressing need to remove the incorrect result that $\mathrm{NL} = \mathrm{C_{=}L}$, which I claimed to be true using a buggy proof in the first edition of this monograph. I am extremely grateful to Jacobo Toran for pointing out this error even before the first edition was published. I however did not understand the argument given by Jacobo Toran all by myself because of keeping sight on other portions of this book. However after publishing the first edition and discussing with Eric Allender who was very patient in explaining the error in my proof, in which I claimed that $\mathrm{C_{=}L} \subseteq \mathrm{NL}$ and with a bit more diligence on my part, I understood that the result I have claimed is incorrect. Consequences of assuming the statement which are incorrect in various chapters have also been taken care of and corrected to the best of my knowledge.

Eric Allender gave invaluable comments about many other portions of the first edition, especially in pointing out in Chapter 2, the result $\mathrm{NL/poly} = (\mathrm{UL} \cap \text{co-UL})\mathrm{/poly}$ requires some more refinement that polynomially many weight functions are required as an advice string to isolate a directed path from s to t in the input directed graph, helped me gain clarity and proper understanding of the results on this topic. I am extremely grateful to Eric Allender for careful proof reading of major portions of the second chapter of this book. I once again thank Jacobo Toran and Eric Allender for proof reading many other portions of this book and sending me their valuable comments.

Shortly after publishing the first edition, I also felt the need to give more precision to the introductory material contained in Chapter 1 of this monograph. In Chapter 1, we have included some more basic material starting from definitions of deterministic and non-deterministic Turing machines as in [**Koz97**], refined the notion of a configuration of a space bounded Turing machine by defining mini configuration and succinct configuration, and to state explicitly and justify observations concerning definitions of these two types of configurations. Following this in Section 1.2 of Chapter 1, we have stated certain very useful complexity upper bound results from [**CDL01**] based on Boolean circuits and the Chinese remainder representation. Following this, we have included an useful subsection on logarithmic space bounded computation.

Certain important typographical errors in various results or algorithms such as in line 26 of the algorithm of Theorem 2.18 which existed in the first edition have been corrected. In page 8 of Chapter 1, I have stated that we usually assume that the computation binary tree of a non-deterministic Turing machine M is a complete binary tree. However, I have clarified in page 29 of Chapter 2 that this is not always

the case; that is, the computation tree of a non-deterministic Turing machine is not always a complete binary tree since this assumption seems to be unrealistic in the context of some logarithmic space bounded counting classes, especially GapL. We carry forward this word of caution in many places in following chapters of this monograph. Sections 2.3 and 2.4 of Chapter 2 have been thoroughly revised.

Chapter 3 remains almost the same as in the first edition except that a theorem of Kummer and its corollary, the Prime Number Theorem and the Chinese Remainder Theorem which are useful for showing closure properties of $\mathrm{Mod}_p\mathrm{L}$ and ModL have been explicitly stated. Chapter 4 also remains almost the same.

Chapter 5 has been thoroughly revised with most notable changes being consequences of dropping the incorrect result that $\mathrm{C_{=}L} \subseteq \mathrm{NL}$. The list of complete problems given in the second edition now separately includes problems that are logspace many-one complete for $\mathrm{C_{=}L}$. Table 5.1 has been revised and Section 5.9 does not deal with the $\mathrm{C_{=}L}$ hierarchy or Boolean circuits that have $\mathrm{C_{=}L}$ oracle gates. Chapter 5 ends with the Notes section which contains Figure 5.1 that shows the landscape of important logarithmic space bounded counting classes. This figure has also been carefully redrawn.

Chapter 6 has undergone major revision in terms of the manner in which we have introduced basic concepts on permutations in Section 6.1. We have introduced certain terminologies such as orbicycle, 2-orbicycle, and orbicycle cover. I felt that using the same term "cycle" in the context of permutations and in the context of directed graphs is obviously confusing. To solve this problem, I decided to introduce the notion of a cycle in a permutation and call it as an orbicycle. The notion of a 2-cycle (also known as a transposition in the first edition) is called as just 2-orbicycle here in the second edition. An even permutation and an odd permutation are defined based on the parity of the number of 2-orbicycles. Some more examples are included in Section 6.1. The notion of the sign of a permutation is defined in Definition 6.19 and it is interestingly justified in Theorem 6.21. The notion of an orbicycle cover is related to a cycle cover of a directed graph in Definitions 6.38-6.39 and in Lemma 6.40 in Section 6.2. Rest of Section 6.2 remains the same. Section 6.3 discusses complexity upper bound results on problems in Linear Algebra which follow as a consequence of the logspace many-one completeness of computing the determinant of integer matrices for GapL. The complexity upper bound results have been revised following us noting that $\mathrm{C_{=}L} \subseteq \mathrm{NL}$ has been incorrectly claimed to be true in the first edition.

Exercise problems of all chapters have been carefully looked into and changes have been made to correct any error which might have been caused due to incorrect statements in the first edition. Notes of all chapters have been read carefully. Any error which remains in this manuscript is due to me, solely.

It is interesting to note that even though this topic of logarithmic space bounded counting classes is a branch of computational complexity and structural complexity, it borrows notions, ideas and results from many other areas of Mathematics. The second edition of this book includes a small Appendix A on Mathematical prerequisites which contains separate sections for Number Theory, Asymptotic notation and Basics of Algebra & notation.

I am once again indebted to V. Arvind for his guidance and continuous encouragement in helping me prepare this manuscript. Any useful comments regarding this book may be sent to `tcv.tclsbc@gmail.com`.

Chennai, *T. C. Vijayaraghavan,*
India. *March 2026.*

Preface to the First Edition

It is a fact that the study of complexity classes has grown enormously and an innumerable number of results have been proved on almost every complexity class that has been defined. A counting class is defined as any complexity class whose definition is based on a function of the number of accepting computation paths and/or the number of rejecting computation paths of a non-deterministic Turing machine. By restricting the space used by a non-deterministic Turing machine to be $O(\log n)$, where n is the size of the input, we can define many logarithmic space bounded counting complexity classes. The first and fundamental logarithmic space bounded counting complexity class that one can easily define is Non-deterministic Logarithmic space (NL). The definition of any other logarithmic space bounded counting complexity class is based on NL.

The purpose of this research monograph is to serve as a textbook for teaching the topic of logarithmic space bounded counting complexity classes and almost all the salient results on them. In this monograph, we introduce the beautiful and sophisticated theory of logarithmic space bounded counting complexity classes. In Chapter 1, we briefly introduce the Turing machine model and the Boolean circuit model of computation. In Chapters 2 to 5 of this monograph, we define logarithmic space bounded counting complexity classes and we prove structural properties of these complexity classes. In particular we study some important results on a number of complexity classes whose definitions are based on Turing machines and which are contained between the circuit complexity classes NC^1 and NC^2. The complexity classes of interest to us are NL, $\sharp$L, GapL, C$_=$L, UL, Mod$_p$L, Mod_kL, ModL and PL, where $k, p \in \mathbb{N}$, $k \geq 2$ and p is a prime. Results we prove in Chapters 2 to 5 are diverse in the ideas and techniques involved such as the following:

- ⋆ the non-deterministic counting method invented to prove NL = co-NL which is the Immerman-Szelepcsenyi Theorem in the logarithmic space setting and some of its useful consequences which is to show that $\mathrm{L}^{\mathrm{NL}} = \mathrm{NL}$ and NL/poly = UL/poly in Chapter 2,
- ⋆ using the Isolating Lemma to show that NL/poly = (UL ∩ co-UL)/poly in Chapter 2,
- ⋆ by proving closure properties of $\sharp$L and GapL, and using results from elementary number theory we prove many interesting closure properties of Mod$_p$L and a characterization of Mod_kL in Chapter 3, where $k, p \in \mathbb{N}$, $k \geq 2$ and p is a prime,

- ⋆ the double inductive counting method used to prove a combinatorial closure property of $\sharp$L under the assumption that $\mathrm{NL} = \mathrm{UL}$ in Chapter 2 and its implications for ModL in Chapter 3, and
- ⋆ using polynomials to approximate the sign of a GapL function and its applications to show closure properties of PL in Chapter 4 such as the closure of PL under logspace Turing reductions.

In Chapter 5, we list a set of problems which are complete for logarithmic space bounded counting classes under logspace many-one reductions, all of which are based on the results that we have discussed in Chapters 2 to 4. In Chapter 5, we also define hierarchies of logarithmic space bounded counting complexity classes and complexity classes based on Turing reductions that involve Boolean circuits containing oracle gates for various logarithmic space bounded counting complexity classes. We show that these two notions coincide for all logarithmic space bounded counting complexity classes and as a consequence of the results shown in Chapters 2 to 4, some of the hierarchies also collapse to their logarithmic space bounded counting complexity class itself. In Chapter 6 of this monograph, we deal exclusively with one of the very important and useful theorems and its consequences on logarithmic space bounded counting classes that computing the determinant of integer matrices is logspace many-one complete for GapL. We state and explain two very beautiful and deep theorems of M. Mahajan and V. Vinay on the determinant and also show many applications of the logspace many-one completeness of the determinant for GapL to classify the complexity of linear algebraic problems using GapL and other logarithmic space bounded counting complexity classes.

Since counting classes are defined based on non-deterministic Turing machines, invariably every theorem statement that intends to prove a property of a logarithmic space bounded counting class which we have covered in this monograph has at least one non-deterministic algorithm which has been either made explicit or explained in an easy to understand manner. We refrain from giving explicit algorithms for logspace many-one reductions since they are routine algorithms computed by deterministic Turing machines. We do not claim uniqueness over the order in which the chapters of this textbook is written since many theorems or statements proved in this monograph may have various proofs depending upon how we introduce this topic and order the results.

As a pre-requisite to understand this monograph, we assume that the student is familiar with computation using Turing machines and has undertaken a basic course on the Theory of Computation in which a proper introduction to complexity classes, reductions, notions of hardness and completeness, and oracles have been given.

I am extremely grateful to my doctoral advisor V. Arvind for the continuous encouragement and support he has given to me in pursuing research ever since I joined the Institute of Mathematical Sciences, C.I.T. Campus, Taramani, Chennai-600113 as a research scholar in Theoretical Computer Science. His invaluable guidance, ever encouraging words and timely help have rescued me from tough situations and helped me shape this research monograph. I am extremely grateful to

Jacobo Toran for a gentle proof reading of the results shown in this manuscript. His valuable comments and suggestions have significantly improved the presentation of the results shown in this monograph. I also thank Meena Mahajan for some useful discussions pertaining to the result that computing the determinant of integer matrices is logspace many-one complete for GapL.

I am extremely thankful to the higher officials of the Vels Institute of Science, Technology & Advanced Studies (VISTAS), Pallavaram, Chennai-600117 for their encouragement given to me in preparing this monograph. I also thank my colleagues in the Department of Computer Science and Information Technology, School of Computing Sciences, VISTAS and my friends for many enjoyable discussions while I was preparing this monograph.

I am extremely indebted to my mother for always being prompt and never late in cooking and providing me food everyday and in taking care and showing attention pertaining to any issue of mine if I lacked confidence and direction in tackling it. Any useful comments regarding this book may be sent by email to the email address `tcv.tclsbc@gmail.com`.

Chennai, *T. C. Vijayaraghavan.*
India.

CHAPTER 1

Introduction

1.1. The Turing machine model of computation

In this monograph, every Turing machine that we deal with has a two-way semi-infinite read-only input tape and a two-way semi-infinite read-write work tape. The input tape and the work tape have separate tape heads. Also the input tape and the work tape are assumed to be two-way to mean that corresponding tape heads can move in both left and right directions. These two tapes are assumed to be semi-infinite to mean that there are only finitely many tape cells on the left-hand side and both these tapes have infinitely many cells on the right-hand side.

Using the input tape head, the Turing machine can only read symbols in the input tape which are from the input alphabet Σ. Also using the work tape head, the Turing machine can read and write symbols from the output alphabet Γ on the work tape. We assume that both the input tape and the work tape have a de-limiter ($\vdash$) in the leftmost tape cell which should prevent tape heads from moving any further to the left. The input tape is assumed to have a de-limiter ($\dashv$) in the tape cell which is to the right of the tape cell containing the last symbol of the input string x. The input tape head is assumed not to move to the right of the right most or the last symbol of x, because it is prevented by the de-limiter ($\dashv$). We follow the standard assumption that the $\vdash$, $\dashv$ and the blank symbol $\sqcup \notin \Sigma$, where Σ is the input alphabet. In every Turing machine, it is assumed that $\Sigma \cup \{\sqcup, \vdash, \dashv\} \subseteq \Gamma$.

We also assume that the accepting state (q_{acc}) and the rejecting state (q_{rej}) of our Turing machine are unique among its set of states. Any Turing machine decides to accept or reject an input string depending on whether it enters its accepting state or rejecting state. The Turing machine stops its computation when it enters the unique accepting state or the unique rejecting state. Given $L \subseteq \Sigma^*$, we say that a Turing machine accepts L if and only if M accepts exactly those strings in L.

DEFINITION 1.1. We formally define a deterministic Turing machine as $M = (Q, \Sigma, \Gamma, \vdash, \dashv, \sqcup, \delta, q_0, F)$, where

(1) Q denotes the set of states of M,
(2) Σ is the input alphabet,
(3) $\vdash$ and $\dashv$ are de-limiters on the left-hand side and on the right-hand side of the input string on the input tape; $\vdash$ is the de-limiter on the left-hand side of the work tape,
(4) $\sqcup$ is the blank symbol,
(5) $\Gamma = \Sigma \cup \{\vdash, \dashv, \sqcup\}$ denotes the output alphabet,
(6) $q_0 \in Q$ is the initial state,

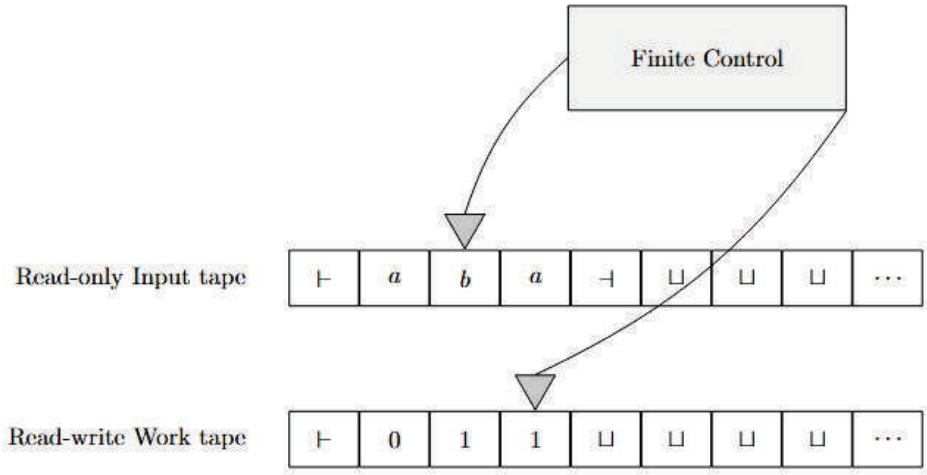

FIGURE 1.1. A Turing machine.

(7) $F = \{q_{acc}, q_{rej}\} \subseteq Q$ is the set of final states of the Turing machine, where q_{acc} denotes the unique accepting state and q_{rej} denotes the unique rejecting state, and
(8) δ is the transition function of M, defined as $\delta : \{Q \setminus F\} \times \Gamma \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\} \times \Gamma \times \{L, R\}$.

DEFINITION 1.2. We formally define a non-deterministic Turing machine as $M = (Q, \Sigma, \Gamma, \vdash, \dashv, \sqcup, \delta, q_0, F)$, where

(1) Q denotes the set of states of M,
(2) Σ is the input alphabet,
(3) $\vdash$ and $\dashv$ are de-limiters on the left-hand side and on the right-hand side of the input string on the input tape; $\vdash$ is the de-limiter on the left-hand side of the work tape,
(4) $\sqcup$ is the blank symbol,
(5) $\Gamma = \Sigma \cup \{\vdash, \dashv, \sqcup\}$ denotes the output alphabet,
(6) $q_0 \in Q$ is the initial state,
(7) $F = \{q_{acc}, q_{rej}\} \subseteq Q$ is the set of final states of the Turing machine, where q_{acc} denotes the unique accepting state and q_{rej} denotes the unique rejecting state, and
(8) δ is the transition function of M, defined as $\delta : \{Q \setminus F\} \times \Gamma \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L,R\} \times \Gamma \times \{L,R\}}$, where $2^{Q \times \Gamma \times \{L,R\} \times \Gamma \times \{L,R\}}$ denotes the set of all subsets of $Q \times \Gamma \times \{L, R\} \times \Gamma \times \{L, R\}$.

The state of a Turing machine along with the transition function is collectively called as the finite control of the Turing machine. Since any Turing machine does no further computation after entering its accepting state or its rejecting state both these states are called as *halting states*.

DEFINITION 1.3. We say that a Turing machine M is an $O(S(n))$-space bounded Turing machine if the number of tape cells of the work tape that are being read or written by the tape head of the work tape of M when it is given an input of length n is $O(S(n))$, where $S(n) \in \Omega(\log n)$.

DEFINITION 1.4. We define the complexity class L to be the class of all languages that are accepted by a $O(\log n)$-space bounded determinsitic Turing machine.

1.1.1. Configuration. Given a Turing machine M, the configuration of M is a snapshot of the computation in progress. It precisely gives information about the Turing machine which includes the state of the Turing machine, contents of the input tape and the work tape. In this monograph, since we deal exclusively with space bounded computation and any Turing machine is $O(S(n))$-space bounded, where $S(n) \in \Omega(\log n)$, we define the mini configuration of M first as follows.

DEFINITION 1.5. (**mini configuration**) Let Σ be the input alphabet, Γ be the output alphabet and let M be a Turing machine. Given an input string $x \in \Sigma^*$, we define the mini configuration of M on x as the tuple $(q, s_1, p_i, s_2 s_3, t_1, p_w, t_2 t_3)$, where $q \in Q$ denotes the state of M, $s_1, s_2, s_3 \in \Sigma$, and $t_1, t_2, t_3 \in \Omega$. The input tape head scans the p_i^{th} cell after the left-most cell which is the first cell containing the de-limiter $\vdash$ of the input tape. The cell which is scanned has the symbol s_2 with the symbol s_1 in the preceding cell and the symbol s_3 in the next cell of the input tape. Here p_i is a positive integer which is $\leq n$, where n is the size of the input. This configuration also indicates that the work tape head scans the p_w^{th} cell of the work tape after the left most cell which is the first cell of the work tape containing the de-limiter $\vdash$. This cell has the symbol t_2 with the symbol t_1 in the preceding cell and t_3 in the next cell of the work tape. Similar to p_i, here p_w is a positive integer which is $\leq k \cdot S(n)$, where k is a constant, $O(S(n))$ is the space bound of the Turing machine and n is the size of the input.

DEFINITION 1.6. (**mini initial configuration**) Let Σ be the input alphabet. Given an input string $x = s_1 s_2 \cdots s_n \in \Sigma^*$ to a Turing machine M, the mini initial configuration of M is $(q_0, \vdash, 1, s_1 s_2, \vdash, 1, \sqcup\sqcup)$.

PROPOSITION 1.7. *Let $S(n) \in \Omega(\log n)$. The amount of space required to store a mini configuration of a $O(S(n))$-space bounded Turing machine is $O(S(n))$.*

PROOF. It follows from Definitions 1.1, 1.2 and 1.5 that the amount of space required to store a mini configuration of $O(S(n))$-space bounded Turing machine is $max(O(\log n), O(S(n)) = O(S(n))$. □

Depending on the transition function of the Turing machine, it changes its state, tape heads of the input tape and the work tape move in exactly one of the either directions by exactly one tape cell and only the contents of the work tape cell scanned by its tape head are modified. As a result, the Turing machine will change from the present mini configuration to another mini configuration based on its state and the transition function.

PROPOSITION 1.8. *Let $S(n) \in \Omega(\log n)$. The number of mini configurations of a $O(S(n))$-space bounded Turing machine is upper bounded by $O(n \cdot S(n))$.*

PROOF. From Definitions 1.1, 1.2 and 1.5 it is easy to see that the number of mini configurations that can exist is upper bounded by $k \cdot |Q| \cdot |\Gamma|^3 \cdot n \cdot |\Gamma|^3 \cdot S(n) = k \cdot |Q| \cdot |\Gamma|^6 \cdot n \cdot S(n)$, where $S(n) \in \Omega(\log n)$, n is the length of the input and $k > 0$ is a constant. This is because the state of the Turing machine contributes to the factor $|Q|$ in the product, the symbols of the input tape $s_1, s_2, s_3 \in \Gamma$ and symbols $t_1, t_2, t_3 \in \Gamma$ of the work tape contribute to factor $|\Gamma|^6$ in the product, and the position of the tape heads on the respective tapes contributes to a factor of $k \cdot n \cdot S(n)$ to the product, where $k > 0$ is a constant. □

Let $S(n) \in \Omega(\log n)$. For any non-deterministic Turing machine M that uses at most $O(S(n))$ space, since there exists an upper bound on the number of mini configurations of M on any given input $x \in \Sigma^*$, we assume without loss of generality that M halts on all inputs either in the unique accepting state or in the unique rejecting state. We also assume without loss of generality that when a Turing machine has arrived at a decision regarding the membership of the input string $x = s_1 s_2 \cdots s_n$ in a language L, before it enters the accepting state or the rejecting state it does a clean up of the work tape contents. In other words, it replaces the contents of the work tape with the symbol $\sqcup$ in all cells by entering a special state called as the *clean-up state* and brings its tape heads to the left most cell of the input tape and the work tape. Following this, our Turing machine enters its unique accepting state (q_{acc}) or its unique rejecting state (q_{rej}).

DEFINITION 1.9. (**mini accepting and mini rejecting configuration**) Let Σ be the input alphabet of a Turing machine M. The mini accepting configuration of M is $(q_{acc}, \vdash, 1, s_1 s_2, \vdash, 1, \sqcup\sqcup)$ and the mini rejecting configuration of M is $(q_{rej}, \vdash, 1, s_1 s_2, \vdash, 1, \sqcup\sqcup)$, where $s_1, s_2 \in \Sigma$.

Note that unlike the configuration of a polynomial time bounded Turing machine introduced in standard texts, we do not include the entire contents of the input tape or the work tape as part of the mini configuration of our space bounded Turing machine.

Since we modify only the contents of the work tape of a Turing machine, let us try to relax this stringent condition by including entire contents of only the work tape in the configuration of a $O(S(n))$-space bounded Turing machine which we call as a *succinct configuration*, where $S(n) \in \Omega(\log n)$.

DEFINITION 1.10. (**succinct configuration**) Let Σ be the input alphabet and Γ be the output alphabet of a Turing machine M. Given an input $x = s_1 s_2 \cdots s_n \in \Sigma^*$, we define a succinct configuration of M as $(q, s_1, p_i, s_2 s_3, x_{w1}, p_w, x_{w2})$, where $q \in Q$ is a state of M, $s_1, s_2, s_3 \in \Sigma$ and $1 \leq p_i \leq n$ is a positive integer, and n is the size of the input. The input tape head scans the p_i^{th} cell after the left-most cell which is the first cell containing the de-limiter $\vdash$ of the input tape. The cell which is scanned has the symbol s_2 in it with s_1 in the preceding cell and s_3 in the

next cell of the input tape. The work tape of the succinct configuration contains the string $x_{w1}x_{w2} \in \Gamma^*$ and the work tape head scans the p_w^{th} cell after the left most cell of the work tape. The left most cell of the work tape is the first cell of the work tape containing the de-limiter $\vdash$ and the symbol of the work tape scanned by M is the first symbol of x_{w2}. Similar to p_i, here $1 \leq p_w \leq k \cdot S(n)$ is a positive integer, where k is a constant, $O(S(n))$ is the space bound on the Turing machine and n is the size of the input.

DEFINITION 1.11. (**succinct initial configuration**) Let Σ be the input alphabet and let $S(n) \in \Omega(\log n)$. Given an input string $x = s_1 s_2 \cdots s_n \in \Sigma^*$ to a Turing machine M, the succinct initial configuration of M is $(q_0, \vdash, 1, s_1 s_2, \vdash, 1, \underbrace{\sqcup \cdots \sqcup}_{O(S(n))})$.

DEFINITION 1.12. (**succinct accepting and succinct rejecting configuration**) Let Σ be the input alphabet, and let Γ be the output alphabet of a Turing machine M. The succinct accepting configuration of M and the succinct rejecting configuration of M are $(q_{acc}, \vdash, 1, s_1 s_2, \vdash, 1, \underbrace{\sqcup \cdots \sqcup}_{O(S(n))})$ and $(q_{rej}, \vdash, 1, s_1 s_2, \vdash, 1, \underbrace{\sqcup \cdots \sqcup}_{O(S(n))})$ respectively, where $s_1, s_2 \in \Sigma$.

As a result of Definition 1.10, we infer the following which is similar to Propositions 1.7 and 1.8.

PROPOSITION 1.13. *Let Σ be the input alphabet, Γ be the output alphabet of a $O(S(n))$-space bounded Turing machine M, where $S(n) \in \Omega(\log n)$. M requires at most $O(S(n))$ space to store its succinct configuration on the work tape and the number of succinct configurations is upper bounded by $O(n \cdot |\Gamma|^{S(n)}) = O(|\Gamma|^{S(n)}) = |\Gamma|^{O(S(n))}$. Clearly the number of mini configurations of M is less that or equal to the number of succinct configurations of M.*

From now onwards, in all chapters to follow, we assume that we uniformly deal either with mini configurations or succinct configurations of $O(\log n)$-space bounded Turing machines, and not both of them. We however call both a mini configuration or a succinct configuration as just a "configuration".

As result of Propositions 1.7, 1.8 and 1.13, for the purpose of results shown in this monograph, irrespective of whether we consider a mini configuration or a succinct configuration, we note the following observation.

THEOREM 1.14. *Let Σ be an input alphabet and let M be a deterministic or a non-deterministic Turing machine that accepts a language $L \subseteq \Sigma^*$. The size of a mini configuration or a succinct configuration of M is $O(\log n)$ and the number of mini configurations or succinct configurations of M is upper bounded by a polynomial n^k on any given input $x \in \Sigma^*$, where $n = |x|$ and $k > 0$ is a constant.*

Once our Turing machine enters its accepting state or rejecting state, it halts and does no further computation. Since the Turing machine halts and does no further computation after it enters the accepting state or the rejecting state, we say that the accepting and rejecting configurations are ***halting configurations***.

1.1.2. Computation tree. Let $S(n) \in \Omega(\log n)$. It is easy to observe that the computation of a $O(S(n))$-space bounded non-deterministic Turing machine M on a given input is in fact a tree which we call the computation tree of M. Since there is an upper bound on the number of configurations of M, the running time of M is at most $|\Gamma|^{O(S(n))}$. Since the running time of M can be taken to be the length of the longest computation path of M, its computation tree is finite and it contains finite number of nodes and edges.

FACT 1.15. *Let $S(n) \in \Omega(\log n)$ and let M be $O(S(n))$-space bounded non-determinsitic Turing machine. Given any input $x \in \Sigma^*$, in the computation of M on input x we assume that no configuration gets repeated. Since otherwise there will exist a path in the computation tree of M on x which extends infinitely long and this contradicts the fact that the computation tree contains finitely many nodes and edges.*

PROPOSITION 1.16. *Let Σ be the input alphabet. Let M be a non-deterministic Turing machine. Given an input $x \in \Sigma^*$, let M be in a configuration c during its computation. The number of configurations to which M can move from c is a constant.*

PROOF. Given a mini configuration c of M, it is easy to note that the number of mini configurations which we can obtain in one step from c based on the transition function of M is upper bounded by $|Q| \cdot 2 \cdot |\Gamma| \cdot 2 \cdot |\Gamma|^2 = 4 \cdot |Q| \cdot |\Gamma|^3$ which is a constant.

Since we define a succinct configuration in Definition 1.10 as an extension of a mini configuration, in one transition on a given input, M can move from the present succinct configuration to at most $4 \cdot |Q| \cdot |\Gamma|^3$ succinct configurations, which is also a constant. □

It is also possible to restrict the possibility of moving from one configuration to more than 2 configurations in a transition, to less than or equal to 2 configurations by modifying the definition of the transition function of M. We can do this by introducing new states in Q, new symbols in the output alphabet Γ and new transitions in δ.

For example, in the Turing machine M, if we have the transition $\delta(q, s, t) = \{(q_1, s, R, t_1, R), (q_2, s, R, t_2, R), (q_3, s, R, t_3, R)\}$ then we introduce new states $p_1, p_1' \in Q$ and a new symbol $\hat{t} \in \Omega$ and modify transitions of M such that,

- $\delta(q, s, t) = \{(q_1, s, R, t_1, R), (p_1, s, R, \hat{t}, R)\}$,
- $\delta(p_1, s', t') = \{(p_1', s', L, t', L)\}$
- $\delta(p_1', s, \hat{t}) = \{(q_2, s, R, t_2, R), (q_3, s, R, t_3, R)\}$

Therefore, treating any configuration as a node u in the computation tree of M such that u has k configurations $v_1, \ldots, v_k$ as children connected by directed edges

(u, v_i), where k is a constant and $1 \leq i \leq k$, it is possible to convert this k-ary tree of depth 1 into a binary tree with u as the root by introducing new intermediate vertices to the k-ary tree such that $v_1, \ldots, v_k$ are the leaf nodes of this binary tree with u as the root. The depth of this binary tree is at least $\log k$ and it is at most k and its size is also a constant.

As a result, it is possible to assume that whenever M makes a non-deterministic choice in its computation, the number of branches in the computation tree is 2. We therefore obtain the following.

PROPOSITION 1.17. *Let M be a non-determinsitic Turing machine. The computation tree of M is a binary tree. It contains finitely many nodes and edges. Every node of the computation binary tree of M denotes a non-deterministic choice of M. Each node is either an internal node which has exactly 2 children or the node is a leaf.*

FACT 1.18. *If M is a non-deterministic Turing machine that requires at most $O(\log n)$ space, where n is the size of the input, then the depth of its computation binary tree of M denotes the running time of M. The length of a computation path from the root to a leaf is upper bounded by the number of configurations of M which is a polynomial in n. The number of nodes in the computation tree of M is exponential in n.*

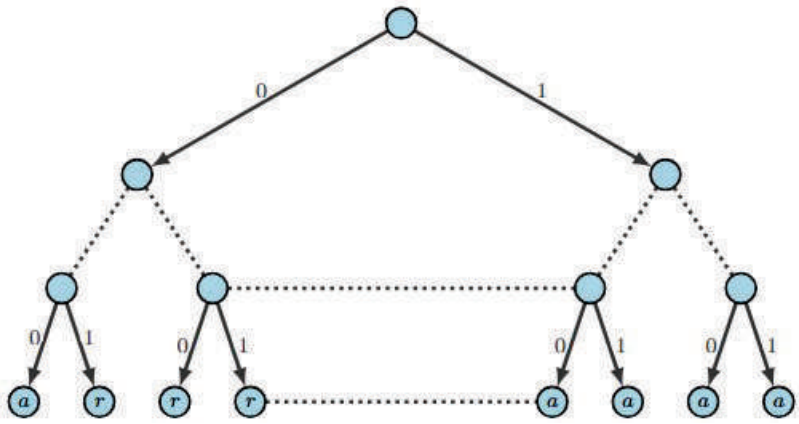

FIGURE 1.2. This figure is the computation tree of a non-deterministic Turing machine M, where 0 and 1 on the edges denote the two non-deterministic choices using which M branches at every stage on its computation path. Leaf nodes are marked as "a" or "r" denoting that the computation path ends in an accepting configuration or in a rejecting configuration respectively. Also all computation paths are of the same length.

In the computation binary tree of a $O(S(n))$-space bounded non-deterministic Turing machine, we usually assume that all computation paths of M on a given input have the same length. That is, the computation proceeds for the same number

of steps along any computation path in the computation tree of a non-deterministic Turing machine, whether it is an accepting computation path or a rejecting computation path. Since in any computation binary tree that we consider, there are no internal nodes which have only one child, if we assume that the length of a directed path from the root to any leaf are equal then we obtain that the tree is in fact a complete binary tree. Note that the converse of this statement is also true: in a complete binary tree the length of a directed path from the root to any leaf is the same. As a result, we usually assume that the computation binary tree of a non-deterministic Turing machine M is a complete binary tree.

The topic of logarithmic space bounded counting complexity classes is dependent on studying the computation binary tree of NL-Turing machines. On a given input $x \in \Sigma^*$, much more than following the computation of a NL-Turing machine M along a computation path, we prove many interesting properties of logarithmic space bounded counting complexity classes by considering an aerial view of the entire computation tree of M on x and designing non-deterministic $O(\log n)$ space algorithms based on observing the number of accepting computation paths and the number of rejecting computation paths of M on x.

1.1.3. Computing functions. Let $g : \Sigma^* \to (\Gamma \setminus \{\sqcup, \vdash, \dashv\})^*$. Given $x \in \Sigma^*$, in order to output $g(x)$, we assume that the Turing machine has a separate two-way semi-infinite write-only output tape which has a separate output tape head that can move in both left and right directions, and which is used by the Turing machine to write $g(x)$ as the output on the output tape. As in the case of the work tape, we assume that the output tape has the de-limiter symbol $\vdash$ in the first cell of the output tape. The transition function δ is defined such that the tape head of the output tape does not move to the left of this first cell of the output cell. The definition of the configuration of any such Turing machine or the notion of its computation tree is the same as a Turing machine that does not have any output tape as described in Sections 1.1.1 and 1.1.2.

Alternately, we may also consider computing $g(x)$ as a language in a symbol-wise manner as follows. We say that $g(x)$ is computable by $O(S(n))$-space bounded Turing machine M, if given an input $x \in \Sigma^*$, we check if the i^{th} symbol of $g(x)$ is $a \in \Gamma \setminus \{\sqcup\}$ by submitting the triple (x, i, a) as an input to M and finding out if M halts in the accepting configuration or in the rejecting configuration in its computation, where $1 \leq i \leq |g(x)|$. Therefore by submitting various input instances (x, i, a) to M for different values of $a \in \Gamma$, it is possible to find out the correct value of the i^{th} symbol of $g(x)$, given $x \in \Sigma^*$. Clearly there is no need for any output tape in these Turing machines.

1.1.4. Oracle Turing machines. If M^A is a $O(S(n)$-space bounded Turing machine that has access to an oracle A then M^A has a separate semi-infinite two-way write-only oracle tape using which it can submit queries to the oracle A. As in the case of the input tape and the work tape of M^A, the oracle tape of M^A is assumed to be semi-infinite, which means that there are only finitely many tape cells to the left and that this tape also has infinitely many cells on the right side. The oracle tape of M^A has a separate two-way tape head, which means that it can

move in both left and right directions. We assume that the oracle tape also has a de-limiter ($\vdash$) in a cell to the left of the first cell to prevent the oracle tape head from moving to the left of the first cell of the oracle tape. Also M^A has three exclusive states q_{QUERY}, q_{YES} and q_{NO}. When M^A completes writing an oracle query string $y \in \Sigma^*$ in the oracle tape of A, it puts the de-limiter ($\dashv$) in the last cell (the $|y|+1$ cell) and enters the q_{QUERY} state. In the next instant the oracle A erases the contents of the oracle tape, brings the oracle tape head to the left most cell of the oracle tape and changes the state of M^A from q_{QUERY} to either q_{YES} or q_{NO} depending on whether $y \in A$ or $y \notin A$ respectively. A configuration of an oracle Turing machine is denoted by the tuple $(q, s_1, p_i, s_2 s_3, t_1, p_w, t_2 t_3, p_{or})$, where $q, s_1, s_2, s_3, t_1, t_2, t_3, p_i, p_w$ are as in the definition of a configuration of a Turing machine. Here, p_{or} is a positive number that indicates the position of the oracle tape head on the oracle tape. It is upper bounded by $|\Gamma|^{O(S(n))}$. Clearly the amount of space required to store a configuration of a $O(S(n))$-space bounded oracle Turing machine is also $O(S(n))$, where $S(n) \in \Omega(\log n)$. Therefore if M^A is a $O(S(n))$ space bounded non-deterministic Turing machine then the number of configurations of M^A is also upper bounded by $|\Gamma|^{O(S(n))}$. From this we infer that the length of any query to the oracle A and the number of queries that M^A can submit to the oracle A are both at most $|\Gamma|^{O(S(n))}$.

1.1.5. Functions as oracles to Turing machines. Let Σ denote the input alphabet and let $\Gamma = \Sigma \cup \{\sqcup, \vdash, \dashv, \#\}$ be the output alphabet. Instead of a language A, if the Turing machine M^f has access to a function $f : \Sigma^* \to \Sigma^*$ as an oracle then M^f first computes an upper bound on the length of any query string that it submits to the oracle f. It also computes an upper bound on the length of the value of the function f. Given an input x of length n, if M^f needs to submit a query to the oracle f, it computes the query string y bit-by-bit and writes it on the oracle tape. M^f actually submits the string $(y\#j\#b\#)$ to the oracle f and enters the q_{QUERY} state, where $1 \leq j \leq m$ denotes the index of the string $f(y)$ that M^f wants to verify if it is b. In the next instant the oracle A erases the contents of the oracle tape, brings the oracle tape head to the left most cell of the oracle tape and changes the state of M^A from q_{QUERY} to either q_{YES} or q_{NO} depending on whether $f(y)_j = b$ or not.

1.1.6. Ruzzo-Simon-Tompa oracle access mechanism. Let us consider a non-deterministic oracle Turing machine M^A with access to a language A as an oracle. We then follow the *Ruzzo-Simon-Tompa oracle access mechanism* in which we assume that M^A always submits its queries to oracle A in a deterministic manner only. This means that all the queries of M^A are submitted to A after doing some computation and even before the first non-deterministic choice has been made in the computation on the given input. All the algorithmic results involving non-deterministic oracle Turing machines in this monograph follow the Ruzzo-Simon-Tompa oracle access mechanism.

1.2. Boolean circuit model of computation

DEFINITION 1.19. We define $\mathcal{B}_0 = \{\neg, (\wedge^2), (\vee^2)\}$ as the standard bounded fan-in basis of Boolean gates.

DEFINITION 1.20. Let $\mathcal{B}_0$ be a standard bounded fan-in basis, and let $s, d : \mathbb{N} \to \mathbb{N}$. We define the complexity class, SIZE-DEPTH$_{\mathcal{B}_0}(s, d)$ as the class of all sets $A \subseteq \{0,1\}^*$ for which there is a circuit family $(C_n)_{n\in\mathbb{N}}$ over the basis $\mathcal{B}_0$ of size $O(s)$ and depth $O(d)$ that accepts A.

DEFINITION 1.21. We define $\mathcal{B}_1 = \{\neg, (\wedge^n)_{n\in\mathbb{N}}, (\vee^n)_{n\in\mathbb{N}}\}$ as the standard unbounded fan-in basis of Boolean gates.

DEFINITION 1.22. Let $\mathcal{B}_1$ be a standard unbounded fan-in basis, and let $s, d : \mathbb{N} \to \mathbb{N}$. We define the complexity class, SIZE-DEPTH$_{\mathcal{B}_1}(s, d)$ as the class of all sets $A \subseteq \{0,1\}^*$ for which there is a circuit family $(C_n)_{n\in\mathbb{N}}$ over the basis $\mathcal{B}_1$ of size $O(s)$ and depth $O(d)$ that accepts A.

DEFINITION 1.23. Let $(C_n)_{n\in\mathbb{N}}$ be a circuit family over a basis $\mathcal{B}$. An *admissible encoding scheme* of the circuit family is defined as follows: First fix an arbitrary numbering of the elements of $\mathcal{B}$. Second, for every n, fix a numbering of the gates of C_n with the following properties:

- in C_n the input gates are numbered $0, \ldots, n-1$,
- if C_n has m output gates then these are numbered $n, \ldots, n+m-1$,
- let $s(n)$ be the size of C_n. There is a polynomial $p(n)$ such that, for every n, the highest number of a gate in C_n is bounded by $p(s(n))$.

The encoding of a gate v in C_n is now given by a tuple $\langle g, b, g_1, \ldots, g_k\rangle$, where g is the gate number assigned to v, b is the number of v in the basis $\mathcal{B}$, and $g_1, \ldots, g_k$ are the numbers of the predecessor gates of v in the order of the edge numbering of C_n. Fix an arbitrary order of the gates of C_n. Let $v_1, v_2 \ldots, v_s$ be the gates of C_n in that order. Let $\overline{v_1}, \overline{v_2}, \ldots, \overline{v_s}$ be their encodings. The admissible encoding of C_n, denoted by $\overline{C_n}$, is defined as $\langle \overline{v_1}, \ldots, \overline{v_s}\rangle$.

We recall the definition of L from Definition 1.4.

DEFINITION 1.24. A circuit family $(C_n)_{n\in\mathbb{N}}$ of size $p(n)$ is L-uniform (also called as *logspace-uniform*) if there is an admissible encoding scheme $\overline{C_n}$ of C_n such that the map $1^n \to \overline{C_n}$ is computable by a deterministic Turing machine using space $O(\log n)$, for all $n \geq 1$, where $p(n)$ is a polynomial in n and n is the size of the input.

DEFINITION 1.25. Let $\mathrm{U_L}$-AC^0 denote L-uniform AC^0. The complexity class $\mathrm{U_L}$-AC^0 = L-uniform SIZE-DEPTH$_{\mathcal{B}_1}(p(n), O(1))$, where $p(n)$ is a polynomial in n, and n is the size of the input.

DEFINITION 1.26. Let $\mathrm{U_L}$-NC^1 denote L-uniform NC^1. The complexity class $\mathrm{U_L}$-NC^1 = L-uniform SIZE-DEPTH$_{\mathcal{B}_0}(p(n), O(\log n))$, where $p(n)$ is a polynomial in n, and n is the size of the input.

As we have it in Definition 1.21, let $\mathcal{B}_1$ denote the standard unbounded fan-in basis of Boolean gates. In this definition, along with the standard Boolean operations, it is easy to include oracle gates that compute a function. An oracle gate that is of special interest to us is the MAJ gate (majority gate) defined as follows:
MAJ
INPUT: n bits $a_{n-1}, \ldots, a_0$.
QUESTION: Are at least half of the a_i one?

DEFINITION 1.27. Let $\mathrm{U_L}$-TC^0 denote L-uniform TC^0. The complexity class $\mathrm{U_L}$-TC^0 = L-uniform SIZE-DEPTH$_{\mathcal{B}_1 \cup \mathrm{MAJ}}(n^{O(1)}, O(1))$.

DEFINITION 1.28. Let $\mathrm{U_L}$-TC^1 denote L-uniform TC^1. The complexity class $\mathrm{U_L}$-TC^1 = L-uniform SIZE-DEPTH$_{\mathcal{B}_1 \cup \mathrm{MAJ}}(n^{O(1)}, \log n)$.

A *Chinese remainder representation* (CRR) of an integer $x \geq 0$ is based on a set $m_1, \ldots, m_n$ of pairwise coprime integers. The set $m_1, \ldots, m_n$ is called the CRR base and each m_i is called a modulus. We will denote this system by CRR(M). Let $M = m_1 \cdots m_n$. By the Chinese Remainder Theorem, every integer $0 \leq x \leq M$ is uniquely represented by its CRR namely $(x_1, \ldots, x_n)$, where $0 \leq x_i < m_i$ and $x_i \equiv x (\mathrm{mod}\ m_i)$.

THEOREM 1.29. *Let CRR(M) denote the CRR system based on the n consecutive primes $3 = m_1 < \cdots < m_n$ and let $M = m_1 \cdots m_n$.*

(1) *Generating and to output the first n primes is in* $\mathrm{U_L}$-NC^1.
(2) *Converting an integer $x < 2^n$ in binary notation to its CRR in CRR(M) is in* $\mathrm{U_L}$-NC^1.
(3) *Converting an integer $x < 2^n$ which is given in its CRR as CRR(M) to its binary notation is in* $\mathrm{U_L}$-NC^1.

COROLLARY 1.30. *Given a 1^n and $O(\log n)$ bit integer $m > 0$ as input, where $n = sizeof(m)$, we can determine if m is a prime in* $\mathrm{U_L}$-NC^1.

We also need the following useful results on the complexity of certain basic operations using Boolean circuits.

THEOREM 1.31.
(1) *Let $\Sigma = \{0, 1\}$ be the input alphabet. Given a string $x \in \Sigma^*$ in the unary notation as the input, the problem of computing the length of x and to output it in binary notation is in* $\mathrm{U_L}$-NC^1.
(2) *Let $\Sigma = \{0, 1\}$ be the input alphabet. Given a string $x \in \Sigma^*$, the problem of outputting the reverse of the input x is in* $\mathrm{U_L}$-NC^1.
(3) *Let $\Sigma = \{0, 1\}$ be the input alphabet. Let x, y be integers given in binary notation as input. Here we assume that if $x = x_0 x_1 \cdots x_{n-1}$ is the binary representation of x, then x_0 is the least significant bit (LSB) and x_{n-1} is the most significant bit (MSB) of x. The MSB denotes the sign of x, where 0 denotes that x is negative and 1 denotes that x is positive. Computing the sum of x and y in binary notation, and the product of x and y in binary notation is in* $\mathrm{U_L}$-NC^1.
(4) *Let $\Sigma = \{0, 1\}$ be the input alphabet. Let x and y be positive integers. Determining if $x < y$ is in* $\mathrm{U_L}$-NC^1.

(5) *Let* $\Sigma = \{0, 1\}$ *be the input alphabet and let* $x, y \in \Sigma^*$ *be integers given as input such that* $y \neq 0$*. Computing the quotient of* $\lfloor x/y \rfloor$ *is in* $\mathrm{U_L}$-NC^1.
(6) *Let* $\Sigma = \{0, 1\}$ *be the input alphabet and let* $x, y \in \Sigma^*$ *be integers given as input such that* $y > 0$*. Computing* $x (mod\ y)$ *is in* $\mathrm{U_L}$-NC^1.

1.2.1. Logarithmic space bounded computation. A standard result in computational complexity is that $\mathrm{U_L}$-$\mathrm{NC}^1 \subseteq \mathrm{L}$ which relates the Boolean circuit model of computation with the Turing machine model. As a result all basic operations that we have listed above in Theorems 1.29 and 1.31 are computable in $O(\log n)$ space using deterministic Turing machines, where n is the size of the input. We therefore get the following observation.

PROPOSITION 1.32. *Let* n *be a positive integer given as input in the unary notation. It is possible to non-deterministically choose an integer* $1 \leq k \leq n$ *and also output* k *in binary notation using* $O(\log n)$*-space.*

PROOF. Given a positive integer n in the unary notation 1^n as input, it follows from Theorem 1.31 that we can compute n in binary notation using space $O(\log n)$ since the length of 1^n is computable in $\mathrm{U_L}$-NC^1 which is contained in L.

Initially let x denote the empty string. Iteratively, we non-deterministically choose 0 or 1 and append it to the string x and check in every iteration if $x \leq n$. Since there exists some iteration in which we are bound to get $x > n$ then we output the string x obtained in the previous iteration as the integer k and stop. Clearly this requires at most $O(\log n)$ space. □

DEFINITION 1.33. Let Σ be the input alphabet and $L_1, L_2 \in \Sigma^*$. We say that L_1 is $\mathrm{U_L}$-AC^0 many-one reducible to L_2, denoted by $L_1 \leq_m^{\mathrm{U_L}-\mathrm{AC}^0} L_2$, if there exists a function $f : \Sigma^* \to \Sigma^*$ such that given any input string $x \in \Sigma^*$, it is possible to compute each symbol of $f(x)$ in $\mathrm{U_L}$-AC^0 and we have $f(x) \in L_2$ if and only if $x \in L_1$. The function f is called as $\mathrm{U_L}$-AC^0 many-one reduction from L_1 to L_2.

We once again recall the definition of L from Definition 1.4.

DEFINITION 1.34. Let Σ be the input alphabet. We define the complexity class FL to be the class of all functions $f : \Sigma^* \to \Sigma^*$ such that given an input $x \in \Sigma^*$, each symbol of $f(x)$ is computable by an $O(\log n)$-space bounded deterministic Turing machine, where $n = |x|$.

We observe that the composition of two functions in FL is a function in FL.

PROPOSITION 1.35. *Let* Σ *be the input alphabet and let* $f_1, f_2 : \Sigma^* \to \Sigma^*$. *For any* $x \in \Sigma^*$*, we have* $f_2(f_1(x)) \in$ FL. *In other words,* FL $\circ$ FL $=$ FL.

PROOF. Let M_i be the $O(\log n)$-space bounded deterministic Turing machine that computes $f_i(x)$ on any input string $x \in \Sigma^*$, where $i = 1, 2$, and $n = |x|$. Given an input string $x \in \Sigma^*$, let M be a deterministic Turing machine that first simulates M_1 on input x and hence computes $f_1(x)$ in a symbol-wise manner such that it stores at most $O(\log n)$ symbols of $f_1(x)$ at any instant on the work tape,

where $n = |x|$. The Turing machine M simulates M_2 on the input $f_1(x)$ after computing the required $O(\log n)$ symbols of $f_1(x)$. If M requires any symbol of $f_1(x)$ for its simulation of M_2 which is not stored in the work tape then it stores its present configuration on the work tape and starts the simulation of M_1 on input x to compute the required symbol. Computation of M on the input x proceeds in this manner and we finally obtain $f_2(f_1(x))$ in the work tape. Since this entire computation requires $O(\log n)$ space and M is deterministic we can compute $f_2(f_1(x))$ in FL. □

DEFINITION 1.36. Let Σ be the input alphabet and $L_1, L_2 \in \Sigma^*$. We say that L_1 is logspace many-one reducible to L_2, denoted by $L_1 \leq_m^{\mathrm{L}} L_2$, if there exists a function $f : \Sigma^* \to \Sigma^*$, which is computable by a deterministic $O(\log n)$ space bounded deterministic Turing machine, such that given any input string $x \in \Sigma^*$, we have $f(x) \in L_2$ if and only if $x \in L_1$, where $n = |x|$. The function f is called as logspace many-one reduction from L_1 to L_2.

NOTE 1. (1) *We note that in the above definition, if $L_1 \leq_m^{\mathrm{L}} L_2$ using a logspace computable function f then given any input $x \in \Sigma^*$ we have $x \in L_1$ if and only if $f(x) \in L_2$. Therefore $x \in \overline{L_1}$ if and only if $f(x) \in \overline{L_2}$. In other words, $\overline{L_1} \leq_m^{\mathrm{L}} \overline{L_2}$.*

(2) *Also it is easy to show that if L_1 is logspace many-one reducible to L_2 and L_2 is in a complexity class $\mathcal{C}$ where $\mathcal{C}$ contains the complexity class* L *of all languages that can be decided in deterministic logarithmic space then L_1 is in $\mathcal{C}$.*

(3) *If $L_1 \leq_m^{\mathrm{L}} L_2$ and $L_2 \leq_m^{\mathrm{L}} L_3$ then $L_1 \leq_m^{\mathrm{L}} L_3$.*

DEFINITION 1.37. Let Σ be the input alphabet and let $\mathcal{C}$ be a complexity class. $L \subseteq \Sigma^*$ is logspace many-one hard for $\mathcal{C}$ if for any $L' \in \mathcal{C}$, we have $L' \leq_m^{\mathrm{L}} L$.

DEFINITION 1.38. Let Σ be the input alphabet and let $\mathcal{C}$ be a complexity class. $L \subseteq \Sigma^*$ is logspace many-one complete for $\mathcal{C}$ if $L \in \mathcal{C}$ and L is logspace many-one hard for $\mathcal{C}$.

DEFINITION 1.39. Let Σ be the input alphabet and $L_1, L_2 \in \Sigma^*$. We say that L_1 is logspace Turing reducible to L_2, denoted by $L_1 \leq_{\mathrm{T}}^{\mathrm{L}} L_2$, if there exists a $O(\log n)$ space bounded Turing machine M^{L_2} that has L_2 as an oracle such that given any input string $x \in \Sigma^*$, we have $M^{L_2}(x)$ accepts if and only if $x \in L_1$, where $n = |x|$.

DEFINITION 1.40. Let Σ be the input alphabet and let $\mathcal{C}$ be a complexity class. We define $\mathrm{L}^{\mathcal{C}}$ as the class of all languages $L \subseteq \Sigma^*$ that is accepted by a $O(\log n)$ space bounded deterministic Turing machine that has oracle access to a language $A \in \mathcal{C}$, where n is the size of the input and $A \subseteq \Sigma^*$.

It is easy to note that if L_1 is logspace Turing reducible to L_2 and L_2 is in a complexity class $\mathcal{C}$, where $\mathcal{C}$ contains the complexity class L of all languages that can be decided in deterministic logarithmic space then L_1 is in $\mathrm{L}^{\mathcal{C}}$.

DEFINITION 1.41. Let Σ be the input alphabet. Given functions $f, g : \Sigma^* \to \mathbb{Z}^+$, we say that f is logspace many-one reducible to g, denoted by $f \leq_m^{\mathrm{L}} g$, if there exists a function $h : \Sigma^* \to \Sigma^*$ computable in $O(\log n)$ space, where n is the length of the input, such that $f(x) = g(h(x))$ for every $x \in \Sigma^*$.

Notes

The Turing machine model of computation explained in Section 1.1 is from the excellent textbook by Dexter Kozen [**Koz97**, Lectures 28-29] and it is as introduced in any standard textbook on the Theory of Computation such as [**HU79, LP98, Sip13**]. The Ruzzo-Simon-Tompa oracle access mechanism for non-deterministic oracle Turing machines stated in Section 1.1.6 has been conceptualized and formulated by Walter L. Ruzzo, Janos Simon and Martin Tompa in [**RST84**].

Background material including definitions and results needed to understand the results of Section 1.2 is from [**Vol99**, Chapter 1] and [**Weg87**]. Definitions 1.19 to 1.24 are due to Heribert Vollmer and they are from [**Vol99**, Chapter 1, pp. 10 & Chapter 2, pp. 43-44, 47]. In [**Vol99**, Lemma 2.25] it is shown that in order to define a uniform family of circuits, the concept of admissible encoding scheme and the concept of "*direct connection language*" are equivalent in terms of complexity. In particular it is stated in [**Vol99**, pp. 51] that if the uniform circuit family $\{C_n\}_{n\geq 1}$ is of size polynomial in the size of the input then, $\{C_n\}_{n\geq 1}$ is logspace-uniform if and only if the admissible encoding scheme is computable from 1^n in space $O(\log n)$ if and only if the direct connection language is in L. We need the MAJ function as an oracle gate along with the standard unbounded fan-in basis of Boolean gates in Boolean circuits to define $\mathrm{U_L}\text{-}\mathrm{TC}^0$ in Definition 1.27 and $\mathrm{U_L}\text{-}\mathrm{TC}^1$ in Definition 1.28.

The concept of Chinese remainder representation and Theorem 1.29, due to A. Chiu, G. Davida and B. Litow [**CDL01**], are based on the Chinese Remainder Theorem. Next, Theorem 1.31 is based on results shown in [**BCH86, CDL01, HAB02**]. In fact, the first log-depth, polynomial size uniform circuit family for computing the quotient upon dividing an integer by a non-zero integer is demonstrated in the seminal paper of Paul Beame, Stephen A. Cook and James Hoover [**BCH86**]. This complexity upper bound is been improved to DLOGTIME-uniform TC^0 by William Hesse, Eric Allender and David A. Mix Barrington in [**HAB02**] and their proof uses descriptive complexity. The well known result that $\mathrm{U_L}\text{-}\mathrm{NC}^1 \subseteq \mathrm{L}$ is due to Alan Borodin [**Bor77**].

We refer to [**Koz97**, Lectures 29-33] for the notion of simulating one Turing machine by another and for the definition of a reduction.

CHAPTER 2

Counting in Non-deterministic Logarithmic Space

2.1. Non-deterministic Logarithmic Space: NL

DEFINITION 2.1. Let Σ be the input alphabet. We define the complexity class Non-deterministic Logarithmic Space, $\mathrm{NL} = \{L \subseteq \Sigma^* |$ there exists a $O(\log n)$ space bounded non-deterministic Turing machine M such that $L = L(M)\}$.

DEFINITION 2.2. Let Σ be the input alphabet. For a language $L \subseteq \Sigma^*$, the complement of L is $\overline{L} = \Sigma^* - L$. Given a complexity class $\mathcal{C}$, we define the complement of $\mathcal{C}$ as co-$\mathcal{C} = \{\overline{L} \subseteq \Sigma^* | L \in \mathcal{C}\}$.

DEFINITION 2.3. Let Σ be the input alphabet. For a language $L \in \Sigma^*$, the complement of the language L is $\overline{L} = \Sigma^* - L$. We define the complexity class co-Non-deterministic Logarithmic Space, co-NL $= \{\overline{L} \subseteq \Sigma^* |$ there exists a $O(\log n)$ space bounded non-deterministic Turing machine M such that $L = L(M)\}$.

2.1.1. NL and the Directed st-Connectivity Problem (DSTCON).

DEFINITION 2.4. Let $G = (V, E)$ be a directed graph given in terms of its adjacency matrix, and let $s, t \in V$. We define $\mathrm{DSTCON} = \{(G, s, t) | \exists$ a directed path from s to t in $G\}$.

In this chapter, any matrix $A \in \mathbb{R}^{n \times n}$ is synonymous with the adjacency matrix of a directed graph G such that the weight of the directed edge (i, j) in G is equal to the entry $A(i, j)$ of the matrix A, where $1 \leq i, j \leq n$. If $A(i, j) = 0$ then the directed edge (i, j) does not exist in G.

LEMMA 2.5. DSTCON *is logspace many-one hard for* NL.

PROOF. Let Σ be the input alphabet and let $L \subseteq \Sigma^*$. Assume that $L \in \mathrm{NL}$ and let M be a non-deterministic Turing machine that accepts L using space at most $O(\log n)$ where n is the size of the input. We show that $L \leq_m^{\mathrm{L}} \mathrm{DSTCON}$. Given an input string $x \in \Sigma^*$, we consider the set of all configurations of M on an input of length $n = |x|$. It follows from Theorem 1.14 that the number of configurations of M on input of length n is at most n^k, for some constant $k > 0$. We now define a directed graph $G = (V, E)$ where the set of all vertices V is the set of all configurations of M and there exists an edge from a vertex u to another vertex v in G if the tape head upon reading the symbol it points to in configuration u results in the configuration v of M. Since $|V| \leq n^k$ and M is $O(\log n)$ space bounded, any configuration of M is also $O(\log n)$ space bounded, and so given a

configuration u of M on x it is always possible to find the neighbors of u in G in $O(\log n)$ space. Also given $x \in \Sigma^*$ we have $x \in L$ if and only if there exists a directed path from the vertex which is the initial configuration of M on input x to the vertex which is the accepting configuration of M on input x. Clearly, given the description of M and an input $x \in \Sigma^*$, we can output the adjacency matrix of the directed graph G and vertices s and t, which denote the initial and final configurations of M on x respectively, using $O(\log n)$ space, where $n = |x|$. This shows that $L \leq_m^{\mathrm{L}}$ DSTCON which implies that DSTCON is hard for NL under $\leq_m^{\mathrm{L}}$ reductions. □

LEMMA 2.6. DSTCON $\in$ NL.

PROOF. Let $G = (V, E)$ be a directed graph which is represented by its adjacency matrix and let $s, t \in V$ be given as input along with G. Let us consider the following algorithm which can be implemented by a non-deterministic Turing machine.

Algorithm 1 Directed-Graph-Accessibility
Input: (G, s, t), where $G = (V, E)$ is a directed graph, and $s, t \in V$.
Output: *accept* if $\exists$ a directed path from s to t in G. Otherwise *reject.*
Complexity: NL.

```
1: Let n ← |V|.
2: Let m ← |E|.
3: Let u ← s.
4: for counter ← 0 to (n − 1) do
5:     Non-deterministically choose a vertex v ∈ V such that (u, v) ∈ E.
6:     if ∄v ∈ V such that (u, v) ∈ E then reject the input.
7:     end if
8:     if v = t then
9:         output there exists a directed path from s to t in G.
10:        accept the input.
11:    end if
12:    u ← v.
13:    counter ← counter + 1.
14: end for
15: reject the input and stop.
```

The non-deterministic Turing machine that implements the Directed-Graph-Accessibility (G, s, t) algorithm needs at most $O(\log n)$ space to store vertices $u, v \in V$, to store variables n, m and to update the variable $counter$. In addition it looks into the adjacency matrix of the directed graph given as input to find and non-deterministically choose vertex v which is the head of the directed edge (u, v) in each iteration. Clearly this also needs at most $O(\log n)$ space.

Now, if there exists a directed path from s to t then any such path is of length less than or equal to $(n-1)$ and it is possible to non-deterministically guess the vertices along a directed path from s to t. Thus, whenever there exists a directed path

there exists a computation path of the computation tree of this non-deterministic Turing machine that accepts. Conversely, if there does not exist any path from s to t in G then no computation path of this non-deterministic Turing machine ends in an accepting state. Therefore DSTCON $\in$ NL. □

THEOREM 2.7. DSTCON *is logspace many-one complete for* NL.

PROOF. The result follows from Lemma 2.6 and Lemma 2.5. □

DEFINITION 2.8. We define SLDAG $= \{G|G = (V, E)$ is a simple layered directed acyclic graph which is represented by its adjacency matrix, and which does not have any self-loops on vertices or directed cycles or parallel edges between vertices. Also vertices in G are arranged as a square matrix such that there are n rows and every row has n vertices. Any edge in this graph is from a vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2\}$.

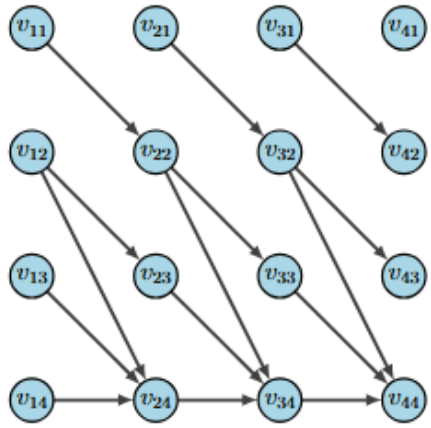

FIGURE 2.1. *This figure is a directed graph G which is an instance of* SLDAG. *Assuming that vertices $s = v_{11}$ and $t = v_{44}$ we also infer that this instance (G, s, t) is in* SLDAGSTCON.

DEFINITION 2.9. Let $G = (V, E) \in$ SLDAG be represented by its adjacency matrix, and let $s, t \in V$ such that s is a vertex in the first row and t is a vertex in the last row of G. We define SLDAGSTCON $= \{(G, s, t)|\exists$ a directed path from s to t in $G\}$.

THEOREM 2.10. SLDAGSTCON *is logspace many-one complete for* NL.

PROOF. We know from Theorem 2.7 that the st-connectivity problem for directed graphs is complete for NL under logspace many-one reductions. Therefore the st-connectivity problem for directed graphs which does not have any self-loops on vertices or parallel edges between vertices is also complete for NL under logspace many-one reductions. In other words, the st-connectivity problem for simple directed graphs is complete for NL under logspace many-one reductions.

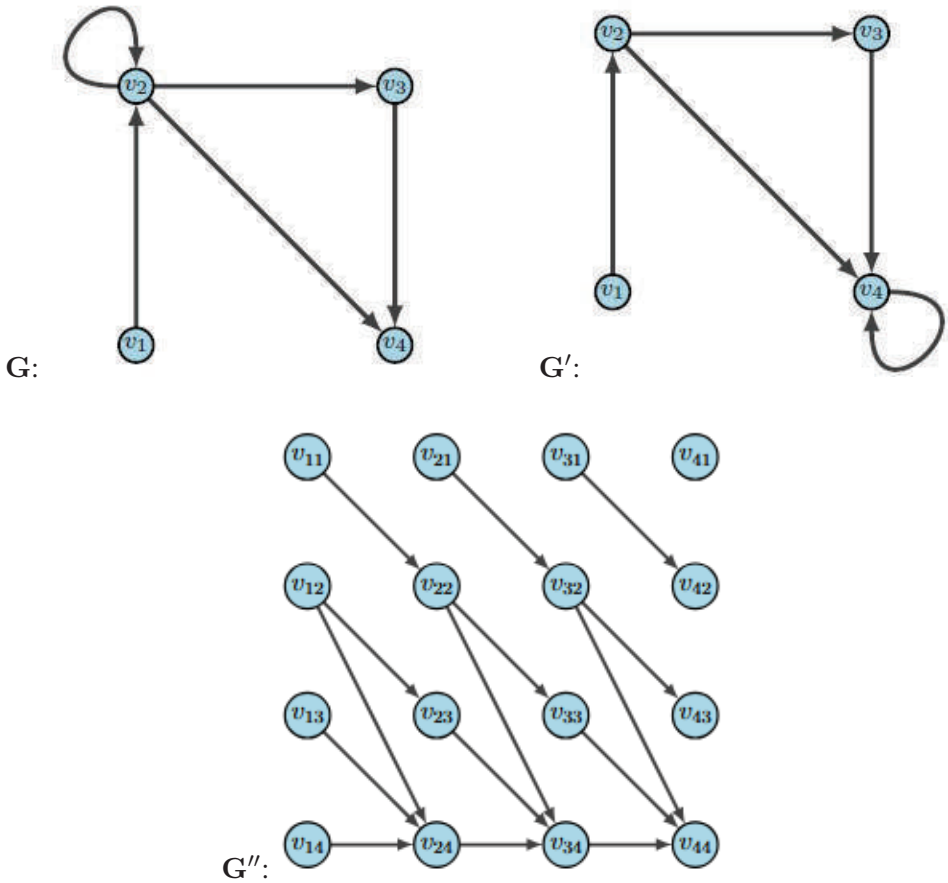

FIGURE 2.2. *This figure explains the reduction from a directed graph G with vertices $s_1 = v_1$ to the vertex $t_1 = v_4$ to a "yes" instance G'' of* SLDAGSTCON *where $s = v_{11}$ and $t = v_{44}$.*

Let $G = (V, E)$ be a simple directed graph given in terms of its adjacency matrix, and let $s, t \in V$. Also let $n = |V|$. Given G, we obtain the directed graph $G' = (V', E')$, where $V' = V$ and $E' = E \cup (t, t)$. We now reduce G' to $G'' = (V'', E'') \in$ SLDAG. In G'', the vertex set V'' has n^2 vertices arranged as a $n \times n$ matrix obtained by creating n copies of V'. As an example refer to Figure 2.2 in which we obtain an instance of SLDAG from the directed graph given as input. For the sake of convenience, we assume in this example that vertices in each row of the instance of SLDAG are arranged vertically.

Now let the vertex s in the first row be denoted by $(1, s)$ and the vertex t in the last row be denoted by (n, t). A (i, j) vertex in G'' is the j^{th} vertex in the i^{th} row of G'', where $1 \leq i, j \leq n$. Here $E'' = \{((i, j_1), (i + 1, j_2))|(j_1, j_2) \in E', \text{ where } 1 \leq i \leq n - 1 \text{ and } 1 \leq j_1, j_2 \leq n\}$. Now it is easy to note that there exists a directed path from s to t in G if and only if there exists a directed path from $(1, s)$ to (n, t) in G''. Since we can obtain the adjacency matrix of G''

from the adjacency matrix of G using a logspace many-one reduction, the result follows. □

DEFINITION 2.11. Let Σ be the input alphabet. The complexity class $\sharp$L is defined to be the class of functions $f : \Sigma^* \to \mathbb{Z}^+$ such that there exists a NL-Turing machine M for which we have $f(x) = acc_M(x)$ where $acc_M(x)$ denotes the number of accepting computation paths of M on input $x \in \Sigma^*$.

REMARK 2.12. If a language $L \subseteq \Sigma^*$ is in NL then there exists a NL-Turing machine M that accepts L. Therefore, given any input $x \in \Sigma^*$ we have $x \in L$ if and only if there exists a $y \in \Sigma^*$ which is of length at most $p(n)$ such that $M(x, y)$ *"accepts"*, where $p(n)$ is a polynomial in $n = |x|$. As a result, the problem of counting the number of witnesses y such that $M(x, y)$ *"accepts"* is in $\sharp$L.

We recall the notion of reduction between functions stated in Definition 1.41.

DEFINITION 2.13. A function $f : \Sigma^* \to \mathbb{Z}^+$ is logspace many-one hard for $\sharp$L if every function $g \in \sharp$L is logspace many-one reducible to f.

DEFINITION 2.14. A function $f : \Sigma^* \to \mathbb{Z}^+$ is logspace many-one complete for $\sharp$L if $f \in \sharp$L and f is logspace many-one hard for $\sharp$L.

PROPOSITION 2.15. *Let (G, s, t) be an input instance of* DSTCON. *Counting the number of directed paths from s to t in G, denoted by* $\sharp$DSTCON*, is logspace many-one complete for* $\sharp$L.

PROPOSITION 2.16. *Let (G, s, t) be an input instance of* SLDAGSTCON. *Counting the number of directed paths from s to t in G, denoted by* $\sharp$SLDAGSTCON*, is logspace many-one complete for* $\sharp$L.

PROPOSITION 2.17. *Let Σ be the input alphabet and $f \in \sharp$L using a* NL*-Turing machine M such that $f(x) = acc_M(x)$ on any input $x \in \Sigma^*$.*

1. *For any input string $x \in \Sigma^*$, $g(x) = (G, s, t)$, where g is the canonical logspace many-one reduction used to show that* DSTCON *is* NL*-hard and (G, s, t) is an input instance of* DSTCON*, we have $f(x)$ is equal to the number of directed paths from s to t in G,*
2. *For any input string $x \in \Sigma^*$, $g(x) = (G, s, t)$, where g is the canonical logspace many-one reduction used to show that* SLDAGSTCON *is* NL*-hard and (G, s, t) is an input instance of* SLDAGSTCON*, we have $f(x)$ is equal to the number of directed paths from s to t in G*

2.2. The Immerman-Szelepcsenyi Theorem

2.2.1. The Immerman-Szelepcsenyi Theorem in logarithmic space.

THEOREM 2.18. NL = co-NL

PROOF. Let $G = (V, E)$ be an input instance of SLDAG which is given in terms of its adjacency matrix and let $s, t \in V$ such that s is a vertex in the first row and t is a vertex in the last row of G. We know from Theorem 2.10 that deciding if $(G, s, t) \in$ SLDAGSTCON is logspace many-one complete for NL. Therefore,

to prove this theorem, it suffices to prove that the problem of deciding if there does not exist any path from s to t in G is also in NL. In other words we show that $\overline{\text{SLDAGSTCON}} \in \text{NL}$.

Algorithm 2 Immerman-Szelepcsenyi-in-Logspace.
Input: (G, s, t), where $G = (V, E)$ is an input instance of SLDAG and $s, t \in V$.
Output: *accept* if $\nexists$ a directed path from s to t in G. Otherwise *reject.*
Complexity: NL.

1: Let $\alpha \leftarrow 1$.
2: **for** $i \leftarrow 1$ to $(n-1)$ **do**
3: Let $\beta \leftarrow 1$.
4: Let $d \leftarrow 0$.
5: $flag \leftarrow 0$.
6: **for** each node v in layer $(i+1)$ in G **do**
7: **for** each node u in layer i in G **do**
8: Non-deterministically either perform or skip {step 9 to step 16}.
9: Non-deterministically follow a path of length $\leq (i-1)$ from s.
10: **if** this path does not end at u **then**
11: reject and stop.
12: **end if**
13: $d \leftarrow d + 1$.
14: **if** (u, v) is an edge of G and $flag = 0$ **then**
15: $\beta \leftarrow \beta + 1$ and $flag \leftarrow 1$.
16: **end if**
17: **end for**
18: **if** $d \neq \alpha$ **then**
19: reject and stop.
20: **end if**
21: $flag \leftarrow 0$.
22: **end for**
23: $\alpha \leftarrow \beta$.
24: **end for**
25: Let $d \leftarrow 0$.
26: **for** each node u in layer n in G **do**
27: Non-deterministically either perform or skip {step 27 to step 34}.
28: Non-deterministically follow a path of length $(n-1)$ from s.
29: **if** this path does not end at u **then**
30: reject and stop.
31: **end if**
32: **if** $u = t$ **then**
33: reject and stop.
34: **end if**
35: $d \leftarrow d + 1$.
36: **end for**

Algorithm 2 Immerman-Szelepcsenyi-in-Logspace (continued)

37: **if** $d \neq \alpha$ **then**
38: reject and stop.
39: **else**
40: accept and stop.
41: **end if**

Let us consider our algorithm Immerman-Szelepcsenyi-in-Logspace(G, s, t). We show that this algorithm can be implemented by a NL-Turing machine M. As a result M must have at least one accepting computation path if and only if there does not exist any directed path from s to t in G.

Let us assume that we know α which is the number of nodes in layer n of G that are reachable from s in G. We assume that α is provided as an input to M and we show in steps 24 to 40 of the above algorithm Immerman-Szelepcsenyi-in-Logspace(G, s, t), that it is possible to use α to prove that $\overline{\text{SLDAGSTCON}} \in \text{NL}$ using a method which we call as ***non-deterministic counting***. Later we prove that M also computes α in steps 1 to 23 using the same non-deterministic counting method.

In steps 24 to 40, given G, s, t and α, the Turing machine M operates as follows. One by one M goes through all the n nodes in layer n of G and non-deterministically guesses whether each one is reachable from s. Whenever a node u is guessed to be reachable, M attempts to verify this guess by guessing a path of length n or less from s to u. If this computation path fails to verify this guess, it rejects. The Turing machine M counts the number of nodes that have been verified to be reachable. When a path has gone through all of G's nodes in layer n, it checks that the number of nodes that are verified to be reachable from s equals α, the number of nodes that are actually reachable from s, and rejects if not. In other words, if M non-deterministically chooses exactly α nodes reachable from s, not including t, and proves that each is reachable from s by guessing the path, M knows that the remaining nodes including t are not reachable and therefore it can accept.

Next we show in steps 1 to 23 described in the above algorithm Immerman-Szelepcsenyi-in-Logspace(G, s, t), how to calculate α: the number of nodes that are reachable from s in layer n in G. We describe a non-deterministic logspace procedure whereby at least one computation path has the correct value of α in which it accepts and all the other paths reject.

Let $A_0 = \{s\}$. For each i from 1 to $(n-1)$, we define A_i to be the collection of all nodes in layer $(i+1)$ that are reachable from s by a directed path of length i. So A_{n-1} is the set of all nodes in layer n that are reachable from s by a directed path of length $(n-1)$. Let $\beta = |A_{i-1}|$. In steps 6 to 22 of the Immerman-Szelepcsenyi-in-Logspace(G, s, t) algorithm we show how to calculate $|A_i|$ from $|A_{i-1}|$. Repeated application of this procedure yields the desired value of α.

We calculate $|A_i|$ from $|A_{i-1}|$ using an idea similar to the one presented earlier in this proof. In the algorithm we go through all the nodes of G in layer $(i+1)$ and determine whether each is a member of A_i and also count members to find $|A_i|$.

To determine whether a node v in layer $(i+1)$ is in A_i, we use the innermost **for** loop starting in line 7, to go through all the nodes of G in layer i and guess whether each node is in A_{i-1}. Each positive guess is verified by a path of length $(i-1)$ from s. For each node u verified to be in A_{i-1}, the algorithm tests whether (u, v) is an edge of G. If it is an edge, v is in A_i. The variable β is used to count the number of vertices which are in A_i. We use the $flag$ variable to prevent the vertex v from being counted more than once. Additionally the number of nodes verified to be in A_{i-1} is counted. At the completion of the innermost **for** loop, if the total number of nodes verified to be in A_{i-1} is not α then entire A_{i-1} has not been found, so this computation path rejects. If the count equals α and v has not been shown to be in A_i, we conclude that v is not in A_i. Then we go to the next vertex v in layer $(i+1)$ in the outer loop. Since we use constant number of variables each of which takes values from 0 to n in the this algorithm, it is easy to note that the Turing machine M that implements it is a NL-Turing machine. As a result we have shown that $\overline{\text{SLDAGSTCON}} \in \text{NL}$ which implies co-NL $\subseteq$ NL from which the result follows. □

We recall the notions of logspace Turing reducibility from Definition 1.39 and the definition of $\text{L}^{\mathcal{C}}$ from Definition 1.40, where $\mathcal{C}$ is a complexity class.

THEOREM 2.19. $\text{L}^{\text{NL}} = \text{NL}$.

PROOF. It is trivial to note that $\text{NL} \subseteq \text{L}^{\text{NL}}$. To prove our result we have to show that $\text{L}^{\text{NL}} \subseteq \text{NL}$. Let $L' \in \text{L}^{\text{NL}}$. Therefore there exists a $O(\log n)$-space bounded deterministic Turing machine M^A, that has access to a language $A \in \text{NL}$ as an oracle such that given any input string $x \in \Sigma^*$, M^A correctly decides if $x \in L'$. Since M requires at most $O(\log n)$ space on any input of size n, we infer that the number of queries that M submits to the oracle A is a polynomial in the size of the input. Let M_A be the NL-Turing machine that decides if an input string is in A or not. Due to Theorem 2.18 it follows that $\overline{A}$ is also in NL. Let $M_{\overline{A}}$ be the NL-Turing machine that correctly decides if any input string is in $\overline{A}$. Now let Σ be the input alphabet and let $x \in \Sigma^*$ be the input string. Let us consider the following algorithm implemented by a non-deterministic Turing machine N.

Algorithm 3 Closure-Logspace-Turing-for-NL.
Input: $x \in \Sigma^*$.
Output: *accepts* if $x \in L$, where $L \in \text{L}^{\text{NL}}$. Otherwise *reject.*
Complexity: NL.

```
1: while N has not reached any of its halting configurations do
2:     N simulates M on input x until a query y is generated.
3:     N simulates M_A on input y.
4:     if M_A accepts y then
5:         N assumes that the reply of the oracle to the query y is "YES".
```

Algorithm 3 Closure-Logspace-Turing-for-NL (continued)

```
    else
        N simulates M_{\overline{A}} on y
        if M_{\overline{A}} accepts y then
            N assumes that the reply of the oracle to the query y is "NO".
        else
            N rejects the input x and stops.
        end if
    end if
    N continues to simulate M on x.
end while
```

We now show that N correctly decides if $x \in L$. It is easy to note that if for any oracle query string y, either M_A or $M_{\overline{A}}$ can accept y and not both of them can accept y. Also if M_A accepts y then $y \in A$. Similarly if $M_{\overline{A}}$ accepts y then $y \in \overline{A}$. As a result when either of these two NL-Turing machines accept then this can be taken to be the output of the oracle and N will proceed with its simulation of M. On the contrary if neither M_A nor $M_{\overline{A}}$ accept since it follows from Theorem 2.18 that NL is closed under complement, we know that there exists at least one accepting computation path in exactly one of these Turing machines and we did not simluate both the Turing machines along any such path. Therefore N also rejects x and stops the computation. Since N simulates only NL-Turing machines, it shows that N is also a NL-Turing machine. This shows that $\mathrm{L}^{\mathrm{NL}} \subseteq \mathrm{NL}$ from which we get that $\mathrm{L}^{\mathrm{NL}} = \mathrm{NL}$. □

COROLLARY 2.20. NL *is closed under union and intersection.*

DEFINITION 2.21. Let Σ be the input alphabet and let $\mathcal{C}$ be a complexity class. We define $\mathrm{NL}^{\mathcal{C}}$ to be the class of all languages $L \subseteq \Sigma^*$ that is accepted by a $O(\log n)$ space bounded non-deterministic Turing machine M that has oracle access to a language $A \in \mathcal{C}$, where n is the size of the input and $A \subseteq \Sigma^*$. Here we assume that M submits queries to the oracle A according to the Ruzzo-Simon-Tompa oracle access mechanism.

THEOREM 2.22. $\mathrm{NL}^{\mathrm{NL}} = \mathrm{NL}$.

PROOF. It is trivial to note that $\mathrm{NL} \subseteq \mathrm{NL}^{\mathrm{NL}}$. Therefore we have to prove that $\mathrm{NL}^{\mathrm{NL}} \subseteq \mathrm{NL}$. Let $L \in \mathrm{NL}^{\mathrm{NL}}$. Therefore there exists a NL-Turing machine M^A that has access to a language $A \in \mathrm{NL}$ as an oracle such that given any input string $x \in \Sigma^*$, M^A decides if $x \in L$ correctly. It follows from Section 1.1.6 that our NL-Turing machine M^A follows the Ruzzo-Simon-Tompa oracle access mechanism to submit queries to the oracle A. Due to Proposition 1.14, since the number of configurations of a NL-Turing machine is at most a polynomial in the size of the input, we that the number of queries that are submitted to the oracle A is also a polynomial in the size of the input. Let M_A be the NL-Turing machine that decides if an input string is in A or not. Since we have shown in Theorem 2.18 that

NL is closed under complement, it follows that $\overline{A}$ is also in NL. Let $M_{\overline{A}}$ be the NL-Turing machine that correctly decides if any input string is in $\overline{A}$. Now let Σ be the input alphabet and let $x \in \Sigma^*$ be the input string. Let us once again consider the Algorithm 3, Closure-Logspace-Turing-for-NL, in Theorem 2.19 implemented by a non-deterministic Turing machine N.

We now show that N correctly decides if $x \in L$. It is easy to note that if for any oracle query string y, either M_A or $M_{\overline{A}}$ can accept y. Also if M_A accepts y then $y \in A$. Similarly if $M_{\overline{A}}$ accepts y then $y \in \overline{A}$. As a result when either of these two NL-Turing machines accept then this can be taken to be the output of the oracle and N will proceed with its simulation of M. On the contrary if neither M_A nor $M_{\overline{A}}$ accept since NL is closed under complement we know that there exists at least one accepting computation path in exactly one of these Turing machines and we did not simluate both the Turing machines along any such path. Therefore N also rejects x and stops the computation. Since N only simulates NL-Turing machines it shows that N is also a NL-Turing machine. This shows that $\mathrm{NL}^{\mathrm{NL}} = \mathrm{NL}$. □

THEOREM 2.23. *Let* 2SAT=$\{\phi | \phi$ *is a Boolean formula in the 2-CNF such that* ϕ *is satisfiable*$\}$. $\overline{\mathrm{2SAT}}$ *is in* NL *and* 2SAT *is also in* NL.

PROOF. We want to show that the language of all unsatisfiable Boolean formulae in the 2-CNF, denoted by $\overline{\mathrm{2SAT}}$, is in NL. Let $\phi = (\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_m)$ be in 2-CNF. We know that each ϕ_i is a conjunction of exactly two literals, where $1 \leq i \leq m$.

Now given a 2-CNF Boolean formula ϕ we define a directed graph $G = (V, E)$ as follows. For each variable x that occurs in ϕ we define two vertices in G. More precisely, we define a vertex for the variable x and a vertex for $\neg x$. Clearly if there are n variables in ϕ then there are $2n$ vertices in G. If $\phi_i = (\neg x \vee y)$ is a clause then we include directed edges (x, y) and $(\neg y, \neg x)$ in E of G. In other words, if we have a clause of ϕ to be $(\neg x \vee y)$ which is $(x \Rightarrow y)$ then we include the directed edge (x, y) in E. By commutativity, since $(\neg x \vee y)$ is also $(y \vee \neg x)$ which is $(\neg y \Rightarrow \neg x)$ we also include the edge $(\neg y, \neg x)$ in E of G. Clearly if there are m clauses in ϕ then there are $2m$ directed edges in G. Also it is easy to observe that there exists a directed path from a vertex α to a vertex β in G if and only if there exists a directed path from $\neg\beta$ to $\neg\alpha$ in G. The 2-CNF Boolean formula ϕ is the conjunction of implications of the form $(\alpha \Rightarrow \beta)$, where α and β are literals in the clauses of ϕ. Given the conjunction of two clauses $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \gamma)$ we can simplify this conjunction and derive the clause $(\alpha \Rightarrow \gamma)$ from them. Using this we infer that the 2-CNF Boolean formula ϕ is unsatisfiable if and only if we can obtain a contradiction which is that there exists at least one variable x such that we can simplify the conjunction of clauses in ϕ to obtain the clause $(x \Rightarrow \neg x)$ and also that $(\neg x \Rightarrow x)$. However this is equivalent to the existence of edges in E of G such that there is a directed path from x to $\neg x$ and a directed path from $\neg x$ to x.

Now given a 2-CNF Boolean formula ϕ such that n is the size of ϕ, a $O(\log n)$-space bounded deterministic Turing machine can output the adjacency matrix of the directed graph G described above. We then submit a query to the NL oracle to determine if there exists a directed path from a variable x to $\neg x$ and also from $\neg x$

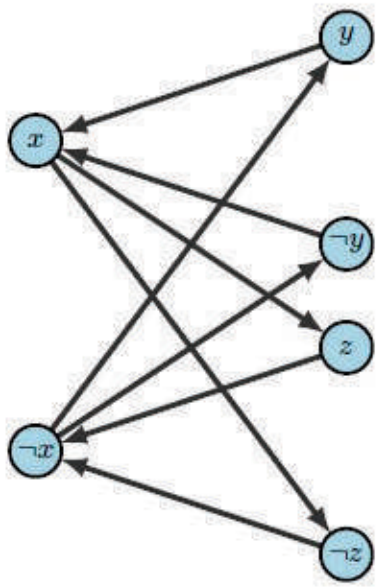

FIGURE 2.3. *This figure is the directed graph G obtained in the logspace many-one reduction from* $\overline{\text{2SAT}}$ *to* DSTCON. *We are given the Boolean formula* $\phi = (x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee z) \wedge (\neg x \vee \neg z)$ *as input and we obtain the directed graph G in this figure as the output of the reduction. Clearly, there are* 4 *clauses in* ϕ, 8 *vertices and* 8 *edges in G. It is easy to see that* ϕ *is not satisfiable and there is a directed path in fact, from every literal to its negation in G.*

to x in G for all the variables x that occur in ϕ. If for some variable x we get the oracle reply to be "YES" for the queries submitted then we accept ϕ and output that ϕ is unsatisfiable. Otherwise we reject the input ϕ and output that ϕ is satisfiable. Now it follows from Theorem 2.19 that $\overline{\text{2SAT}}$ is in NL. Using Theorem 2.18 it follows that $\text{2SAT} \in \text{NL}$. □

NOTE 2. *In the proof of Theorem 2.23 we first show that* $\overline{\text{2SAT}} \in \text{NL}$. *Given a subformula ψ of a Boolean formula ϕ which is in 2-CNF, we say that ψ is a contradiction if ψ is not satisfiable. We say that ψ does not have any redundant clause if there does not exist any clause ψ' in ψ such that ψ is unsatisfiable even after deleting ψ' from ψ. Any directed path from a variable x to $\neg x$ and therefore a directed path from $\neg x$ to x is a proof or a witness that the input 2-CNF formula ϕ is not satisfiable. This is equivalent to stating that a there exists a subformula ψ of ϕ which is a contradiction and which is free from any redundant clauses and that it is a proof that ϕ is not satisfiable. As a result counting the number of directed paths from a variable x to $\neg x$ is in* $\sharp$L. *Equivalently counting the number of subformulae*

of ϕ, each of which is a contradiction and which do not have any redundant clauses among them, is in $\sharp$L.

THEOREM 2.24. $\overline{\text{2SAT}}$ *is logspace many-one hard for* NL.

PROOF. We prove this statement by showing that DSTCON $\leq_m^{\text{L}}$ $\overline{\text{2SAT}}$. Let $G = (V, E)$ be a directed graph and let (G, s, t) be the input instance of DSTCON. Let $n = |V|$. We construct a 2-CNF Boolean formula ϕ from (G, s, t) such that there exists a directed path from s to t in G if and only if ϕ is not satisfiable. Let x be the Boolean variable corresponding to the vertex s in G and $\neg x$ be the literal corresponding to the vertex t. For each of the remaining vertices v_i in G, let there be a Boolean variable v_i. For each edge $(u, v) \in E$ let us include the clause $(u \Rightarrow v)$ in ϕ. In other words, for each edge $(u, v) \in E$ we include the clause $(\neg u \vee v)$ in ϕ. Apart from these clauses we also include the conjunction of the clause (x) in ϕ. Since the clause (x) is not a disjunction of two distinct literals we introduce a new variable y and include the conjunction of the following 2-CNF Boolean subformula $(x \vee y) \wedge (x \vee \neg y)$ in ϕ. Clearly a $O(\log n)$-space bounded deterministic Turing machine can output ϕ given (G, s, t) as input.

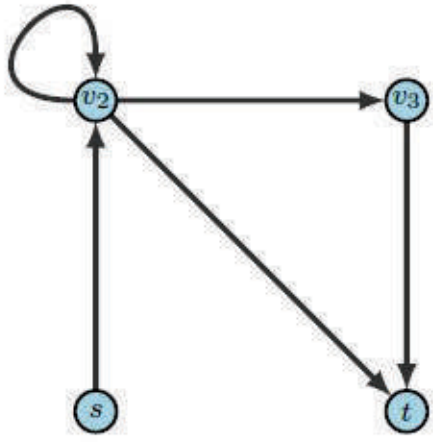

FIGURE 2.4. *This figure is a directed graph G in the reduction from* DSTCON *to* $\overline{\text{2SAT}}$. *We therefore obtain the Boolean formula $\phi = (\neg x \vee v_2) \wedge (\neg v_2 \vee v_2) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_2 \vee \neg x) \wedge (\neg v_3 \vee \neg x) \wedge (x \vee y) \wedge (x \vee \neg y)$. Since there exists a directed path from s to t in G it is easy to see that ϕ is not satisfiable.*

Now consider the case when there exists a directed path from s to t in G. Let the vertices that form the directed path be the following in sequence: $s, v_i, v_j, \ldots, v_k, t$, where $1 \leq i, j, k \leq n$. We claim that ϕ is not satisfiable. To prove this claim consider the case when $x = False$. Then ϕ is not satisfiable since we have clauses $(x \vee y) \wedge (x \vee \neg y)$ in ϕ. For the case when $x = True$ let us consider the 2-CNF Boolean subformula of ϕ formed by the clauses due to the edges in a directed path

from s to t in G: $(\neg x \vee v_i) \wedge (\neg v_i \vee v_j) \wedge \cdots \wedge (\neg v_k \vee \neg x)$. It is easy to see that the Boolean subformula $(\neg x \vee v_i) \wedge (\neg v_i \vee v_j) \wedge \cdots \wedge (\neg v_k \vee \neg x)$ is not satisfiable and so ϕ is also not satisfiable.

Conversely, assume that there does not exist any directed path from s to t in G. Let A be the subset of vertices in V that are reachable from s in G, B be the subset of vertices in V from which t is reachable and $C = V \setminus (A \cup B)$. Since there is no directed path from s to t in G it is easy to observe that the subsets of V of vertices in A, B and C are pairwise disjoint and there are no edges from A to $B \cup C$ and from $A \cup C$ to B in G. Now let us assign the truth value T to every

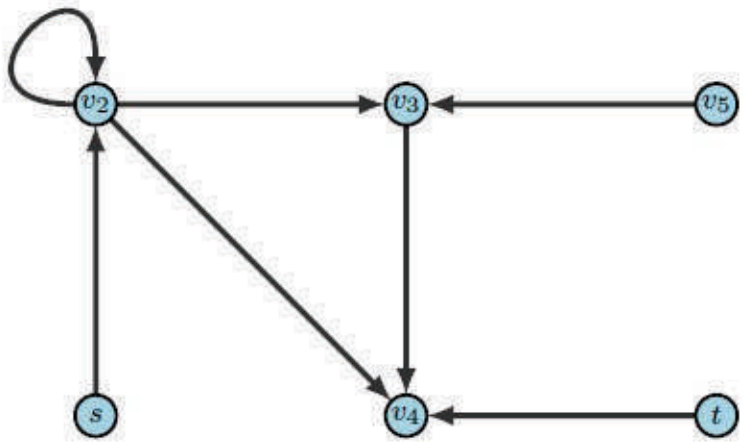

FIGURE 2.5. *This figure is a directed graph G in the reduction from* DSTCON *to* $\overline{\text{2SAT}}$. *We therefore obtain the Boolean formula* $\phi = (\neg x \vee v_2) \wedge (\neg v_2 \vee v_2) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_2 \vee v_4) \wedge (\neg v_3 \vee v_4) \wedge (\neg v_5 \vee v_3) \wedge (\neg x \vee v_4) \wedge (x \vee y) \wedge (x \vee \neg y)$. *Here* $A = \{x, v_2, v_3, v_4\}, B = \{t\}$ *and* $C = \{v_5\}$. *Clearly,* $A \cap B = B \cap C = A \cap C = \emptyset$. *Also* $\nexists$ *any directed edge from a vertex in A to $B \cup C$ and from a vertex in $A \cup C$ to B. Since there does not exist any directed path from s to t in G it is easy to see that ϕ is satisfiable: we can assign the truth value* True *to every variable in $A \cup C$ including the variable x and the truth value* False *to the literal $\neg x$ corresponding to the vertex t. We assign either the truth value* True *or* False *to the variable y.*

variable in $A \cup C$ including the variable x and the truth value F to every variable in B including the literal $\overline{x}$. It is easy to verify that this truth value assignment satisfies ϕ. This shows that DSTCON $\leq_m^{\mathrm{L}} \overline{\text{2SAT}}$. □

THEOREM 2.25. 2SAT *is logspace many-one complete for* NL.

PROOF. This result follows from Theorem 2.23, Theorem 2.24 and Theorem 2.18. □

CONJECTURE: Counting the number of satisfying assignments of a 2SAT formula, denoted by ♯2SAT, is logspace many-one complete for ♯L.

2.2.2. Non-deterministic space bounded complexity classes above NL.

DEFINITION 2.26. Let Σ be the input alphabet and $S(n) \in \Omega(\log n)$. We define the complexity class NSPACE$(S(n)) = \{L \subseteq \Sigma^* | \exists$ a $O(S(n))$ space bounded non-deterministic Turing machine M such that $L = L(M)\}$.

DEFINITION 2.27. Let Σ be the input alphabet and $S(n) \in \Omega(\log n)$. We define the complexity class co-NSPACE$(S(n)) = \{\overline{L} \subseteq \Sigma^* | L \in$ NSPACE$(S(n))\}$.

THEOREM 2.28. **(The Immerman-Szelepcsenyi Theorem)**NSPACE$(S(n))$ *is closed under complement, where* $S(n) \in \Omega(\log n)$. *In other words,* NSPACE $(S(n))$ =co-NSPACE$(S(n))$*, where* $S(n) \in \Omega(\log n)$.

PROOF. Let Σ be the input alphabet and let $L \subseteq \Sigma^*$ such that $L \in$ NSPACE $(S(n))$. Let M be the NSPACE$(S(n))$ Turing machine that accepts L. As in the proof of Lemma 2.5 we observe that given an input $x \in \Sigma^*$, all the configurations of $M(x)$ is computable by a deterministic Turing machine using space at most $O(S(n))$ and therefore the adjacency matrix of the configuration graph of $M(x)$, which is a directed graph, can also be output by a $O(S(n))$ space bounded deterministic Turing machine. As in Theorem 2.10 we observe that the directed st-connectivity problem for the simple layered directed acyclic graph obtained from the configuration graph G of a NSPACE$(S(n))$ Turing machine $M(x)$ is a canonical complete problem for NSPACE$(S(n))$. Now following the algorithm in the proof of Theorem 2.18 we are able to decide if there does not exist a directed path from a vertex s in layer 1 of the above graph G to the vertex t in layer n of G in NSPACE$(S(n))$. This shows that co-NSPACE$(S(n)) \subseteq$ NSPACE$(S(n))$ from which our theorem follows. □

2.3. Logarithmic Space Bounded Counting classes

In this section we assume without loss of generality that the input alphabet $\Sigma = \{0, 1\}$. We informally say that a complexity class $\mathcal{C}$ is a logarithmic space bounded counting class if the criterion to decide if an input string $x \in \Sigma^*$ is in a language $L \in \mathcal{C}$ is based on the number of accepting computation paths and/or the number of rejecting computation paths of a $O(\log n)$ space bounded non-deterministic Turing machine M. **It follows from this informal definition that** NL **is the first and fundamental logarithmic space bounded counting class.** We recall the defintion of ♯L from Definition 2.11.

PROPOSITION 2.29. *Let* Σ *be the input alphabet and let* M *be a* $O(\log n)$*-space bounded non-deterministic Turing machine. Counting the number of computation paths of* M *on input* x *is in* ♯L*, where* $x \in \Sigma^*$.

PROOF. We define a $O(\log n)$-space bounded non-deterministic Turing machine M' which simulates M on input $x \in \Sigma^*$ such that M' accepts x along all computation paths. In other words, given any input $x \in \Sigma^*$, if $M(x)$ accepts then

$M'(x)$ also accepts. Otherwise if $M(x)$ rejects then $M'(x)$ still accepts. As a result $L(M') = \Sigma^*$. Also the number of computation paths of M' on x is equal to $acc_{M'}(x) = acc_M(x) + rej_M(x)$, which is the number of computation paths of M on x, where $x \in \Sigma^*$. □

DEFINITION 2.30. Let Σ be the input alphabet. The complexity class GapL is defined to be the class of functions $f : \Sigma^* \to \mathbb{Z}$ such that there exists a NL-Turing machine M for which we have $f(x) = acc_M(x) - rej_M(x)$ where $acc_M(x)$ and $rej_M(x)$ denote the number of accepting computation paths and the number of rejecting computation paths of M on any input $x \in \Sigma^*$ respectively. We also denote $(acc_M(x) - rej_M(x))$ by $gap_M(x)$.

In Section 1.1.2, we have assumed that the computation binary tree of an $O(S(n))$-space bounded nondeterministic Turing machine is a complete binary tree. That is, all computation paths in the computation binary tree of an $O(S(n))$-space bounded nondeterministic Turing machine have the same length. However, note that under this assumption any $f \in \mathrm{GapL}$ will never be an odd integer on any input $x \in \Sigma^*$, where Σ is the input alphabet, since the number of leaves in the computation tree is an integer which is a power of 2. This suggests an unrealistic computation tree. So to overcome this problem, we assume that in many places such as when considering GapL functions that the computation binary tree for $L \in \mathrm{NL}$ contains computation paths which are not all of the same length. In other words, the computation binary tree of a non-deterministic Turing machine on any given input is not necessarily a complete binary tree.

DEFINITION 2.31. A language L belongs to the logarithmic space bounded counting class PL if there exists a GapL function f such that $x \in L$ if and only if $f(x) > 0$.

DEFINITION 2.32. A language L belongs to the logarithmic space bounded counting class $\mathrm{C_=L}$ if there exists a GapL function f such that $x \in L$ if and only if $f(x) = 0$.

We recall the definition of FL from Definition 1.34. We need the following result.

PROPOSITION 2.33. (Folklore)

(1) *Let Σ be the input alphabet and $f : \Sigma^* \to \mathbb{Z}$ such that $f(x) \geq 0$, $\forall x \in \Sigma^*$ and $f \in$ FL. Then $f \in \sharp\mathrm{L}$.*
(2) *$\sharp\mathrm{L}$ is closed under addition.*
(3) *$\sharp\mathrm{L}$ is closed under multiplication.*
(4) *GapL is closed under addition.*
(5) *GapL is closed under multiplication.*

PROOF. (1) Let f be a FL function that takes non-negative values on all inputs. Given an input $x \in \Sigma^*$, without loss of generality, we assume that $f(x) \in \mathbb{Z}^+$ is represented in binary notation.

Let us define a NL-Turing machine M, which on input x first computes $f(x)$, and thereby finds the size of $f(x)$. If $f(x) = 0$ then M rejects x and stops. Otherwise, let l denote the size of $f(x)$. It is easy to see that $l \in O(\log n)$, where $n = |x|$. Given $f(x)$, we denote the i^{th} bit of $f(x)$ by $f(x)_i$, where $1 \leq i \leq l$. Let $f(x) = f(x)_l f(x)_{l-1} \cdots f(x)_1$. Here $f(x)_l$ denotes the most significant bit of $f(x)$ and $f(x)_1$ denotes the least significant bit of $f(x)$. Since we assume that $f(x) \neq 0$, we have $f(x)_l = 1$ always.

After finding l, let us define M as follows: M starts by sequentially choosing exactly l bits from $\{0, 1\}$ in a non-deterministic fashion. For $1 \leq i \leq l$, after choosing the i^{th} bit, M computes $f(x)_{l-i+1}$. If $f(x)_{l-i+1} = 1$ and the i^{th} bit chosen by M is 0, then M accepts the input x along all of its computation paths obtained by choosing the remaining $(l - i)$ bits and stops. Instead, if $f(x)_{l-i+1}$ equals the i^{th} bit chosen by M, then M continues to iterate the above step of choosing the $(i + 1)^{st}$ bit non-deterministically as long as $(i + 1) \leq l$. However, if $f(x)_{l-i+1} = 0$ and the i^{th} bit chosen by M is 1 then M rejects the input x along all of its computation paths obtained by choosing the remaining $(l - i)$ bits and stops.

It is easy to see that M accepts x if and only if any of the computation paths which M has guessed is among the first $f(x)$ lexicographically least computation paths in the computation tree of M which has depth l and 2^l leaves. Clearly $acc_M(x) = f(x)$ whenever $f(x) \geq 0, \forall x \in \Sigma^*$.

(2) Let $f_1, f_2 \in \sharp\mathrm{L}$. Since we can define a NL-Turing machine which first branches once and simulates the NL-Turing machine corresponding to f_1 along one branch and the NL-Turing machine corresponding to f_2 along the other branch, we get the result that $(f_1 + f_2) \in \sharp\mathrm{L}$.

(3) Let $f_1, f_2 \in \sharp\mathrm{L}$. We define a NL-Turing machine M which simulates the NL-Turing machine corresponding to f_1 and then at the end of all of its accepting computation paths it simulates the NL-Turing machine corresponding to f_2. At the end of the rejecting computation paths of the NL-Turing machine corresponding to f_1, M also rejects the input. It is the easy to see that $(f_1 f_2) \in \sharp\mathrm{L}$.

(4) Proof is identical to (2).

(5) Let $f_1, f_2 \in \mathrm{GapL}$. We define a NL-Turing machine M which simulates the NL-Turing machine corresponding to f_1 and then at the end of all of its computation paths it simulates the NL-Turing machine corresponding to f_2. M accepts if and only if the NL-Turing machine corresponding to f_2 accepts. It is the easy to see that $(f_1 f_2) \in \mathrm{GapL}$.

□

LEMMA 2.34. *Let Σ be the input alphabet and $L \subseteq \Sigma^*$ be a language in* NL. *$\exists f \in \sharp\mathrm{L}$ such that for any given input $x \in \Sigma^*$, we have $x \in L$ if and only if $f(x) > N$, where $2 \times N$ is the number of computation paths of the* NL*-Turing machine M of f.*

PROOF. Let M' be the NL-Turing machine that accepts L. For any given input $x \in \Sigma^*$, we know that $x \in L$ if and only if $acc_{M'}(x) > 0$. Let M'' be the $O(\log n)$-space bounded non-deterministic Turing machine as shown in Proposition 2.29 which accepts Σ^* such that, given any input $x \in \Sigma^*$, M'' simulates M' and finally accepts x and never rejects any input x. It is easy to see that the number of accepting computation paths of $M''(x)$, denoted by $acc_{M''}(x)$, is equal to the number of computation paths of M' on x.

Given an input $x \in \Sigma^*$, let M be the non-deterministic Turing machine that non-deterministically decides to either simulate M' on input x along the left branch or it decides to simulate M'' on input x on the right branch of its computation tree. It is easy to see that, given input $x \in \Sigma^*$, the number of computation paths of M on x is equal to $2 \times N$, where N is the number of computation paths of M' on x. Also if $x \in L$ then we have $acc_M(x) > N$. Otherwise if $x \notin L$ then $acc_M(x) = N$. We take $f \in \sharp\mathrm{L}$, as the $\sharp\mathrm{L}$ function which is the number of accepting computation paths of M on input $x \in \Sigma^*$. □

THEOREM 2.35. $\mathrm{NL} \subseteq \mathrm{PL}$.

PROOF. Follows from Definition 2.31 and Lemma 2.34. □

DEFINITION 2.36. The symmetric difference of two sets A and B is the set of elements that belong to exactly one of A and B. The symmetric difference of two sets A and B is denoted by $A \triangle B$. In other words, $A \triangle B = (\overline{A} \cap B) \cup (A \cap \overline{B})$.

DEFINITION 2.37. Let Σ be the input alphabet. We define $(\mathrm{NL}\triangle\mathrm{NL}) = \{L \subseteq \Sigma^* | \exists L_1, L_2 \in \mathrm{NL}$ and $L = L_1 \triangle L_2$, where $L_1, L_2 \subseteq \Sigma^*\}$.

THEOREM 2.38. $(\mathrm{NL}\triangle\mathrm{NL}) \subseteq \mathrm{L}^{\mathrm{NL}}$.

PROOF. Given two sets A and B it follows from the definition that $(A \triangle B) = (\overline{A} \cap B) \cup (A \cap \overline{B})$. Now, let $A, B \in \mathrm{NL}$. Due to Theorem 2.18 and Corollary 2.20, it follows that $(A \triangle B) \in \mathrm{NL}$. Clearly, to determine if an input string $x \in \Sigma^*$ is in $(A \triangle B)$ a logarithmic space bounded deterministic Turing machine that has access to a NL oracle shall make one query and output if x is in $(A \triangle B)$ or not. This shows that $(\mathrm{NL}\triangle\mathrm{NL}) \subseteq \mathrm{L}^{\mathrm{NL}}$. □

THEOREM 2.39. $\mathrm{NL} = (\mathrm{NL}\triangle\mathrm{NL})$.

PROOF. Let Σ be the input alphabet and let $L \subseteq \Sigma^*$ be a non-trivial language such that $L \in \mathrm{NL}$. It is easy to see that $L = L \triangle \emptyset$, where $\emptyset$ denotes the empty set. We therefore get $\mathrm{NL} \subseteq \mathrm{NL}\triangle\mathrm{NL}$. Converse follows from Theorem 2.38 and the closure of NL under $\leq^{\mathrm{L}}_{\mathrm{T}}$ reductions, which we have shown in Theorem 2.19. □

LEMMA 2.40. *Let Σ be the input alphabet, $L \subseteq \Sigma^*$, $L \in \mathrm{NL}$ and let M be a* NL*-Turing machine such that $L(M) = L$. There exists a* NL*-Turing machine M' such that $L(M') = L$ and, on any input $x \in \Sigma^*$, the computation tree of M' is a complete binary tree such that it has has exactly 2^{n^k} computation paths and $acc_{M'}(x) = acc_M(x)$, where $n = |x|$ and $k > 0$.*

PROOF. Let M be the NL-Turing machine such that the number of configurations of M on input $x \in \Sigma^*$ is $\leq n^c$, where $c > 0$ is a constant. Without loss of generality, we assume that the computation tree of M on x is a binary tree which has finite number of nodes and edges. Let us define another NL-Turing machine M' which on input x does the following:

(1) simulate M on x, and
(2) start a (deterministic) counter ctr with its initial value n^k, where $k > c > 0$, and decrement ctr by 1 in each step until $ctr > 0$.

In the simulation of M by M', each configuration we encounter has either 0 or 1 or 2 successors.

- if the present configuration has 2 successors, then M' also has two successor configurations and it continues to simulate M on x,
- if the present configuration has 1 successor, then M' is defined such that it has 2 successor configurations among which one successor configuration continues to simulate M on x, and the other is the root of a subtree which rejects along all of its computation paths until $ctr > 0$.
- if the present configuration has 0 successors then it is a halting configuration of M on x. If the halting configuration is a rejecting configuration, then M' forms a rejecting subtree, as done above by decrementing ctr by 1 in each step until $ctr > 0$, with this node as the root. Otherwise if the halting configuration is an accepting configuration, then M' will generate a complete subtree, as done above by decrementing ctr by 1 in each step until $ctr > 0$, wherein the lexicographically least computation path accepts and remaining computation paths of this subtree reject the input.

It is easy to see that the computation tree of M' is a complete binary tree and $acc_{M'}(x) = acc_M(x)$. □

LEMMA 2.41. NL $\subseteq$ co-C$_=$L.

PROOF. Let $L \subseteq \Sigma^*$ such that $L \in$ NL and let M' be the NL-Turing machine which accepts L as in Lemma 2.40. Let $f \in \sharp$L such that on any input $x \in \Sigma^*$ we have $f(x) = acc_{M'}(x),\ \forall x \in \Sigma^*$. For any input $x \in \Sigma^*$ such that $n = |x|$, we assume without loss of generality that all the computation paths of the computation tree of M' on input x have length n^k, where $k > 0$ is a constant and which clearly depends on the $O(\log n)$ space used by $M'(x)$ in any of its computation paths.

Algorithm 4 NL-contained-in-co-C$_=$L
Input: $x \in \Sigma^*$
Output: *accept* if $x \in L$, where $L \in$ NL. Otherwise *reject*.
Complexity: co-C$_=$L.

1: Non-deterministically choose one of the following two options.
2: Simulate $M(x)$, or
3: Form a computation tree having depth n^k and accept the input x on all of these computation paths

Let us consider the algorithm given above which can be implemented by a NL-Turing machine, say M'', on a given input $x \in \Sigma^*$, where $n = |x|$. On any given input $x \in \Sigma^*$, if $x \in L$ then $f(x) \geq 1$. As a result $gap_{M''}(x) = acc_{M''}(x) - rej_{M''}(x) > 0$. However if $x \notin L$ then all the computation paths of M' on input x end in the rejecting configuration and therefore $acc_{M''}(x) = rej_{M''}(x)$ which clearly implies $gap_{M''}(x) = 0$. This shows that $L \in$ co-C$_=$L from which we get NL $\subseteq$ co-C$_=$L. □

THEOREM 2.42. NL $\subseteq$ C$_=$L.

PROOF. From Lemma 2.41 we infer that co-NL $\subseteq$ C$_=$L. Also as a consequence of Theorem 2.18 we obtain that NL $\subseteq$ C$_=$L. □

PROPOSITION 2.43. C$_=$L *is closed under union.*

PROOF. It follows from Proposition 2.33 due to the closure of GapL under multiplication. □

2.3.1. Properties of ♯L and GapL functions. We recall the definition of ♯L and GapL from Definitions 2.11 and 2.30

LEMMA 2.44. *Let Σ be the input alphabet. If $f(x)$ is a* ♯L *or* GapL *function then there exists a polynomial $p(n)$ such that the absolute value of $f(x)$ is bounded above by $2^{p(n)}$ for every $x \in \Sigma^*$, where $n = |x|$.*

PROOF. Either $f(x) = acc_M(x)$ or $gap_M(x)$ for some NL-Turing machine M. Let $s(n) \in O(\log n)$ be an upper bound the amount of space that M uses on input strings of length n. It is easy to note that the running time of $M(x)$ is at most $p(|x|) = 2^{s(|x|)}$, which is a polynomial in $|x|$. As a result the number of distinct computation paths of M on any input x is upper bounded by $2^{p(|x|)}$ from which it follows that the number of accepting computation paths and rejecting computation paths of M on any input x is at most $2^{p(|x|)}$ and this proves our claim. □

PROPOSITION 2.45. *If $f \in$ GapL then $-f \in$ GapL.*

THEOREM 2.46. *Let Σ be the input alphabet. For every* NL*-Turing machine M, there is a* NL*-Turing machine N such that $gap_N(x) = acc_M(x)$, on any input $x \in \Sigma^*$.*

PROOF. Given an input x, let $\overline{M}(x)$ denote the NL-Turing machine obtained by reversing the decision of computation paths of $M(x)$. In other words, $\overline{M}(x)$ rejects every accepting computation path of $M(x)$ and accepts every rejecting computing path of $M(x)$. Our NL-Turing machine N guesses a path p of $M(x)$. If p is accepting, N accepts. Otherwise, N branches once, accepting on one branch and rejecting on the other. We have for all x, $gap_N(x) = acc_N(x) - rej_N(x) = acc_N(x) - acc_{\overline{M}}(x) = (acc_M(x) + acc_{\overline{M}}(x)) - acc_{\overline{M}}(x) = acc_M(x)$. Note that all the computation paths of N are not necessarily having the same length. □

COROLLARY 2.47. ♯L $\subset$ GapL.

PROOF. Follows from Theorem 2.46 and the fact that GapL contains functions that take negative values on many inputs. □

Note that a $\sharp$L function can take only nonnegative values and so this class cannot capture all the functions in FL that evaluate to integer values on any given input. However, it does capture those functions in FL that only take nonnegative integer values on all inputs as shown in Proposition 2.33. The class GapL includes any function in FL that takes integer values on any input without any restriction. For the rest of this chapter, we assume without loss of generality that, any function $f \in$ FL which we consider always takes integer values on any input.

THEOREM 2.48. *Every* FL *function* f *is in* GapL.

PROOF. Let $f \in$ FL. Define a NL-Turing machine N which on input x first computes $f(x)$. For any $f(x)$, since the absolute value of $f(x)$ (denoted by $|f(x)|$) is greater than 0, it follows from Proposition 2.33 that there exists a NL-Turing machine N'' such that $acc_{N''}(x) = |f(x)|$ on any input $x \in \Sigma^*$. Using Corollary 2.47 we get that there exists a NL-Turing machine N' such that $gap_{N'}(x) = acc_{N''}(x) = |f(x)|$. Therefore, after computing $f(x)$, if $f(x) > 0$ then N simulates N' on input x. However, if $f(x) < 0$, then N simulates N' on input x and rejects at the end of the computation paths in which N' accepts and accepts at the end of the computation paths in which N' rejects. This shows that $gap_N(x) = f(x)$. □

THEOREM 2.49. *Let* Σ *be the input alphabet. If* $h \in$ GapL*, then there exists* $f, g \in \sharp$L *such that* $h(x) = f(x) - g(x)$ *on any input* $x \in \Sigma^*$*. Conversely, if* $f, g \in \sharp$L*, then there exists* $h \in$ GapL *such that* $h(x) = f(x) - g(x)$ *on any input* $x \in \Sigma^*$*. In other words, we can write every function in* GapL *as the difference of two functions in* $\sharp$L*. Equivalently, we say that* GapL $= \sharp$L $- \sharp$L.

PROOF. Let $h \in$ GapL and let M denote the NL-Turing machine corresponding to h. Given a NL-Turing machine M and an input x, let $\overline{M}(x)$ denote the NL-Turing machine obtained by reversing the decision of computation paths of $M(x)$. In other words, $\overline{M}(x)$ rejects every accepting computation path of $M(x)$ and accepts every rejecting computing path of $M(x)$. For any M we have $gap_M(x) = acc_M(x) - acc_{\overline{M}}(x)$ by definition, so if $h \in$ GapL then h is the difference of two $\sharp$L functions.

Conversely, let $f, g \in \sharp$L. Using Corollary 2.47 we get that $f, g \in$ GapL. It follows from Theorem 2.46 that there exists NL-Turing machines M and N such that $gap_M(x) = f(x)$ and $gap_N(x) = g(x)$ on any input $x \in \Sigma^*$. We use Proposition 2.45 and Lemma 2.33 and obtain that $(f(x) - g(x)) = (gap_M(x) - gap_N(x)) \in$ GapL, on any input $x \in \Sigma^*$. □

THEOREM 2.50. *Let* Σ *be the input alphabet. If* $h \in$ GapL*, then there exists* $f \in \sharp$L *and* $g \in$ FL *such that* $h(x) = f(x) - g(x)$ *on any input* $x \in \Sigma^*$*. Conversely, if* $f \in \sharp$L *and* $g \in$ FL*, then there exists* $h \in$ GapL *such that* $h(x) = f(x) - g(x)$ *on any input* $x \in \Sigma^*$*. In other words, we can write every function in* GapL *as the difference of a function in* $\sharp$L *and a function in* FL*. Equivalently, we say that* GapL $= \sharp$L $-$ FL.

PROOF. Let $h \in$ GapL. We know from Theorem 2.49 that there exists $f, g \in \sharp$L such that, on any given input $x \in \Sigma^*$, we have $h(x) = f(x) - g(x)$. Let M_f

and M_g be NL-Turing machines corresponding to f and g respectively. As shown in Lemma 2.40, we assume that the computation tree of both f and g are complete binary trees such that both M_f and M_g have exactly 2^{n^k} computation paths on any input of length n, where $k > 0$ is a constant. Let $M_{\overline{g}}$ be the NL-Turing machine which reverses the decision of a computation path of M_g. In other words, along any computation path, $M_{\overline{g}}$ accepts if and only if M_g rejects. Let us consider a Turing machine M which non-deterministically branches once and simulates M_f along the left branch and $M_{\overline{g}}$ along the right branch. Then, given any input $x \in \Sigma^*$, we have

$$\begin{aligned}(f(x) - g(x)) &= acc_{M_f}(x) - acc_{M_g}(x) \\ &= acc_{M_f}(x) + acc_{M_{\overline{g}}}(x) - 2^{n^k} \\ &= acc_M(x) - 2^{n^k}.\end{aligned}$$

Since a NL-Turing machine can output 2^{n^k} in binary notation when it is given an input $x \in \Sigma^*$ such that $n = |x|$, we get $h = (f - g) \in \sharp\mathrm{L} - \mathrm{FL}$.

Conversely, let $f \in \sharp\mathrm{L}$ and let $g \in \mathrm{FL}$. We know from Theorem 2.46 that there exists a NL-Turing machine N_1 such that $gap_{N_1}(x) = f(x)$, for any $x \in \Sigma^*$. We also know from Theorem 2.48 that $g \in \mathrm{GapL}$. Therefore there exists a NL-Turing machine N_2 such that $gap_{N_2}(x) = g(x)$, for any $x \in \Sigma^*$. It is now easy to see using Propositions 2.45 and 2.33 that $(f - g) \in \mathrm{GapL}$. □

We first recall the notion of an oracle Turing machine from Section 1.1.4. We also recall the notion of having a function as an oracle from Section 1.1.5. Let us also recall the definition of logspace Turing reducibility using functions from Definitions 1.39 and 1.40.

NOTE 3. (1) *In Definition 1.39, if we replace L_2 by a function $f \in \mathcal{F}$, where $\mathcal{F}$ is a complexity class of functions such as $\sharp\mathrm{L}$ then $L_1 \leq_{\mathrm{T}}^{\mathrm{L}} f$ and we say that M^f accepts L_1. In this context of having a function f such as $\sharp$DSTCON as an oracle, we note that to obtain the value of the oracle function when it is given an oracle query string as input, the $O(\log n)$-space bounded deterministic Turing machine has to submit at least one and at most $p(n)$ many queries, where $p(n)$ is a polynomial in n, and n is the size of the input.*

(2) *In Definition 1.40 if we replace $\mathcal{C}$ by $\mathcal{F}$, where $\mathcal{F}$ is a complexity class of functions such as $\sharp\mathrm{L}$ then $\mathrm{L}^{\mathcal{F}}$ is the complexity class of all languages that is accepted by a $O(\log n)$ space bounded deterministic Turing machine that has access to a function $f \in \mathcal{F}$ as an oracle.*

DEFINITION 2.51. We define $\mathrm{FL}^{\sharp\mathrm{L}}$ to be the complexity class of all functions that are logspace Turing reducible to the complexity class of functions in $\sharp\mathrm{L}$.

LEMMA 2.52. $\mathrm{GapL} \subseteq \mathrm{FL}^{\sharp\mathrm{L}}$.

PROOF. We know from Proposition 2.15 that $\sharp$DSTCON is logspace many-one complete for $\sharp\mathrm{L}$. Let Σ be the input alphabet. It follows from Proposition 2.17 that if $L \in \mathrm{NL}$ such that M is the NL-Turing machine which decides if an input $x \in \Sigma^*$

is in L, then there exists an instance (G, s, t) of DSTCON which is obtained using the logspace many-one reduction shown in Theorem 2.5 that $acc_M(x)$ is equal to the number of directed paths from s to t in G.

Let $f \in \mathrm{GapL}$ and let M be the NL-Turing machine such that given an input $x \in \Sigma^*$, we have $f(x) = acc_M(x) - rej_M(x)$. Let n be the size of the input x. Let $\overline{M}$ be the NL-Turing machine which is obtained from M by reversing the decision of the computation paths of $M(x)$. In other words, $\overline{M}(x)$ rejects every accepting computation path of $M(x)$ and accepts every rejecting computation path of $M(x)$. As a result, $acc_{\overline{M}}(x) = rej_M(x)$. Since $\sharp$DSTCON is logspace many-one complete for $\sharp\mathrm{L}$, using a $O(\log n)$-space bounded determinsitic Turing machine M', we can obtain 2 instances (G_1, s_1, t_1) and (G_2, s_2, t_2) of DSTCON from x such that $acc_M(x)$ is equal to the number of directed paths from s_1 to t_1 in G_1 and $acc_{\overline{M}}(x)$ is equal to the number of directed paths from s_2 to t_2 in G_2. M' can therefore use these 2 instances of DSTCON and submit $p(n)$ many oracle queries to find $acc_M(x)$ and $acc_{\overline{M}}(x) = rej_M(x)$ in a bit-wise manner, where $p(n)$ is a polynomial in n. After finding these values, M' computes $f(x) = acc_M(x) - rej_M(x)$ and outputs it in the output tape of M'. □

PROPOSITION 2.53. $\mathrm{L}^{\mathrm{L}^{\sharp\mathrm{L}}} = \mathrm{L}^{\sharp\mathrm{L}}$.

PROOF. It is trivial to show that $\mathrm{L}^{\sharp\mathrm{L}} \subseteq \mathrm{L}^{\mathrm{L}^{\sharp\mathrm{L}}}$. We have to therefore show that $\mathrm{L}^{\mathrm{L}^{\sharp\mathrm{L}}} \subseteq \mathrm{L}^{\sharp\mathrm{L}}$. Let $L \in \mathrm{L}^{\mathrm{L}^{\sharp\mathrm{L}}}$ such that there exists a deterministic $O(\log n)$ space bounded oracle Turing machine $N^{L'}$ which correctly decides if any input string $x \in \Sigma^*$ is in L, where $L' \in \mathrm{L}^{\sharp\mathrm{L}}$ is the oracle for N. Let M' denote a deterministic oracle Turing machine which correctly decides if an input string $y' \in L'$.

Let M be a deterministic $O(\log n)$ space bounded Turing machine which simulates $N^{L'}$ on the given input string $x \in \Sigma^*$ till $N^{L'}$ generates an oracle query y' which is to be submitted to the oracle L'. In other words, M continues its simulation of $N^{L'}$ and stops its simulation in the step before it enters the q_{QUERY} state. Now M stores the present configuration of $N^{L'}$ which includes its present state, position of the tape head of the input tape, the contents of the work tape, and the position of the tape head of the work tape. Clearly M requires at most $O(\log n)$ space to store this information. M then simulates M' on the query y' with access to a function in $\sharp\mathrm{L}$ as the oracle. Based on this simulation of M' by M, it is possible to correctly decide if the oracle query string y' is in L'. Therefore after correctly deciding if y' is in the oracle, M restores itself to the state of $N^{L'}$ which it had saved and continues with its simulation repeating these steps whenever a oracle query string is generated. As a result it correctly decides if the input string $x \in L$ or not. Since M simulates a $\mathrm{L}^{\sharp\mathrm{L}}$ computation and uses at most $O(\log n)$ space we get that $L \in \mathrm{L}^{\sharp\mathrm{L}}$. □

PROPOSITION 2.54. $\mathrm{L}^{\mathrm{L}^{\mathrm{GapL}}} = \mathrm{L}^{\mathrm{GapL}}$.

THEOREM 2.55. $\mathrm{L}^{\mathrm{GapL}} = \mathrm{L}^{\sharp\mathrm{L}}$.

PROOF. It follows from Corollary 2.47 that $\sharp\mathrm{L} \subset \mathrm{GapL}$. As a result $\mathrm{L}^{\sharp\mathrm{L}} \subseteq \mathrm{L}^{\mathrm{GapL}}$. To prove the other way inclusion, we have already shown that $\mathrm{GapL} \subseteq \mathrm{L}^{\sharp\mathrm{L}}$ in Lemma 2.52. So using Proposition 2.53, we get $\mathrm{L}^{\mathrm{GapL}} \subseteq \mathrm{L}^{\sharp\mathrm{L}}$. We therefore get $\mathrm{L}^{\mathrm{GapL}} = \mathrm{L}^{\sharp\mathrm{L}}$. □

LEMMA 2.56. *Let f be a* FL *function and g be a* $\sharp\mathrm{L}$ *or* GapL *function. Then $g(f(x))$ is a* $\sharp\mathrm{L}$ *or* GapL *function, respectively.*

PROOF. Let M be a NL-Turing machine that defines a $\sharp\mathrm{L}$ function g. Define N to be a NL-Turing machine that on input x simulates $M(f(x))$. Then $acc_N(x)$ is exactly $g(f(x))$. The proof for GapL is identical. □

We refer to Definition A.3 and Theorem A.4 in the Appendix A for $\binom{n}{k}$, where $n, k \in \mathbb{Z}^+$.

THEOREM 2.57. *If $f \in \sharp\mathrm{L}$ and $k \in \mathbb{Z}^+$ is a constant, then $g(x) = \binom{f(x)}{k} \in \sharp\mathrm{L}$.*

PROOF. Let M be a NL-Turing machine which has $f(x)$ accepting computation paths on any input $x \in \Sigma^*$. We now define a NL-Turing machine N such that the number of accepting computation paths of N on input $x \in \Sigma^*$ is $\binom{f(x)}{k}$.

Without loss of generality, assume that $k > 0$. Otherwise, if $k = 0$ the NL-Turing machine N is defined such that it accepts the input $x \in \Sigma^*$ at the end of the lexicographically least computation path and rejects on all the other computation paths. Now let $k > 0$. N starts to simulate M on the given input $x \in \Sigma^*$, and it first stores the initial configuration of M on the work tape. Clearly, this requires $O(\log n)$ space, where $n = |x|$. N maintains a counter which is initialized to 1. N will store constantly many configurations of M on its work tape and the counter will keep track of how many configurations have been stored.

Now every time when N has to make a non-deterministic choice, in an iterative manner N cycles through all the configurations stored on its work tape and chooses one of the following three possibilities:

(a) replace the configuration which it is at present considering by its left successor configuration

(b) replace the configuration which it is at present considering by its right successor configuration

(c) replace the configuration by its left successor configuration and add the right successor configuration to the list of configurations stored on the work tape, and increment the counter by 1.

In the above non-determinisitic algorithm, N rejects x if the counter exceeds k, or if any of the stored configurations are rejecting configurations of M on x, or if all the configurations are accepting; however the counter is less than k. N accepts x if all the configurations are accepting and the counter is exactly k. Clearly, the space used by N is $O(k \log n) = O(\log n)$ since k is a constant, where $n = |x|$. Also it is clear that N chooses k many distinct computation paths from the computation of $M(x)$. As a result, the number of accepting computation paths of N on input $x \in \Sigma^*$ is $\binom{f(x)}{k}$. □

Let us recall Proposition 2.33.

THEOREM 2.58. (1) *Let Σ be the input alphabet, $f \in \sharp\mathrm{L}$ and $p(n)$ be a polynomial. Let $h : \Sigma^* \to \mathbb{Z}$ be defined for all $x \in \Sigma^*$ by $h(x) = \Sigma_{1 \leq i \leq p(|x|)} f(\langle x, i \rangle)$. $h \in \sharp\mathrm{L}$.*

(2) *Let Σ be the input alphabet, $f \in \sharp\mathrm{L}$ and $p(n)$ be a polynomial. Let $h : \Sigma^* \to \mathbb{Z}$ be defined for all $x \in \Sigma^*$ by $h(x) = \prod_{1 \leq i \leq p(|x|)} f(\langle x, i \rangle)$. $h \in \sharp\mathrm{L}$.*

(3) *Let Σ be the input alphabet, $f \in \mathrm{GapL}$ and $p(n)$ be a polynomial. Let $h : \Sigma^* \to \mathbb{Z}$ be defined for all $x \in \Sigma^*$ by $h(x) = \Sigma_{1 \leq i \leq p(|x|)} f(\langle x, i \rangle)$. $h \in \mathrm{GapL}$.*

PROOF. (1) Let $f(\langle x, i \rangle) = acc_M(\langle x, i \rangle)$ for some NL-Turing machine M. Define N to be the NL-Turing machine that on input $x \in \Sigma^*$, first guesses a positive integer i between 1 and $p(|x|)$. If the guessed integer i is greater than $p(|x|)$ then N rejects and stops.

Let us assume that $1 \leq i \leq p(|x|)$. N simulates a computation path of M on the input $\langle x, i \rangle$. N accepts the input $\langle x, i \rangle$ if M accepts and rejects the input otherwise. Then, for every $x \in \Sigma^*$, we get $acc_N(x) = h(x)$.

(2) Let $f(\langle x, i \rangle) = acc_M(\langle x, i \rangle)$ for some NL-Turing machine M. Define N to be the NL-Turing machine that on input $x \in \Sigma^*$ simulates a computation path of M on the input $\langle x, i \rangle$, for all $1 \leq i \leq p(|x|)$, in sequence one after the other. If for some i in the simulation of M on $\langle x, i \rangle$ by N, the computation path rejects then N also rejects and stops. Otherwise N continues to simulate M on $\langle x, i \rangle$. Then for every $x \in \Sigma^*$ we get $acc_N(x) = h(x)$.

(3) Let $f(\langle x, i \rangle) = gap_M(\langle x, i \rangle)$ for some NL-Turing machine M. Define N to be the NL-Turing machine which on input $x \in \Sigma^*$, first guesses a positive integer i between 1 and $p(|x|)$. If the guessed integer i is greater than $p(|x|)$ then N branches exactly once more and accepts along one branch and rejects along the other branch.

Let us assume that $1 \leq i \leq p(|x|)$. N simulates a computation path of M on the input $\langle x, i \rangle$. N accepts the input $\langle x, i \rangle$ if M accepts and rejects the input otherwise. Then, for every $x \in \Sigma^*$, we get $gap_N(x) = h(x)$.

□

THEOREM 2.59. *Let Σ be the input alphabet, $f \in \mathrm{GapL}$ and $p(n)$ be a polynomial. Let $h : \Sigma^* \to \mathbb{Z}$ be defined for all $x \in \Sigma^*$ by $h(x) = \prod_{1 \leq i \leq p(|x|)} f(\langle x, i \rangle)$. $h \in \mathrm{GapL}$.*

PROOF. Let $f = gap_M$ for some NL-Turing machine M. For each $x \in \Sigma^*$ and i such that $1 \leq i \leq p(|x|)$, let $S(x, i)$ denote the set of all computation paths of x on input $\langle x, i \rangle$ and, furthermore, for each $\pi \in S(x, i)$, define $\alpha(x, i, \pi) = 1$ if π is an accepting computation path and it is -1 otherwise. Then, for each $x \in \Sigma^*$, $h(x) = \Sigma_{\pi_1 \in S(x,1)} \cdots \Sigma_{\pi_{p(|x|)} \in S(x,p(|x|))} \alpha(x, 1, \pi_1) \cdots \alpha(x, p(|x|), \pi_{p(|x|)})$.

Define N to be the NL-Turing machine that on input $x \in \Sigma^*$ behaves as follows: N non-deterministically guesses and simulates a path of M on input $\langle x, i\rangle$, for all $1 \leq i \leq p(|x|)$. In the course of doing this, N computes the parity of the number of values of i, $1 \leq i \leq p(|x|)$, such that M on input $\langle x, i\rangle$ rejects. When all the simulations have been completed, N accepts x on its current computation path if and only if this value is even. Note that, for every $x \in \Sigma^*$, on the path of N on input x corresponding to the guesses $(\pi_1, \pi_2, \ldots, \pi_{p(|x|)})$, the product $\alpha(x, 1, \pi_1) \cdots \alpha(x, p(|x|), \pi_{p(|x|)})$ is 1 if and only if N accepts at the end of this computation path and that the product is -1 if and only if N rejects at the end of this computation path. Thus for every $x \in \Sigma^*$, we have $gap_N(x) = h(x)$. Since p is a polynomial and M is a NL-Turing machine, N is also a NL-Turing machine. Thus $h \in \mathrm{GapL}$. □

2.4. The Isolating Lemma

Let $A = \{a_1, a_2, \ldots, a_m\}$ be a set of elements and let $\mathcal{F}$ be a collection of non-empty subsets of A. Let W be a finite number of consecutive integers, which are called as integer weights, and assume that we shall assign integer weights to elements of A using a function $f : A \to W$. Define the weight of a subset S of A, denoted by $f(S)$, to be the sum of the weights of the elements in the subset S.

We say that a weight function *f is good for $\mathcal{F}$* if there is exactly one minimum-weight set in $\mathcal{F}$ with respect to f and we say that *f is bad for $\mathcal{F}$* otherwise.

THEOREM 2.60. **(The Isolating Lemma)** *Let A be a finite set of cardinality m and let $\mathcal{F}$ be a family of non-empty subsets of A. Let $r > 2m$ and let f be a function $f : A \to \{1, \ldots, r\}$ that assigns a value independently and uniformly at random to each element in $a_i \in A$ from the set $\{1, \ldots, r\}$, where $1 \leq i \leq m$. Then,*

$$\Pr_f[f \text{ is good for } \mathcal{F}] \geq \frac{1}{2}.$$

PROOF. Let us define the $MinWeight_f(\mathcal{F}) = \min\{f(S) | S \in \mathcal{F}\}$ and let $MinWeightSet_f(\mathcal{F}) = \{S \in \mathcal{F} | f(S) = MinWeight_f(\mathcal{F})\}$. Recall from the definition of good and bad functions stated above that any weight function f is bad for $\mathcal{F}$ if and only if $|MinWeightSet_f(\mathcal{F})| \geq 2$.

For $x \in A$, we say that minimum-weight sets of $\mathcal{F}$ with respect to f are ambiguous about the inclusion of x if there exists at least one pair of sets $S, S' \in MinWeightSet_f(\mathcal{F})$ such that $x \in (S \setminus S') \cup (S' \setminus S)$. In other words, we say that minimum-weight sets of $\mathcal{F}$ with respect to f are ambiguous about the inclusion of x if there exists at least one pair of sets $S, S' \in MinWeightSet_f(\mathcal{F})$ such that x is in exactly one of S or in S', and that x is not in both S and in S'. Conversely, minimum-weight sets of $\mathcal{F}$ with respect to f are unambiguous about the inclusion of x, if x is either in the intersection of all minimum-weight sets or x is not in any minimum-weight set of $\mathcal{F}$. It is easy to see that f is bad for $\mathcal{F}$ if and only if there exists some $x \in A$ such that minimum weight sets of $\mathcal{F}$ with respect to f are ambiguous about the inclusion of x. Since any weight function f assigns only positive weights to elements of A, it is also easy to see that, if f is

a bad weight function then $\exists S, S' \in MinWeightSet_f(\mathcal{F})$ such that $S \neq S'$ and $|(S \setminus S') \cup (S' \setminus S)| \geq 2$. As a result, if f is a bad weight function, then there exists an element $y \in A$ such that $y \neq x$, $x \in S$, $y \in S'$, and, minimum-weight sets of $\mathcal{F}$ with respect to f are ambiguous about the inclusion of y also. In fact, we will have at least $2 \leq k \leq min(m, \binom{|MinWeightSet_f(\mathcal{F})|}{2})$ such elements in A such that minimum-weight sets of $\mathcal{F}$ with respect to f are ambiguous about the inclusion of each of these k elements.

Let us assume that a function f is bad for $\mathcal{F}$. Therefore $|MinWeightSet_f(\mathcal{F})| \geq 2$, and minimum weight sets of $\mathcal{F}$ with respect to f are ambiguous about the inclusion of x, for some $x \in A$. Let us fix this $x \in A$, and let us not bother about the number of ambiguous elements of A with respect to f. Let us define $f' : A \to \{1, \ldots, r\}$ suitably so that $MinWeight_{f'}(\mathcal{F}) = MinWeight_f(\mathcal{F})$. In defining f', we first ensure that $MinWeightSet_{f'}(\mathcal{F}) = \{S \in MinWeightSet_f(\mathcal{F})|x \notin S\}$.

Let $B = \cup_{\{T|T \text{ does not contain } x,\ T \in MinWeightSet_f(\mathcal{F})\}} T$. It is easy to see that $|B| \geq 2$. First, determine if there exists $z' \notin B$ such that $z' \neq x$. If there does not exist any such z' then we arbitrarily delete a pair of elements from B, say z_1, z_2, and we do not assign any weight to these two elements immediately as we do for other elements in B: if $z \in B$ then let $f'(z) = f(z)$. We define f' for elements $z'' \notin B$, where $z'' \neq x$, or the pair of elements z_1, z_2 that might exist as mentioned above such that $f'(z'') \neq f(z'')$, or $\{f'(z_1) \neq f(z_1)$ and $f'(z_2) \neq f(z_2)\}$. This is to ensure that we obtain a function f' which is not f itself. Before we assign weights for x in f', let us consider $\mathcal{F}'$ which is the collection of subsets of A in $\mathcal{F}$ such that if x is in a set $S \in \mathcal{F}$ then we delete x from S. It is important to note that by defining f' in this manner we obtain $MinWeight_{f'}(\mathcal{F}') = MinWeight_f(\mathcal{F}') = MinWeight_f(\mathcal{F})$. Let us finally assign a weight $f'(x) \neq f(x)$ for x and and let us consider $\mathcal{F}$.

Claim. Minimum weight sets of $\mathcal{F}$ with respect to f' are unambiguous about the inclusion of x.

Proof of Claim. To prove this, let $\delta(x) = f'(x) - f(x)$. Suppose that $\delta(x) > 0$. For any $S \subseteq A$, if $x \in S$ then $f'(S) = f(S) + \delta(x)$ and $f'(S) = f(S)$ otherwise. This implies that $MinWeight_f(\mathcal{F}) = MinWeight_{f'}(\mathcal{F})$ and $MinWeightSet_{f'}(\mathcal{F}) = \{S \in MinWeightSet_f(\mathcal{F})|x \notin S\}$. Therefore, if $\delta(x) > 0$ then there does not exist any minimum-weight set of $\mathcal{F}$ with respect to f' that contains x. Next, suppose that $\delta(x) < 0$. Then, for all $S \in \mathcal{F}$, we have $f'(S) = f(S) - |\delta(x)|$ if $x \in S$ and $f'(S) = f(S)$ otherwise. This implies that $MinWeight_f(\mathcal{F}) = MinWeight_{f'}(\mathcal{F}) - |\delta(x)|$ and $MinWeightSet_{f'}(\mathcal{F}) = \{S \in MinWeightSet_f(\mathcal{F})|x \in S\}$. Therefore, if $\delta(x) < 0$ then all minimum-weight sets of $\mathcal{F}$ with respect to f' contain x. This proves our claim.

Therefore, if $f'(x) = f(x)$ then f' is a bad weight function and minimum weight sets of $\mathcal{F}$ with respect to f' are ambiguous about the inclusion of x. Given $\mathcal{F}$, we need not have every $x \in A$ as a possible candidate for being ambiguous since some x can be in every subset of A in $\mathcal{F}$. However if we have obtained one element $x \in A$ as ambiguous with respect to a bad weight function f, then the

number of bad weight functions f' that we can obtain from f, for each of which x is ambiguous, is $\leq r^{m-1}$. Since there can be at most m choices for x, the number of bad weight functions that can exist is at most mr^{m-1}. As a result, the proportion of weight functions f such that f is bad for $\mathcal{F}$ is $\leq \frac{mr^{m-1}}{r^m} < \frac{1}{2}$. Therefore, the proportion of weight functions f that are good for $\mathcal{F}$ is $\geq \frac{1}{2}$ and this proves our theorem. □

COROLLARY 2.61. *Let $A = \{a_1, \ldots, a_m\}$ be a finite set of cardinality m and let $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$ be a collection of families of non-empty subsets of A, where $n < m^k$ for some $k \geq 0$. Let $r > 2nm$ and let f be a function $f : A \to \{1, \ldots, r\}$, which assigns a value independently and uniformly at random to each element $a_i \in A$ from the set $\{1, \ldots, r\}$, where $1 \leq i \leq m$. Then,*

$$\Pr_f[f \text{ is good for each } \mathcal{F}_i] \geq \frac{1}{2},$$

where $1 \leq i \leq n$.

PROOF. If we have only one family $\mathcal{F}$, then we have shown in Theorem 2.60 that

$$\Pr_f[f \text{ is bad for } \mathcal{F}] \leq \left(\frac{mr^{m-1}}{r^m}\right).$$

If there are n families $\mathcal{F}_1, \ldots, \mathcal{F}_n$, then

$$\Pr_f[f \text{ is bad for at least one } \mathcal{F}_j, \text{ where } 1 \leq j \leq n] \leq \frac{nmr^{m-1}}{r^m} < \frac{1}{2}.$$

As a result, we get

$$\Pr_f[f \text{ is good for all } \mathcal{F}_j, \text{ where } 1 \leq j \leq n] > \left(1 - \frac{nmr^{m-1}}{r^m}\right) \geq \frac{1}{2}.$$

□

COROLLARY 2.62. *Let $A = \{a_1, \ldots, a_m\}$ be a finite set of cardinality m and let $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$ be a collection of families of non-empty subsets of A, where $n < m^k$, for some $k \geq 0$. Let $r > 4nm$. There is a collection of weight functions $f_1, \ldots f_m$ such that, each $f_i : A \to \{1, \ldots, r\}$ and for every collection $\mathcal{F}_1, \ldots, \mathcal{F}_n$ of families of non-empty subsets of A, there exists some f_i such that f_i is good for all $\mathcal{F}_j$, where $1 \leq i \leq m$ and $1 \leq j \leq n$.*

PROOF. Since we have chosen $r > 4nm$, it follows from Corollary 2.61 that

$$\Pr_f[f \text{ is good } \forall \mathcal{F}_j, \text{ where } 1 \leq j \leq n] \geq \left(\frac{3}{4}\right).$$

We first note that there are 2^m possible subsets of A. Among these subsets, let us consider $S \subseteq A$ such that $a_i \in S$ if and only if a_i is a member of at least one subset of A in some family $\mathcal{F}_j$, where $1 \leq i \leq m$ and $1 \leq j \leq n$. As a result considering

S and the given a collection of families $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$ of non-empty subsets of A, if $f_1, \ldots f_m$ are random weight functions then,

$$\Pr_f[\nexists\, 1 \leq j \leq n :\ (f_j \textit{ is not good } \forall \mathcal{F}_j,\ \textit{where } 1 \leq j \leq n)] < \left(\frac{1}{4^m}\right).$$

Since there 2^m possible subsets of A, given any $S \subseteq A$ from whose elements we get families $\mathcal{F}_1, \ldots, \mathcal{F}_n$ we get

$$\Pr_f[\nexists\, 1 \leq j \leq n :\ (f_j \textit{ is not good } \forall \mathcal{F}_j,\ \textit{where } 1 \leq j \leq n)] < \left(\frac{2^m}{4^m}\right) < 1.$$

Therefore, there exists a collection of weight functions $f_1, \ldots, f_m$ such that for every collection of subsets $\mathcal{F}_1, \ldots, \mathcal{F}_n$ of A, there exists some f_i such that f_i is good for all $\mathcal{F}_j$, where $1 \leq i \leq m$ and $1 \leq j \leq n$. □

COROLLARY 2.63. *Let A be a finite set of cardinality m and let $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$ be a collection of families of non-empty subsets of A, where $n < m^k$ for some $k \geq 0$. Also let $r > \max(m^4, n^4)$. There exists a function $f : A \to \{1, \ldots, r\}$ such that for all families $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$ of non-empty subsets of A, we have f is good for every $\mathcal{F}_i$, where $n < m^k$, for some $k \geq 0$, and $1 \leq i \leq n$.*

PROOF. Form a $0, 1$ matrix M of dimension $n \times r^m$ with n rows and r^m columns, where the i^{th} row of M corresponds to the family $\mathcal{F}_i$ and the j^{th} column corresponds to the function f_j in the lexicographically increasing order, for $1 \leq i \leq n$ and $1 \leq j \leq r^m$. (Here we assume that every function f is considered as a vector, and given two functions f and g we say that f is lexicographically smaller than g, denoted by $f \prec g$, if f is less than g when compared componentwise as vectors). We put $M_{ij} = 1$ if the function f_j is good for the family $\mathcal{F}_i$ and $M_{ij} = 0$ otherwise. For a given $1 \leq i \leq n$, by the choice of r and from the proof of Theorem 2.60, it follows that

$$\Pr_f\left[f \textit{ is good for } \mathcal{F}_i\right] > \left(1 - \frac{1}{m^k}\right),$$

where f is a function chosen independently and uniformly at random. For sufficiently large m, it is easy to see that there exists a column in M that contains only 1. The function corresponding to this column satisfies the condition stated in this theorem. □

2.4.1. Min-unique graphs and Unambiguous Logarithmic space, UL.

DEFINITION 2.64. A min-unique graph is a weighted directed graph with positive weights associated with each edge where for every pair of vertices u, v, if there is a path from u to v, then there exists a unique minimum weight path from u to v. Here, the weight of a path is the sum of the weights on its edges. Any such weight function on the edges of the min-unique graph G is defined as a min-unique weight function. If the maximum of the positive integer weights assigned by the weight function are less than or equal to a polynomial in the size of the directed graph then we say that the weight function is polynomially bounded.

Let f be a function that assigns positive weights to the edges of the directed graph. If only for some vertex s the minimum weight directed path from s to every other vertex reachable from s is unique, then the weight function f is min-unique with respect to s.

We define the complexity class unambiguous logarithmic space, UL.

DEFINITION 2.65. Let Σ be the input alphabet. We say that $L \subseteq \Sigma^*$ is a language in the complexity class UL if there exists a function $f \in \sharp\mathrm{L}$ such that for any input $x \in \Sigma^*$ we have $f(x) = 1$ if $x \in L$ and $f(x) = 0$ if $x \notin L$.

We now define co-UL, based on the definition of co-$\mathcal{C}$ from Definition 2.2 where $\mathcal{C}$ is a complexity class.

DEFINITION 2.66. Let Σ be the input alphabet. For a language $L \in \Sigma^*$, the complement of L is $\overline{L} = \Sigma^* - L$. We define the complexity class co-UL $= \{\overline{L} \subseteq \Sigma^* | L \in \mathrm{UL}\}$.

DEFINITION 2.67. Let Σ be the input alphabet. We say that $L \in \mathrm{UL} \cap$ co-UL if $L \in \mathrm{UL}$ and $L \in$ co-UL.

THEOREM 2.68. *Let $\mathcal{G}$ be a class of graphs, where each graph in $\mathcal{G}$ is given in terms of its adjacency matrix, and let $H = (V, E) \in \mathcal{G}$. If there is a polynomially bounded logarithmic space computable function f that on input H outputs a weighted graph $f(H)$ so that*

(1) *$f(H)$ is min-unique, and*
(2) *H has an st-path if and only if $f(H)$ has an st-path*

then the st-connectivity problem for $\mathcal{G}$ is in $\mathrm{UL} \cap$ co-UL.

PROOF. It suffices to give a $\mathrm{UL} \cap$ co-UL algorithm for the reduced graph. For $H \in \mathcal{G}$, let $G = f(H)$ be a directed graph with a min-unique weight function w on its edges. We first construct an unweighted graph G' from G by replacing every edge e in G with a directed path of length $w(e)$. It is easy to see that st-connectivity is preserved. That is, there is an st-path in G if and only if there is one in G'. Since G is a min-unique graph, it is straightforward to argue that the shortest path between any two vertices in G' is unique. Let us call this directed graph G' as ***unweighted min-unique graph***.

Let c_k and Σ_k denote the number of vertices which are at a distance at most k from s and the sum of the lengths of the shortest path to each of them, respectively. Let $d(v)$ denote the length of the shortest path from s to v. If no such path exists, then $d(v) = |V| + 1$. We have,

$$\Sigma_k = \sum_{\{v \in V \,|\, d(v) \leq k\}} d(v).$$

We first give an unambiguous routine (Algorithm 5) to evaluate the predicate "$d(v) \leq k$" when given the values of c_k and Σ_k. The algorithm will output the correct value of the predicate (True/False) on a unique path and outputs "?" on the rest of the paths.

Algorithm 5 Shortest-Path: Determining whether $d(v) \leq k$ or not.
Input: (G, v, k, c_k, Σ_k), where $G = (V, E)$ is a unweighted min-unique graph given in terms of its adjacency matrix, $v \in V$ and $k \in \mathbb{N}$.
Output: True if $d(v) \leq k$. Otherwise False or ?.
Complexity: UL.

```
1: Initialize count ← 0; sum ← 0; path.to.v ←False
2: for each x ∈ V do
3:     Non-deterministically guess if d(x) ≤ k
4:     if guess is Yes then
5:         Non-deterministically guess a path of length l ≤ k from s to x
6:         if guess is correct then
7:             Set count ← count + 1
8:             Set sum ← sum + l
9:             if x = v then
10:                Set path.to.v ←True
11:            end if
12:        else
13:            return "?"
14:        end if
15:    end if
16: end for
17: if count = c_k and sum = Σ_k then
18:    return path.to.v
19: else
20:    return "?"
21: end if
```

We will argue that Algorithm 5 is unambiguous.

(1) If Algorithm 5 incorrectly guesses that $d(x) > k$ for some vertex x then $count < c_k$ and so it returns "?" in line 20. Thus, consider the computation paths that correctly guess the set $\{x | d(x) \leq k\}$.
(2) If at any point the algorithm incorrectly guesses the length l of the shortest path to x, then one of the following two cases occur.
   (a) If $d(x) > l$ then no path from s to x would be found and the algorithm returns "?" in line 13.
   (b) If $d(x) < l$ then the variable *sum* would be incremented by a value greater than $d(x)$ and thus *sum* would be greater than Σ_k causing the algorithm to return "?" in line 20.

Thus there will remain only one computation path where all the guesses are correct and the algorithm will output the correct value of the predicate on this unique path. Finally, we note that Algorithm 5 is easily seen to be computable in logarithmic space.

Next, we describe an unambiguous procedure (Algorithm 6) that computes c_k and Σ_k given c_{k-1} and Σ_{k-1}.

Algorithm 6 Computing c_k and Σ_k.
Input: $(G, k, c_{k-1}, \Sigma_{k-1})$, where $G = (V, E)$ is a unweighted min-unique graph given in terms of its adjacency matrix, and $k \in \mathbb{N}$.
Output: c_k, Σ_k.
Complexity: UL.

```
1: Initialize c_k ← c_{k-1} and Σ_k ← Σ_{k-1}
2: for each v ∈ V do
3:     if ¬(d(v) ≤ k − 1) then
4:         for each x such that (x, v) ∈ E do
5:             if d(x) ≤ k − 1 then
6:                 Set c_k ← c_k + 1
7:                 Set Σ_k ← Σ_k + k
8:             end if
9:         end for
10:    end if
11: end for
12: return c_k and Σ_k
```

Algorithm 6 uses Algorithm 5 as a subroutine. Other than making function calls to Algorithm 5, this routine is deterministic, and so it follows that Algorithm 6 is also unambiguous. We will argue that Algorithm 6 computes c_k and Σ_k. The subgraph consisting only of s ($d(x) \leq 0$) is trivially min-unique and $c_0 = 1$ and $\Sigma_0 = 0$. Inductively, it is easy to see that

$$c_k = c_{k-1} + |\{v | d(v) = k\}|,$$
$$\Sigma_k = \Sigma_{k-1} + k \times |\{v | d(v) = k\}|.$$

In addition, $d(v) = k$ if and only if there exists $(x, v) \in E$ such that $d(x) \leq k - 1$ and $\neg(d(v) \leq k - 1)$. Both of these predicates can be computed using Algorithm 5. Combining these facts, we see that Algorithm 6 computes c_k and Σ_k given c_{k-1} and Σ_{k-1}.

Algorithm 7 Determining if there exists a path from s to t in a min-unique graph G.
Input: A min-unique graph $G = (V, E)$ given in terms of its adjacency matrix, and vertices $s, t \in V$.
Output: True if there exists a directed path from s to t in G. Otherwise False.
Complexity: UL.

```
1: We obtain the unweighted min-unique graph G' from G as described in the
   beginning of this proof.
2: Initialize c_0 ← 1, Σ_0 ← 0, k ← 0
3: for k ← 1, ... n do
4:     Compute c_k and Σ_k by invoking Algorithm 6 on (G', k, c_{k-1}, Σ_{k-1})
5: end for
6: Invoke Algorithm 5 on (G', t, n, c_n, Σ_n) and return its value
```

As a final step, we give the main routine that invokes Algorithm 6 to check for st-connectivity in a min-unique graph. Since there is an st-path if and only if $d(t) \leq n$, it suffices to compute c_n and Σ_n and invoke Algorithm 5 on the input (G, t, n, c_n, Σ_n). This procedure is presented as Algorithm 7. To ensure that the algorithm runs in logarithmic space, we do not store all intermediate values for c_k, Σ_k. Instead, we only keep the most recently computed values and re-use space. Similar to Algorithm 6, this procedure is deterministic and so the entire routine is unambiguous. Thus reachability in min-unique graphs can in fact be decided in UL $\cap$ co-UL. □

NOTE 4. *We recall Definition 2.64 and note that, in the proof of Theorem 2.68, it is not necessary to have a weight function that is min-unique such that for every pair of vertices u and v there exists a directed path from u to v if and only if there exists a directed path from u to v which has unique minimum weight. Instead, it is sufficient that the min-uniqueness is with respect to only the vertex s. In other words, it is sufficient that the function f outputs $f(H)$ such that there exists a directed path from s to any vertex v in H if and only if there is a unique minimum weight directed path from s to v in $f(H)$.*

DEFINITION 2.69. Let Σ be the input alphabet and let Γ be the output alphabet. We define FNL to be the complexity class of all functions $f : \Sigma^* \to \Gamma^*$ such that for any given input string $x \in \Sigma^*$ there exists a NL-Turing machine M which outputs $f(x)$ at the end of all of its accepting computation paths.

DEFINITION 2.70. Let Σ be the input alphabet and let Γ be the output alphabet. We define FUL to be the complexity class of all functions $f : \Sigma^* \to \Gamma^*$ such that for any given input string $x \in \Sigma^*$ there exists a NL-Turing machine M which has at most one accepting computation path and M outputs $f(x)$ at the end of its unique accepting computation path.

Let Σ be the input alphabet and Γ be the output alphabet. We say that a function $f : \Sigma \to \Gamma^*$ is UL computable if $f \in$ FUL.

THEOREM 2.71. NL = UL *if and only if there is a polynomially-bounded* UL *computable weight function f so that for any directed acyclic graph G, we have $f(G)$ is min-unique with respect to s.*

PROOF. Assume NL = UL. Therefore, given a directed graph $G = (V, E)$ in terms of its adjacency matrix, and vertices $s, t \in V$ as input, the problem of determining if there exists a directed path from s to t in G is in UL. Also, we know from Theorem 2.18 that NL = co-NL. So UL = co-UL.

Claim: The language $A = \{(G, s, t, k) | \exists$ a path from s to t of length $\leq k\}$ is in UL.

Proof of Claim. We reduce (G, s, t) to an instance of SLDAG in $O(\log n)$ space as in Theorem 2.10. Using a deterministic $O(\log n)$ space bounded Turing machine We then query the DSTCON oracle if there exists a directed path from s to the copy of the vertex t in every layer starting from the second layer of the instance of SLDAG. We accept the input if there is a directed path from s to t in layer i where

$i \leq k$. Otherwise we reject. Our claim now follows because of Theorem 2.19 and our assumption that NL = UL.

We now compute a subgraph of G which is a directed tree rooted at s such that there exists a directed path from s to a vertex v in G if and only if there exists a directed path from s to v in the tree. We say that a vertex v is in level k if the minimum length of any directed path from s to v is of length $k > 0$. A directed edge (u, v) is in the tree if for some $k > 0$,

(1) v is in level k, and
(2) u is the lexicographically first vertex in level $k-1$ so that (u, v) is a directed edge.

It is clear that this is indeed a well-defined tree and to decide if an edge $e = (u, v)$ is in this tree, we submit the following query to A: is it true that $\forall w < u,\ l \leq (k-1)\ [(\bigwedge(G, s, w, l) \notin A) \wedge (w, v) \in A]$. If an edge is in the tree we assign the weight 1 to it. For the rest of the edges we assign weight n^2. It is clear that the shortest path from s to any vertex with respect to this weight function is min-unique. It is also easy to see that this weight function is computable in $\mathrm{L}^A \subseteq$ UL.

Conversely, by following the proof of Theorem 2.68 under the assumption that the complexity of computing weighted graph $f(H)$ using the weight function f is UL and the observation in Note 4 we get NL = UL. □

2.4.2. Some more applications of Isolating Lemma and non-deterministic counting.

DEFINITION 2.72. Let $\Sigma = \{0, 1\}$ be the input alphabet and let $\mathcal{C}$ be a complexity class. We define $\mathcal{C}/poly$ as the complexity class of all languages $L \subseteq \Sigma^*$ for which there exists a sequence of "advice strings" $\{\alpha(n) | n \in \mathbb{N}\}$, where $|\alpha(n)| \leq p(n)$ for some polynomial p, and a language $L' \in \mathcal{C}$ such that $x \in L$ if and only if $(x, \alpha(|x|)) \in L'$, for every $x \in \Sigma^*$.

THEOREM 2.73. (UL ∩ co-UL)/poly ⊆ NL/poly.

PROOF. It is easy to see from Definition 2.65 and Definition 2.1 that UL ⊆ NL. Also we know from the Immerman-Szelepcsenyi Theorem (also see Theorem 2.18) that NL = co-NL and so (UL ∩ co-UL) ⊆ NL. Using Definition 2.72, it is now trivial to show that (UL ∩ co-UL)/poly ⊆ NL/poly. □

THEOREM 2.74. NL/poly ⊆ (UL ∩ co-UL)/poly.

PROOF. Using Definition 2.72, we infer that it is sufficient to show that NL ⊆ (UL ∩ co-UL)/poly. We know from Theorem 2.7 that, DSTCON is complete for NL under $\leq^{\mathrm{L}}_m$. Therefore to prove our result, it is sufficient to show that DSTCON ∈ (UL ∩ co-UL)/poly.

Let us now consider an input instance (G, s, t) of DSTCON, where $G = (V, E)$ is a directed graph and $n = |V|$. We assume that G is given as input in terms of its $n \times n$ adjacency matrix M. Also vertices s and t of G are marked using two special symbols of the input alphabet Σ. As a result the size of the input (G, s, t) is $(n^2 + 2)$ symbols. We also assume the following ordering of edges in

G: if $M(i,j) = 1$ then it is the $((i-1) \times |V| + j)^{th}$ edge of G. Otherwise if $M(i,j) = 0$ then the $((i-1) \times |V| + j)^{th}$ edge does not exist in G. In total there can be at most n^2 edges in G.

Let us define a family $\mathcal{F}_{u,v}$ of subsets of E such that if $S \subseteq E$ and $S \in \mathcal{F}_{u,v}$ then edges in S form a directed path from u to v in G. Number of such families $\mathcal{F}_{u,v}$ is at most $2 \times \binom{n}{2}$. Let $r > 4n^4$ and $A = \{1, \ldots, (n^2+2)\}$. It follows from Corollary 2.62 that there is a collection of weight functions $f_1, \ldots, f_{n^2+2}$ such that each $f_i : A \to \{1, \ldots, r\}$, and for every collection $\mathcal{F}_{u,v}$ of families of non-empty subsets of E, $\exists$ some f_i such that f_i is good for all $\mathcal{F}_{u,v}$, where $1 \leq i \leq (n^2+2)$ and $u, v \in V$. In other words, the set of weight functions $f_1, \ldots, f_{n^2+2}$ which we obtain using Corollary 2.62 is such that, there exists some f_i using which there exists a directed path from u to v in $f_i(G)$ of unique minimum weight between every pair of vertices u, v for which there exists a directed path from u to v in G, where the weight of a directed path is the sum of the weights of the edges in the directed path, $1 \leq i \leq (n^2+2)$ and $f_i(G)$ is the weighted directed graph obtained after assigning weights to edges of G according to f_i.

Given f_i, we replace every edge (u,v) that exists in G with a directed path of length $f_i((u,v))$, where $1 \leq i \leq (n^2+2)$. Let the resulting directed graph be G_i. We assume that vertices in G which are in G_i are numbered from 1 to $|V|$ and newly added vertices get numbering greater than $|V|$ in G_i. As a result we obtain (n^2+2) directed graphs $G_1, \ldots G_{n^2+2}$ such that

- the directed graph G_i has a directed path from a vertex u to another vertex v if and only if there exists a directed path from u to v in G_i, and
- there exists some $1 \leq i \leq (n^2+2)$ such that G_i is min-unique.

It is easy to see that, given the directed graph $G = (V, E)$ as input, if we are given (n^2+2) weight functions then we can output the (n^2+2) directed graphs $G_1, \ldots, G_{n^2+2}$ in $O(\log n)$ space, where $n = |V|$.

A useful observation is that if G_i is min-unique, then we can use Algorithm 5 (Shortest-Path) on input $(G_i, v, k, c_k, \Sigma_k)$ to decide whether $d(v) \leq k$ or not in UL. Here we assume that correct values of c_k and Σ_k are provided. We complete the proof of this theorem by giving two algorithms. The proof of correctness and the complexity analysis of following algorithms is similar to Theorem 2.68.

The following algorithm computes c_k and Σ_k if the input directed graph is min-unique, or it outputs that the input directed graph is a $BAD.GRAPH$ and therefore it is not min-unique.

Algorithm 8 Computing c_k and Σ_k for directed graphs.
Input: $(G, k, c_{k-1}, \Sigma_{k-1})$, where $G = (V, E)$ is a directed graph given in terms of its adjacency matrix, and $k \in \mathbb{N}$.
Output: c_k, Σ_k and the flag $BAD.GRAPH$.
Complexity: UL.

1: Initialize $c_k \leftarrow c_{k-1}$ and $\Sigma_k \leftarrow \Sigma_{k-1}$
2: **for** each $v \in V$ **do**
3: **if** $\neg(d(v) \leq k-1)$ **then**

Algorithm 8 Computing c_k and Σ_k for directed graphs (continued).

```
4:          for each x such that (x, v) ∈ E do
5:              if d(x) ≤ k − 1 then
6:                  Set c_k ← c_k + 1
7:                  Set Σ_k ← Σ_k + k
8:                  for x' ≠ x do
9:                      if ((x', v) is an edge and d(x') ≤ (k − 1)) then
10:                         BAD.GRAPH ← True
11:                     end if
12:                 end for
13:             end if
14:         end for
15:     end if
16: end for
17: return c_k and Σ_k and BAD.GRAPH
```

Algorithm 9 Determining if there exists a path from s to t in a directed graph G which is assumed to be min-unique.
Input: A directed graph $G = (V, E)$ given in terms of its adjacency matrix which is assumed to be min-unique, and vertices $s, t \in V$.
Output: True if there exists a directed path from s to t in G. Otherwise False. Either G is not min-unique or there does not exist any directed path from s to t in G.
Complexity: UL.

```
1: Initialize BAD.GRAPH = False, c_0 ← 1, Σ_0 ← 0, k ← 0
2: repeat
3:     k ← (k + 1)
4:     Compute c_k and Σ_k by invoking Algorithm 8 on (G, k, c_{k−1}, Σ_{k−1})
5: until (c_{k−1} = c_k or BAD.GRAPH = True)
6: if BAD.GRAPH = False then
7:     there is a directed path from s to t in G if and only if d(t) ≤ k
8: end if
```

Assume that we are given an input instance (G, s, t) of DSTCON. Let functions $f_i : A \to \{1, \ldots, r\}$ be as stated above, which we have obtained from Corollary 2.62 be given as the advice string, where $1 \leq i \leq (n^2 + 2)$. Each weight function f_i is a vector of weights $\vec{\mathbf{f_i}} = (w_1, w_2, \ldots, w_l)$, where $l = (n^2 + 2)$ and $1 \leq i \leq (n^2 + 2)$. Clearly the length of each $\vec{\mathbf{f_i}}$ is $(n^2 + 2)^k$ (where $k > 0$ is a constant), which is a polynomial in the size of the input (G, s, t). Therefore the length of the advice string is also a polynomial in the size of the input.

Also it is possible to use this advice string to assign weights to edges of G in $O(\log n)$ space to obtain $(n^2 + 2)$ weighted directed graphs $G_1, \ldots, G_{n^2+2}$. It is clear that there exists i such that G_i is min-unique, where $1 \leq i \leq (n^2 + 2)$. Using

Algorithm 9 given above, it follows that we can decide if there exists a directed path from s to t in G in $\mathrm{UL}/\mathrm{poly}$. Since we have shown in Theorem 2.18 that $\mathrm{NL} = \text{co-NL}$, it follows that we can in fact decide if there exists a directed path from s to t in an input instance of DSTCON in $(\mathrm{UL} \cap \text{co-UL})\mathrm{poly}$. As a result it follows that $\text{DSTCON} \in (\mathrm{UL} \cap \text{co-UL})/\mathrm{poly}$ which proves our result.

□

COROLLARY 2.75. $\mathrm{NL}/\mathrm{poly} = (\mathrm{UL} \cap \text{co-UL})/\mathrm{poly}$.

2.5. A combinatorial property of ♯L

2.5.1. Some more results on directed st-connectivity.

LEMMA 2.76. *Let $X = \{1, \ldots, n\}$ be a set and let $\mathcal{F} \subseteq 2^X$ such that $|\mathcal{F}| \leq p(n)$ for a polynomial $p(n)$. Let $r > (n+1)^2 p^2(n)$ be a prime number and for each $1 \leq i \leq r$ and $j \in X$ define the weight function $w_i : [n] \longrightarrow \mathbb{Z}_r$ as $w_i(j) = (i^j \text{mod } r)$. Further for each subset $Y \subseteq X$ define*

$$w_i(Y) = \Sigma_{j \in Y} w_i(j) (\text{mod } r).$$

There exists a weight function w_m such that $w_m(Y) \neq w_m(Y') (\text{mod } r)$ for any two distinct $Y, Y' \in \mathcal{F}$.

PROOF. For any $1 \leq m \leq r$ and $Y \in \mathcal{F}$, we can interpret $w_m(Y)$ as the value of the polynomial $q_Y(z) = \Sigma_{j \in Y} z^j$ at the point $z = m$ over the field $\mathbb{Z}_r$. For $Y \neq Y'$, notice that the polynomials $q_Y(z)$ and $q_{Y'}(z)$ are distinct and their degrees are at most n. Hence, $q_Y(z)$ and $q_{Y'}(z)$ can be equal for at most n values of z in the field $\mathbb{Z}_r$. Equivalently, if $Y \neq Y'$ then $w_i(Y) = w_i(Y')$ for at most n weight functions w_i. Since there are $\binom{|\mathcal{F}|}{2}$ pairs of distinct sets in $\mathcal{F}$, it follows that there are at most $\binom{|\mathcal{F}|}{2} \cdot n < n \cdot p^2(n)$ weight functions w_i for which $w_i(Y) = w_i(Y')$ for some pair of sets $Y, Y' \in \mathcal{F}$. Since $r > n \cdot p^2(n)$, there is a weight function as claimed by the lemma. □

THEOREM 2.77. *Let (G, s, t) be an input instance of* SLDAGSTCON *and let the number of directed paths from s to t in G be at most $p(n)$ for some polynomial $p(n)$, where $G = (V, E) \in$ SLDAG and $n = |E|$. Also let $E = \{1, \ldots, n\}$ and let $r > (n+1)^2 p^2(n)$ be a prime number and for each $1 \leq i \leq r$ and $j \in E$ define the weight function $w_i : E \longrightarrow \mathbb{Z}_r$ as $w_i(j) = (i^j \text{mod } r)$. Further for each subset $Y \subseteq E$ define*

$$w_i(Y) = \Sigma_{j \in Y} w_i(j) (\text{mod } r).$$

Now let $\mathcal{F} \subseteq 2^X$ such that if $X \in \mathcal{F}$ then edges in X form a directed path from s to t in G. It is then possible to determine a weight function w_m such that $w_m(X) \neq w_m(X') (\text{mod } r)$ for any two distinct $X, X' \in \mathcal{F}$, and the number of directed paths from s to t in G in FNL.

PROOF. We iteratively start with the first weight function and first replace each edge in G with weight w by a directed path of length w. Let the resulting directed graph obtained be G'. It is easy to note that the number of directed paths from s to t in G is equal to the number of directed paths from s to t in G'. Since it is shown

in Theorem 2.10 that SLDAGSTCON is NL-complete under logspace many-one reductions it is possible to obtain an input instance (G'', s'', t'') of SLDAGSTCON when we are given (G', s, t) as input. Upon following the proof of Theorem 2.10 we infer that the number of directed paths from s to t in G' is equal to the number of directed paths from s'' to t'' in G''. Let us initialize a counter c to 0. We consider the simple layered directed acyclic graph structure of G'' and vertices which are copies of t in all the polynomially many layers of G''. We query the NL oracle if there exists a directed path in G'' from s'' to the copy of the vertex t in each of the layers of G''. If the oracle returns "yes" then we increment the counter c by 1. For a given weight function we can compute c in $O(\log |G|)$ space. Using Lemma 2.76, it follows that there exists a weight function w_m such that $w_m(X) \neq w_m(X')(\text{mod } r)$ for any two distinct $X, X' \in \mathcal{F}$, where $1 \leq m \leq r$. Clearly any such weight function will result in the maximum value for the counter c. As a result by storing c for successive weight functions and updating it by comparison we can find the weight function w_m also. Note that the maximum value of c is the number of directed paths from s to t in G itself. Since $|G''|$ is a polynomial in $|G|$ it follows that we can find the weight function w_m in FL^{NL} which is FNL due to Theorem 2.19. □

THEOREM 2.78. *Let $G = (V, E)$ be a directed graph given in terms of its adjacency matrix as input, and let $s, t \in V$. Also let p be a positive integer whose unary representation can be computed by a deterministic $O(\log |G|)$ space bounded Turing machine. Then we can determine if the number of directed paths from s to t in G is at least $(p + 1)$ in* FNL. *Otherwise if the number of directed paths from s to t in G is lesser than $(p + 1)$ then the* FNL *machine outputs the number of directed paths from s to t in G.*

PROOF. Due to Theorem 2.10 we know that SLDAGSTCON is NL-complete under logspace many-one reductions. We therefore follow Theorem 2.10 and obtain $H \in$ SLDAG and vertices $s', t' \in V(H)$ from G. Upon following the proof of Theorem 2.10 we infer that the number of directed paths from s to t in G is equal to the number of directed paths from s' to t' in H. Let us consider the subgraph of H induced by vertices that are in at least one directed path from s' to t' in H. Let this subgraph be H'. We use induction on the number of layers $\lambda \geq 2$ in H' and consider subgraphs of H' formed by vertices in the first λ layers of H' such that the number of directed paths from s' to vertices in layer λ of H' is at most $(p + 1)$ or a polynomial in $|G|$. Note that it is easy to compute this upper bound on the number of directed paths using $O(\log |G|)$ space. We now once again use Theorem 2.10 and Theorem 2.77 to compute the number of directed paths from s' to all the vertices in layer λ in H'. If the number of directed paths is greater than or equal to $(p + 1)$ then we move to the accepting configuration, output $(p + 1)$ and stop. Otherwise finally we would have computed the number of directed paths from s' to t' in H which is lesser than $(p + 1)$. We then move to the accepting configuration, output this value and stop. In all stages of this proof, we use a deterministic $O(\log |G|)$ space bounded Turing machine with access to the FNL oracle.

The deterministic $O(\log |G|)$ space bounded Turing machine submits queries deterministically to the FNL oracle. In the intermediary stages, the reductions are done by submitting queries deterministically to the FNL oracle and after reading the reply given by the oracle on the oracle tape. As a result given the input G it is possible to determine if the number of directed paths from s to t is at least $(p+1)$ is in $\mathrm{FL}^{\mathrm{FNL}}$ which is FNL due to Theorem 2.19. □

COROLLARY 2.79. *Verifying if there are polynomially many accepting computation paths for a* NL*-Turing machine on a given input is in* FNL.

PROOF. Let M be a NL-Turing machine and let g be the canonical logspace many-one reduction from $L(M)$ to SLDAGSTCON obtained by using the seminal result that the st-connectivity problem for directed graphs is complete for NL under logspace many-one reductions and Theorem 2.10. We note that if x is the input string then $g(x) = (G, s, t)$ and the number of accepting computation paths of M on x is equal to the number of directed paths from s to t in G. We now use Theorem 2.78 to complete the proof. □

COROLLARY 2.80. *Assume that* NL = UL. *Let $G = (V, E)$ be a directed graph given in terms of its adjacency matrix as input, and let $s, t \in V$. Also let p be a positive integer whose unary representation can be computed by a deterministic $O(\log |G|)$ space bounded Turing machine. Then we can determine if the number of directed paths from s to t in G is at least $(p+1)$ in* FUL. *Otherwise if the number of directed paths from s to t in G is lesser than $(p+1)$ then the* FUL *machine outputs the number of directed paths from s to t in G.*

NOTE 5. *The number of accepting computation paths of a* NL*-Turing machine is not altered if it simulates a* NL*-Turing machine of a language $L \in$ UL during intermediate stages to verify if some input string is in L.*

2.5.2. Choosing at most polynomially many number of computation paths from the computation tree of a NL-Turing machine. We refer to Definition A.3 and Theorem A.4 in the Appendix A for $\binom{n}{k}$, where $n, k \in \mathbb{Z}^+$. In Theorem 2.81, after assuming NL = UL, we show that if $f \in \sharp\mathrm{L}$ and $g \in$ FL such that $g(x)$ is the unary representation of a positive integer k, where $x \in \Sigma^*$ is the input, then the number of ways of choosing exactly $k = |g(x)|$ distinct paths from amongst the $f(x)$ accepting computation paths of the NL-Turing machine corresponding to f is in $\sharp$L.

THEOREM 2.81. *Assume that* NL = UL. *Let Σ be the input alphabet, $f \in \sharp$L and $g \in$ FL such that, for the given input string $x \in \Sigma^*$, $g(x)$ is a positive integer k in the unary representation. Then the function $\binom{f(x)}{k} \in \sharp$L.*

PROOF. Let $x \in \Sigma^*$ be the input string. Given x, we obtain the number $k = |g(x)|$ in FL. It is easy to note that k is upper bounded by a polynomial in $|x|$. It follows from our assumption that NL = UL and Definition 2.65 that there exists a NL-Turing machine M' to which if we give a directed graph G' along with two vertices s' and t' in G' as input then M' outputs "yes" at the end of the unique

accepting computation path and "no" at the end of all the other computation paths if there exists a directed path from s' to t' in G'. Otherwise if there does not exist any directed path from s' to t' in G' then M' outputs "no" at the end of all of its computation paths. Now given the input x, we obtain an instance of SLDAGSTCON, say (G, s, t), using a logspace many-one reduction from Theorem 2.10. It is easy to note that the adjacency matrix of the graph G that we obtain is of size polynomial in $|x|$. Once again we assume that s is in row 1 and t is in row n. Also any two paths are distinct if there exists a vertex in one of the paths that is not in the other path. It follows from Proposition 2.17 that $f(x)$ is equal to the number the directed paths from s to t in G. We associate the label (i, j) to a vertex of G if it is the j^{th} vertex in the i^{th} row of G, where $1 \leq i, j \leq n$. In the following algorithm the row number is denoted by λ and a vertex in row λ is denoted by (λ, ω), where $1 \leq \lambda, \omega \leq n$. The number of distinct paths we have chosen till row λ is denoted by φ and η denotes the number of vertices in row λ that are in the φ distinct paths. We use non-determinism to compute φ and η. Let us consider the SHARPLCFL algorithm described below. Input to the SHARPLCFL algorithm is $(0^k, (G, s, t))$ where (G, s, t) is an input instance of SLDAGSTCON. If using SHARPLCFL, we are able to non-deterministically choose exactly $k = |g(x)|$ distinct directed paths from s to t in G then that computation path ends in the accepting configuration. Otherwise the computation path ends in the rejecting configuration.

Algorithm 10 SHARPLCFL
Input: $(0^k, (G, s, t))$, where $k \in \mathbb{N}$, $G = (V, E)$ is an input instance of SLDAGSTCON and $s, t \in V$.
Output: *accepts* if k distinct directed paths from s to t have been chosen in G. Otherwise *rejects*.
Complexity: ♯L.

$\lambda \leftarrow 1,\ \varphi \leftarrow 1,\ \eta \leftarrow 1$
while $\lambda \leq (n-1)$ **do**
  $\varphi' \leftarrow 0,\ \eta' \leftarrow 0,\ \varphi'' \leftarrow 0,\ \eta'' \leftarrow 0,\ \omega \leftarrow 1$
  while $(\omega \leq n)$ **do**
    Nondeterministically either choose (λ, ω) or skip (λ, ω)
    if (λ, ω) is chosen nondeterministically **then**
      if $(M'(G, s, (\lambda, \omega))$ returns"yes") and $(M'(G, (\lambda, \omega), t)$ returns "yes") **then**
        $\eta' \leftarrow \eta' + 1$
        if $\eta' > \eta$ **then**
          reject the input
        else
          $\sigma \leftarrow 0,\ \psi \leftarrow 0$
          while $\exists$ a neighbour $(\lambda + 1, \rho)$ of (λ, ω) that is yet to be visited **do**
            Nondeterministically either choose $(\lambda + 1, \rho)$ or skip $(\lambda + 1, \rho)$

Algorithm 8 SHARPLCFL (continued)

```
                        if (λ + 1, ρ) is chosen nondeterministically then
                            σ ← σ + 1
                            if M′(G, (λ + 1, ρ), t) returns "no" then
                                reject the input
                            else if (φ = k or φ″ = k) and σ ≥ 2 then
                                reject the input
                            end if
                        end if
                    end while
                    if (σ = 0 or σ > k) then
                        reject the input
                    end if
                    if #(directed paths from s to (λ, ω)) > φ − φ′ then
                        ψ ← max(1, φ − φ′)
                    else
                        ψ ← #(directed paths from s to (λ, ω))
                    end if
                    Nondeterministically choose a number α from 1 to ψ
                    φ′ ← φ′ + α
                    Nondeterministically choose a number β from 0 to σ
                    φ″ ← αβ + φ″
                    η″ ← η″ + β
                    if (η″ = 0) or ((φ = k or φ″ > k) and η″ > η) then
                        reject the input
                    end if
                    if φ″ > k then
                        φ″ ← k
                    end if
                end if
            else
                reject the input
            end if
        end if
        ω ← ω + 1
    end while
    if λ + 1 < n and (η′ ≠ η or φ′ < φ) then
        reject the input
    else if λ + 1 = n and (η′ ≠ η or φ″ ≠ k) then
        reject the input
    end if
    λ ← λ + 1, φ ← φ″, η ← η″
end while
accept the input
```

We note the following points about the SHARPLCFL algorithm.

Always $1 \leq \lambda \leq n$ and $1 \leq \omega \leq n$. In the SHARPLCFL algorithm, if we choose a vertex non-deterministically we verify if it is in a directed path from s to t in lines 7 and 17.

The variable η' is the number of vertices we are choosing non-deterministically in row λ and it must be equal to η after we have made non-deterministic choices on all the vertices in row λ failing which we reject the input in lines 50-54. φ' is used to verify if the number of distinct directed paths from s to t that pass through η' vertices chosen non-deterministically in row λ is atleast φ failing which we reject the input in lines 50-54.

The variable σ computed in line 16 inside the nested while loop from line 13 to 23 is the number of vertices chosen non-deterministically as the neighbours of (λ, ω) in row $\lambda + 1$ such that these vertices are in at least one directed path from s to t in G. Here $1 \leq \sigma \leq n$.

We compute ψ in lines 28 and 30 in FUL by Corollary 2.80 without altering the number of accepting computation paths. Note that $\psi \geq 1$ since (λ, ω) is in at least one directed path from s to t in G. After lines 28 and 30, ψ is either $\max(1, \varphi - \varphi')$ or #(directed paths from s to (λ, ω)). Therefore $1 \leq \psi \leq \varphi$ always. From lines 1, 37-38 and 55 it follows that $1 \leq \varphi \leq k$ always and so in line 35 we always have $\varphi'' \leq kn + \varphi'' \leq 2kn^2$. $1 \leq \alpha \leq \psi$ and so $0 \leq \varphi' \leq 2kn$ always. Also $0 \leq \varphi'' \leq k$ in line 4 at the begining of the while loop. Therefore $0 \leq \varphi'' \leq 2kn^2$ always.

Always $0 \leq \beta \leq \sigma$, $0 \leq \eta' \leq n$, $0 \leq \eta'' \leq n^2$ and $1 \leq \eta \leq n^2$. If $\eta > n$ then we reject the input due to the condition in lines 50 and 52. Variables φ', η' and η'' are updated in the while loop from line 4 to 49 and these values do not decrease inside this loop.

Using non-determinism to increment η'' in lines 34 and 36 by β is to non-deterministically avoid the possibility of counting vertices in row $\lambda + 1$ that are common neighbours of two distinct vertices in row λ more than once.

In lines 32 and 34, assume that we are always non-deterministically choosing the correct value of α and β respectively in the algorithm. Then our algorithm proceeds correctly and ends in an accepting configuration if and only if we have non-deterministically chosen exactly k distinct directed paths from s to t in G. On the contrary if φ'' is updated with an incorrect value of α or if η'' is incremented by an incorrect value of β, then in those iterations of the while loop from line 4 to 49, the algorithm proceeds by assuming that an alternate set of non-deterministic choices have been made on vertices in row $\lambda + 1$ which agree with η'' and φ''.

Any two directed paths formed by non-deterministically choosing two different vertices in row λ in lines 5-6 are distinct irrespective of their neighbours non-deterministically chosen in the row $\lambda + 1$ in lines 14-15. As a result if the number of directed paths from s to t in G is lesser than k then the value of φ'' computed in line 30 is always lesser than k and these inputs are rejected in lines 52-53.

At the end of the while loop in line 49, φ'' is the number of directed paths from s to t computed non-deterministically, that we need for subsequent stages of our

algorithm. Also η'' is the number of distinct vertices that are in row $\lambda+1$ in φ'' distinct directed paths from s to t in G which is also computed non-deterministically.

Now assume that the number of directed paths from s to t in G is at least k. The cases where we reject the input since the non-deterministic choices made on the neighbours of (λ, ω) in row $\lambda + 1$ results in increasing the number of distinct directed paths non-deterministically chosen to be greater than k is in lines 19, 20, 37 and 38.

Lines 19 and 20 are pertaining to the case when we are in vertex (λ, ω) in row λ and we are visiting the neighbours of (λ, ω) in row $\lambda + 1$. In this case we have already non-deterministically chosen k distinct directed paths from s to t and we have also chosen vertices in row $\lambda + 1$ in excess that results in increasing the number of distinct directed paths chosen non-deterministically from s to t to be greater than k. Similarly in lines 37 and 38 we have the case when we have chosen a (λ, ω) in row λ and we have visited all the neighbours of (λ, ω) in row $\lambda + 1$. If ($\varphi = k$ or $\varphi'' > k$) and $\eta'' > \eta$ then it implies that we have chosen more than k distinct directed paths from s to t in G that most recently includes directed paths that pass through (λ, ω) and η'' distinct neighbours of (λ, ω) in row $\lambda + 1$ and so in lines 37 and 38 we reject the input.

The case when we reject the input since we have not chosen η vertices in row $\lambda < n - 1$ or at least φ distinct directed paths till row $\lambda < n - 1$ is in lines 50 and 51. The case when we reject the input since we have not chosen exactly k distinct directed paths from s to t, however we have moved till row $n - 1$ is in lines 52 and 53.

In the SHARPLCFL algorithm, since we keep track of only a constant number of variables all of which take non-negative integer values and the values of these variables are upper bounded by a polynomial in the size of the graph G, which is once again polynomial in the input size $|x|$ we get a NL-Turing machine that executes the SHARPLCFL algorithm. Since we have assumed that $\mathrm{NL} = \mathrm{UL}$ and we use M' to verify if there exists a directed path from a vertex s' to a vertex t' in lines 13 and 27, it follows that the number of accepting computation paths of the NL-Turing machine described by our SHARPLCFL algorithm does not get altered upon simulating M' (also see Note 5). As a result it follows that the NL-Turing machine described by the SHARPLCFL algorithm stops in an accepting state if and only if we start from vertex s in row 1 and reach vertex t in row n via exactly k distinct directed paths. Now as mentioned in Proposition 2.17, since $f(x)$ is equal to the number of directed paths from s to t in G and since we have assumed $\mathrm{NL} = \mathrm{UL}$, it follows that the number of accepting computation paths of this NL-Turing machine is $\binom{f(x)}{k}$. When $f(x) < k$ this NL-Turing machine has no accepting computation paths and so we get $\binom{f(x)}{k} = 0$. This shows that $\binom{f(x)}{k} \in \sharp\mathrm{L}$. □

Exercises

(1) A directed graph $G = (V, E)$ is called strongly connected if every pair of vertices $u, v \in V$ are connected by a directed path in each direction.

Show that language of all directed graphs that are strongly connected is logspace many-one complete for NL, where the directed graph is given as input in terms of its adjacency matrix.

(2) Show that the language of all directed graphs that contain at least one directed cycle is logspace many-one complete for NL where the directed graph is given as input in terms of its adjacency matrix.

(3) Show that NL is closed under union without using Theorem 2.19.

(4) Show that NL is closed under intersection without using Theorem 2.19.

(5) Show that NL is closed under star operation (Kleene closure).

(6) Point out the error in the following deduction:

> Let Σ be the input alphabet, $L \subseteq \Sigma^*$ and $L \in \mathrm{NL}$. Let M' be the NL Turing machine that accepts L such that on any input x of length n there are 2^{n^k} computation paths of M' on x, where $k > 0$ is a constant. For any given input $x \in \Sigma^*$, we know that $x \in L$ if and only if $acc_{M'} > 0$. Let $f(x) = acc_{M'}(x) + 2^{n^k}$, where $n = |x|$. Then $f \in \sharp\mathrm{L}$. Also if M is the NL Turing machine of f then the number of computation paths of M on any input x of length n is 2^{n^k+1}. Clearly $L \in \mathrm{NL}$, and $x \in L$ if and only if $f(x) > 2^{n^k}$. Now, if $L' \in \mathrm{PL}$ then there exists $g \in \mathrm{GapL}$ such that on any input x we have $x \in L'$ if and only if $g(x) > 0$. So PL $\subseteq$ NL which implies that NL = PL.

(7) (**Polynomially bounded Isolating Lemma**) Let $p(x)$ be a polynomial and let $r > n^2p^2(n)$ be a prime, where $n > 0$ and $n \in \mathbb{Z}$. Also let $0 < m < r$ and $m \in \mathbb{Z}$. Define $e_m(\pi) = \Sigma_{i=1}^{n}(m^{ni+\pi(i)} \bmod r)$, where $\pi \in S_n$ is a permutation of n elements. Let $\pi_1, \pi_2 \ldots, \pi_t \in S_n$ for some $t \leq p(n)$. Show that there exists a $m < r$ such that $e_m(\pi_i) \neq e_m(\pi_j)$, for all $i \neq j$ and $1 \leq i, j \leq t$.

(8) The Directed Graph Isomorphism problem consists of deciding whether there is a bijection between the set of nodes of the two input directed graphs G and H such that the bijection preserves edge relations. Consider the Colored Directed Graph Isomorphism problem, in which the nodes are colored by different colors, such that we have to decide whether there is a bijection between the set of nodes of the two input directed graphs G and H which preserves the colors of the nodes and edge relations, where G and H are given in terms of their adjacency matrix. In other words, the bijection maps vertices colored with the same color (say, red) in G to vertices colored with the same color (say, red) in H and it also preserves edge relations between every pair of vertices. The k-Colored Directed Graph Isomorphism problem is the Colored Directed Graph Isomorphism problem where the number of vertices colored with a color is at most a constant $k > 0$.

The Directed Graph Automorphism problem consists of deciding if there exists an isomorphism from a directed graph to itself, where the input directed graph is given in terms of its adjacency matrix. Similarly,

the Colored Directed Graph Automorphism problem consists of deciding if there exists an isomorphism from a directed graph, whose nodes are colored, to itself. The k-Colored Directed Graph Automorphism problem consists of deciding if there exists an automorphism from a colored directed graph to itself, where the number of vertices having a color is less than or equal to k, for some constant $k > 0$.

- Show that if $k = 2, 3$ then k-Colored Directed Graph Isomorphism is logspace many-one complete for NL.
- Show that if $k = 2, 3$ then k-Colored Directed Graph Automorphism is in NL.

Open problems

(1) Is $\sharp\mathrm{L}$ closed under division by a function in FL, which outputs a non-negative integer on a given input? In other words, is the quotient obtained by dividing the value of a $\sharp\mathrm{L}$ function by a function in FL, which outputs a non-negative integer on any given input, also in $\sharp\mathrm{L}$?
(2) Is $\sharp\mathrm{L}$ closed under the modulo operation by a function in FL, which outputs a non-negative integer on a given input? In other words, is the remainder obtained by dividing the value of a $\sharp\mathrm{L}$ function by a function in FL, which outputs a non-negative integer on any given input, also in $\sharp\mathrm{L}$?
(3) Is it possible to strengthen Theorem 2.71 by showing the following: "$\mathrm{NL} = \mathrm{UL}$ if and only if there is a polynomially bounded UL computable weight function f so that for any directed acyclic graph G, we have $f(G)$ is ***min-unique***?"
(4) Is it possible to prove Theorem 2.81 unconditionally, that is without assuming $\mathrm{NL} = \mathrm{UL}$?
(5) Is GapL closed under division by a function in FL, which outputs an integer on any given input? In other words, is the quotient obtained by dividing the value of a GapL function by a function in FL which outputs an integer on any given input, also in GapL?
(6) Is GapL closed under the modulo operation by a function in FL, which outputs an integer on all inputs? In other words, is the remainder obtained by dividing the value of a GapL function by a function in FL which outputs an integer on any given input, also in GapL?

Notes

The Immerman-Szelepcsenyi Theorem (Section 2.2, Theorem 2.28) is a celebrated result in the Theory of Computational Complexity [**BDG95, AB09, DK14, Sip13**] which generalizes the result that $\mathrm{NL} = \mathrm{co\text{-}NL}$ shown in Theorem 2.18 for non-deterministic space bounded complexity classes above NL. Even though the algorithm in the proof of Theorem 2.18 is based on the algorithm given by Michael Sipser in [**Sip13**, Section 8.6, Theorem 8.27], our proof is slightly different from this proof since it depends on an input instance of the NL-complete language SLDAGSTCON. However our proof has a shortcoming that it is not as much

general as the proof in [**Sip13**, Section 8,6, Theorem 8.27] which seems to work even for showing that the directed st-connectivity problem is closed under complement for restricted type of inputs such as directed planar graphs and so on. Also both these proofs use the counting method which is commonly called as the double inductive counting method (for example, see [**BDG95**, Appendix]). However the term ***non-deterministic counting*** seems to be more apt and akin to the method used to prove our theorem and so we refer to the method as the non-deterministic counting method in this manuscript.

Logarithmic space bounded counting classes were first introduced in analogy with the counting classes in the polynomial time setting such as $\sharp$P. The logarithmic space bounded complexity class $\sharp$L was defined by Carme Alvarez and Birgit Jenner in [**AJ93**]. Complexity classes GapL were defined by Eric Allender and Mitsunori Ogihara in [**AO96**].

Results shown in Section 2.3.1 are from [**AO96, BDH$^+$92**] and the excellent textbook by Lane A. Hemaspaandra and Mitsunori Ogihara [**HO01**, Chapter 9].

The Isolating Lemma was invented by Ketan Mulmuley, Umesh Vazirani and Vijay Vazirani, see [**MR95**, Chapter 12]. It plays an important role in designing randomized parallel algorithms for problems such as the perfect matching in undirected graphs. Our proof of the Isolating Lemma in Section 2.4 is based on Lemma 4.1 in the textbook by Lane A. Hemaspaandra and Mitsunori Ogihara [**HO01**]. The complexity class UL was first defined and studied by Gerhard Buntrock, Birgit Jenner, Klaus Jorn Lange and Peter Rossmanith in [**BJL$^+$91**]. The result that $\mathrm{NL/poly} = \mathrm{UL/poly}$ is based on the Isolating Lemma and it was proved by Klaus Reinhardt and Eric Allender in [**RA00**] (see also [**HO01**, Section 4.3]). Our exposition of this result in Sections 2.4.1 and 2.4.2 (and especially our proof of Theorem 2.68) is from results shown by Chris Bourke, Raghunath Tewari and N. V. Vinodchandran in [**BTV09**]. Theorem 2.71 is due to Aduri Pavan, Raghunath Tewari and N. V. Vinodchandran and it is from [**PTV12**].

We also note that Manindra Agrawal, Thanh Minh Hoang and Thomas Thierauf have shown a restricted version of the Isolating Lemma in [**AHT07**], where the weight function is computable in deterministic logspace when the number of subsets of the set A in the family $\mathcal{F}$ is at most a polynomial in the size of A. This result is stated in Exercise no.(7) of this chapter. They use this result to show that the problem of counting the number of perfect matchings in instances of the polynomially bounded perfect matching problem for bipartite and general undirected graphs is in $\mathrm{C_{=}L}$. Exercise problem (8) of this chapter on the directed graph isomorphism is adapted from the results proved on the undirected graph isomorphism by [**JKM$^+$03**].

Restricted versions of the directed st-connectivity problem, such as for planar graphs, grid graphs and graphs embedded on surfaces, each of which defines and uses deterministic logspace computable weight functions to isolate directed paths between a pair of vertices in the input graph, have been investigated and many interesting results on their computational complexity that use logarithmic space

bounded counting classes are known due to Eric Allender, David A. Mix Barrington, Chris Bourke, Tanmoy Chakraborty, Samir Datta, Sambuddha Roy, Raghunath Tewari and N. V. Vinodchandran, from [**BTV09, ABC$^+$09**]. We also note that Raghunath Tewari and N. V. Vinodchandran define a deterministic logspace computable weight function in [**TV12**] that can be used to isolate a directed path between any two vertices in a directed planar graph.

We refer to [**TM97**, Chapter 1, Section 1-3] for a nice introduction to Boolean logic and its normal forms such as the disjunctive normal form (DNF) and the conjunctive normal form (CNF) of propositional formulae. It is a standard result in Boolean logic on the satisfiability of Boolean formulae that a Boolean formula ϕ is equivalent to a Boolean formula ϕ' in the CNF such that ϕ is satisfiable if and only if ϕ' is satisfiable. We recall the definition of 3-CNF from [**CLR$^+$22**, Chapter 34, Section 34.4, pp.1076]. Also any Boolean formula ϕ' which is in CNF, is equivalent to a Boolean formula ϕ'' which is in the 3-CNF [**CLR$^+$22**, Chapter 34, Theorem 34.10] such that ϕ' is satisfiable if and only if ϕ'' is satisfiable. It is well known and fundamental result of computational complexity that the problem (denoted by 3SAT), of determining if a Boolean formula which is in 3-CNF has a satisfying assignment of its Boolean variables is NP-complete under polynomial time many-one reductions. Similar to 3-CNF and 3SAT, it is easy to define 2-CNF and 2SAT. In analogy with the NP-completeness of 3SAT we are able to show in Theorem 2.25 that 2SAT is logspace many-one complete for NL which is an interesting observation. Note that unlike Propositions 2.16 and 2.17, we are unable to immediately conclude from Theorem 2.25 that $\sharp$2SAT is logspace many-one complete for $\sharp\mathrm{L}$ and we leave this question as a conjecture.

Computer programs which solve decision problems or compute functions essentially map inputs to outputs. Descriptive Complexity is a branch of Theoretical Computer Science developed by N. Immerman [**Imm99**], which uses first- and second-order mathematical logic to describe computer programs. In descriptive complexity, using first-order languages it is demonstrated that all measures of complexity can be mirrored in logic and most important complexity classes have very elegant and clean descriptive characterizations. In fact, N. Immerman gives a descriptive complexity proof of the Immerman–Szelepcsenyi Theorem (Theorem 2.28) in [**Imm99**].

CHAPTER 3

Modulo-based Logarithmic space bounded counting classes

In this chapter we introduce modulo-based logarithmic space bounded counting classes and prove many interesting properties about these complexity classes.

DEFINITION 3.1. Let Σ be the input alphabet. Also let $k \in \mathbb{Z}^+$ and let $k \geq 2$. We say that $L \subseteq \Sigma^*$ is a language in the complexity class $\mathrm{Mod}_k\mathrm{L}$ if there exists a function $f \in \sharp\mathrm{L}$ such that for any input $x \in \Sigma^*$ we have $x \in L$ if and only if $f(x) \not\equiv 0 (\mathrm{mod}\ k)$.

PROPOSITION 3.2. *Let Σ be the input alphabet. Also let $k \in \mathbb{Z}^+$ and let $k \geq 2$. For every $j \in \mathbb{Z}$, there exists a function $g \in \sharp\mathrm{L}$ such that on any input $x \in \Sigma^*$, we have $x \in L$ if and only if $g(x) \not\equiv j (mod\ k)$.*

PROOF. Let $L \in \mathrm{Mod}_k\mathrm{L}$ such that on any input $x \in \Sigma^*$, we have $x \in L$ if and only if $f(x) \not\equiv 0 (\mathrm{mod}\ k)$, where $f \in \sharp\mathrm{L}$. Without loss of generality, assume that $0 \leq j < k$. Since $\sharp\mathrm{L}$ is closed under addition (Proposition 2.33), it follows that $g(x) = (f(x) + j) \in \sharp\mathrm{L}$. As a result, if M_g is the NL-Turing machine corresponding to g then $x \in L$ if and only if $f(x) \not\equiv 0 (\mathrm{mod}\ k)$ if and only if $g(x) \not\equiv j (\mathrm{mod}\ k)$. □

LEMMA 3.3. *Let Σ be the input alphabet. Also let $p \in \mathbb{N}$ such that $p \geq 2$ and p is a prime.*

(1) *There exists a function $g \in \sharp\mathrm{L}$ such that on any input $x \in \Sigma^*$, we have*
   - *if $x \in L$ then $g(x) \equiv 1 (mod\ p)$, and*
   - *if $x \notin L$ then $g(x) \equiv 0 (mod\ p)$.*

(2) *There exists a function $g \in \sharp\mathrm{L}$ such that on any input $x \in \Sigma^*$, we have*
   - *if $x \in L$ then $g(x) \equiv i (mod\ p)$, and*
   - *if $x \notin L$ then $g(x) \equiv j (mod\ p)$.*

PROOF. (1) Let $L \in \mathrm{Mod}_p\mathrm{L}$ such that on any input $x \in \Sigma^*$, we have $x \in L$ if and only if $f(x) \not\equiv 0 (\mathrm{mod}\ p)$, where $f \in \sharp\mathrm{L}$. Now we define $g(x) = (f(x))^{p-1}$, where $x \in \Sigma^*$. It follows from Proposition 2.33 that $g \in \sharp\mathrm{L}$. Using the Fermat's Little Theorem, it follows that if $x \in L$ then $\{f(x) \not\equiv 0 (\mathrm{mod}\ p)$ if and only if $g(x) \equiv 1 (\mathrm{mod}\ p)\}$. On the other hand if $x \notin L$ then $\{f(x) \equiv 0 (\mathrm{mod}\ p)$ if and only if $g(x) \equiv 0 (\mathrm{mod}\ p)\}$.

(2) Let $L \in \mathrm{Mod}_p\mathrm{L}$ such that on any input $x \in \Sigma^*$, there exists $f \in \sharp\mathrm{L}$ such that if $x \in L$ then $f(x) \equiv 1 (\mathrm{mod}\ p)$ and if $x \notin L$ then $f(x) \equiv 0 (\mathrm{mod}\ p)$. Now we define $g(x) = ((i-j)f(x)) + j$, where $x \in \Sigma^*$. It follows from

Proposition 2.33 that $g \in \sharp\mathrm{L}$. Also if $x \in L$ then $\{f(x) \equiv 1(\mathrm{mod}\, p)$ if and only if $g(x) \equiv i(\mathrm{mod}\, p)\}$. On the other hand if $x \notin L$ then $\{f(x) \equiv 0(\mathrm{mod}\, p)$ if and only if $g(x) \equiv j(\mathrm{mod}\, p)\}$.

□

THEOREM 3.4. *Let Σ be the input alphabet. Also let $p \in \mathbb{N}$ such that $p \geq 2$ and p is a prime. Then,*

1. $\mathrm{Mod}_p\mathrm{L}$ *is closed under intersection,*
2. $\mathrm{Mod}_p\mathrm{L}$ *is closed under complement, and*
3. $\mathrm{Mod}_p\mathrm{L}$ *is closed under union.*

PROOF. (1) Let $L_1, L_2 \in \mathrm{Mod}_p\mathrm{L}$ and let $f_1, f_2 \in \sharp\mathrm{L}$ such that given any input $x \in \Sigma^*$, we have $x \in L_i$ if and only if $f_i(x) \not\equiv 0(\mathrm{mod}\, p)$, where $i = 1, 2$. Let us define $f = f_1 f_2$. It follows from Proposition 2.33 that $f \in \sharp\mathrm{L}$. Since p is a prime, on any input $x \in \Sigma^*$, we have $f(x) \not\equiv 0(\mathrm{mod}\, p)$ if and only if $f_1(x) \not\equiv 0(\mathrm{mod}\, p)$ and $f_2(x) \not\equiv 0(\mathrm{mod}\, p)$. Therefore $L_1 \cap L_2 \in \mathrm{Mod}_p\mathrm{L}$.

(2) Let $L \in \mathrm{Mod}_p\mathrm{L}$ using $f \in \sharp\mathrm{L}$. For any $j \in \mathbb{N}$, define $L_j = \{x | f(x) \not\equiv j(\mathrm{mod}\, k)\}$. Then, $\overline{L} = \cap_{1 \leq j < p} L_j$. However $L_j \in \mathrm{Mod}_p\mathrm{L}$ due to Proposition 3.2 and $\mathrm{Mod}_p\mathrm{L}$ is closed under intersection from (1). Therefore $\overline{L} \in \mathrm{Mod}_p\mathrm{L}$.

(3) Follows from (1) and (2).

□

DEFINITION 3.5. Let Σ be the input alphabet such that $\{0, 1\} \subseteq \Sigma$ and let $L_1, L_2 \subseteq \Sigma^*$. We define the join of L_1 and L_2 to be $L_1 \vee L_2 = \{x1 | x \in L_1\} \cup \{y0 | y \in L_2\}$.

PROPOSITION 3.6. *Let Σ be the input alphabet. Also let $k \in \mathbb{Z}^+$ such that $k \geq 2$. $\mathrm{Mod}_k\mathrm{L}$ is closed under join.*

PROOF. Let $L_i \in \mathrm{Mod}_k\mathrm{L}$ and assume that we decide if an input string $x \in \Sigma^*$ is in L_i using functions $f_i \in \sharp\mathrm{L}$, where $i = 1, 2$. We assume without loss of generality that the size of any input string x is at least 1. Let $x = x_1 x_2 \cdots x_{n+1}$ and let $y = x_1 \cdots x_n$ where $n \in \mathbb{Z}^+$. Also let

$$f(x) = \begin{cases} f_1(y) & \text{if } x_{n+1} = 1 \\ f_2(y) & \text{otherwise if } x_{n+1} = 0. \end{cases}$$

It is easy to note that $f \in \sharp\mathrm{L}$. Now $f(x) \not\equiv 0(\mathrm{mod}\, k)$ for any input $x = yx_{n+1}$ if and only if exactly one of the following is true: ($x_{n+1} = 1$ and $y \in L_1$) or ($x_{n+1} = 0$ and $y \in L_2$). Clearly this shows that $L_1 \vee L_2 \in \mathrm{Mod}_k\mathrm{L}$ which completes the proof. □

THEOREM 3.7. *Let Σ be the input alphabet, $p \in \mathbb{N}$ and $p \geq 2$ such that p is a prime.* $\mathrm{L}^{\mathrm{Mod}_p\mathrm{L}} = \mathrm{Mod}_p\mathrm{L}$.

PROOF. Our proof of this result is similar to the proof of Theorem 2.19. Let $L \in \mathrm{L}^{\mathrm{Mod}_p\mathrm{L}}$. Then there exists a $O(\log n)$-space bounded deterministic Turing

machine M^A, that has access to a language $A \in \text{Mod}_p\text{L}$ as an oracle such that given any input $x \in \Sigma^*$, M^A decides if $x \in L$ correctly. Since M requires at most $O(\log n)$ space on any input of size n, we infer that the number of queries that M submits to the oracle A is a polynomial in the size of the input. Also it follows from Lemma 3.3(1) that, for the oracle A there exists a NL-Turing machine M_A corresponding to which we have $g \in \sharp\text{L}$ such that on any input $x \in \Sigma^*$, we have if $x \in A$ then $g(x) \equiv 1(\text{mod}\, p)$, and if $x \notin A$ then $g(x) \equiv 0(\text{mod}\, p)$. Since we know from Theorem 3.4(2) that Mod_pL is closed under complement, let $M_{\overline{A}}$ be the NL-Turing machine corresponding to which we have $h \in \sharp\text{L}$ such that on any input $x \in \Sigma^*$, we have if $x \in \overline{A}$ then $h(x) \equiv 1(\text{mod}\, p)$, and if $x \notin \overline{A}$ then $h(x) \equiv 0(\text{mod}\, p)$. Let us now consider the following algorithm implemented by a NL-Turing machine N given input $x \in \Sigma^*$.

Algorithm 9 Closure-Logspace-Turing-for-Mod_pL
Input: $x \in \Sigma^*$.
Output: *accept* or *reject*. The NL-Turing machine that implements this algorithm obeys the property of Mod_pL in Lemma 3.3(1).
Complexity: Mod_pL.

1: **while** N has not reached any of its halting configurations **do**
2: N simulates M on input x until a query y is generated.
3: N non-deterministically guesses if $y \in A$.
4: **if** N guesses that $y \in A$ **then**
5: N simulates M_A on input y.
6: **if** M_A rejects y **then**
7: N also rejects x and stops.
8: **end if**
9: **else**
10: N simulates $M_{\overline{A}}$ on y
11: **if** $M_{\overline{A}}$ rejects y **then**
12: N also rejects x and stops.
13: **end if**
14: **end if**
15: N continues to simulate M on x.
16: **end while**

Let $f \in \sharp\text{L}$ denote the number of accepting computation paths of N. Since N rejects the input x in step 4 of the above stated algorithm whenever its simuation of M_A or $M_{\overline{A}}$ rejects their input y, it follows that N accepts the input x at the end of any computation path if and only if it guesses the oracle reply correctly which is the same as in the end of the computation path of M_A or $M_{\overline{A}}$. As a result, on any given input $x \in \Sigma^*$, if $x \in L$ and if N has made correct guesses for each of the query strings generated then $f(x) \equiv 1(\text{mod}\, p)$, since f is a polynomial length product of g and h. However if $x \in L$ and N has made at least one incorrect guess for the query string y that is generated then $f(x)$ remains unaffected since N rejects

at the end of such computation paths. Conversely, if $x \notin L$ and if N has made the correct guesses for each of the query strings generated then $f(x) \equiv 0(\text{mod } p)$, once again since f is a polynomial length product of g and h. Similar to the previous case, if $x \notin L$ and N has made at least one incorrect guess for the query string y that is generated then $f(x)$ remains unaffected since N rejects at the end of such computation paths. This shows that there exists $f \in \sharp\mathrm{L}$ such that for any input $x \in \Sigma^*$, we have if $x \in L$ then $f(x) \equiv 1(\text{mod } p)$ and if $x \notin L$ then $f(x) \equiv 0(\text{mod } p)$ which proves that $L \in \mathrm{Mod}_p\mathrm{L}$. □

THEOREM 3.8. *Let $p \in \mathbb{N}$ and $p \geq 2$ such that p is a prime.* $\mathrm{Mod}_p\mathrm{L}^{\mathrm{Mod}_p\mathrm{L}} = \mathrm{Mod}_p\mathrm{L}$.

PROOF. Let $L \in \mathrm{Mod}_p\mathrm{L}^{\mathrm{Mod}_p\mathrm{L}}$. Then there exists a NL-Turing machine M^A, that has access to a language $A \in \mathrm{Mod}_p\mathrm{L}$ as an oracle such that given any input $x \in \Sigma^*$, M^A decides if $x \in L$ correctly. Since M is a non-deterministic Turing machine, we assume that M follows the Ruzzo-Simon-Tompa oracle access mechanism stated in Section 1.1.6 to access the oracle A. Also since it requires at most $O(\log n)$ space on any input of size n, we infer that the number of queries that M submits to the oracle A is a polynomial in the size of the input. Now remaining part of our proof of this theorem is is based on Theorem 3.7 and it is similar to Theorem 2.22. □

PROPOSITION 3.9. *If $j, k \in \mathbb{Z}^+$ such that $j, k \geq 2$ and $j|k$ then* $\mathrm{Mod}_j\mathrm{L} \subseteq \mathrm{Mod}_k\mathrm{L}$.

PROOF. Let $k = cj$. If $L \in \mathrm{Mod}_j\mathrm{L}$ using a NL-Turing machine N such that $f \in \sharp\mathrm{L}$ is the number of accepting computation paths of N on any input, then we can define $g = cf$ which is in $\sharp\mathrm{L}$ due to Proposition 2.33. Clearly, given an input $x \in \Sigma^*$, we have $x \in L$ if and only if $f(x) \not\equiv 0(\text{mod } j)$ if and only if $g(x) \not\equiv 0(\text{mod } k)$ which proves the result. □

THEOREM 3.10. **(Kummer)** *Let p be a prime, and let $n = a + b$. Then,*

$$\binom{n}{a} \equiv 0(\text{mod } p^c)$$

if and only if the number of carries when adding a to b in base-p is at least c.

COROLLARY 3.11. *Let p be a prime. Then,*

$$\binom{n}{p^k} \equiv 0(\text{mod } p)$$

if and only if the coefficient of p^k in the base-p expansion of n is zero.

PROOF. Suppose that the coefficient of p^k in the base-p expansion of n is not zero. Using base-p arithmetic, we have

$$\begin{aligned} n &= d_m p^m + \cdots + d_k p^k + d_{k-1} p^{k-1} + \cdots + d_0 p^0, \\ p^k &= 0 + p^k + 0 + \cdots + 0, \\ n - p^k &= d_m p^m + \cdots + (d_k - 1) p^k + d_{k-1} p^{k-1} + \cdots + d_0 p^0, \end{aligned}$$

so p can be added to $n - p^k$ in base-$p$without carrying. On the other hand, if the coefficient of p^k in the base-p expansion of n is zero then the coefficient of p^k in the base-p expansion of $n - p^k$ must be $p - 1$; therefore, there must be a carry when adding 1 to $p-1$ in the p^k's position. The corollary, follows from Kummer's theorem with $c = 1$. □

LEMMA 3.12. *Let $p, e \in \mathbb{N}$ and $p \geq 2$ such that p is a prime.* $\mathrm{Mod}_{p^e}\mathrm{L} = \mathrm{Mod}_p\mathrm{L}$.

PROOF. It follows from Proposition 3.9 that $\mathrm{Mod}_p\mathrm{L} \subseteq \mathrm{Mod}_{p^e}\mathrm{L}$. To prove the converse, we use induction on e. Assume that the converse result is true for some $e \geq 1$. Let $L \in \mathrm{Mod}_{p^{e+1}}\mathrm{L}$ using a NL-Turing machine N. Let $g \in \sharp\mathrm{L}$ denote the number of accepting computation paths. A number g is divisible by p^{e+1} if and only if

(1). g is divisible by p^e, and

(2). the coefficient of p^e in the base-p expansion of g is 0.

Also using Corollary 3.11, we get that condition (2) is equivalent to $\binom{g}{p^e} \equiv 0 (\mathrm{mod}\, p)$. Therefore, on a given input $x \in \Sigma^*$, we have $g(x) \not\equiv 0 (\mathrm{mod}\, p^{e+1})$ if and only if $\{g(x) \not\equiv 0 (\mathrm{mod}\, p^e)$ or $\binom{g(x)}{p^e} \not\equiv 0 (\mathrm{mod}\, p)\}$. Now let $L_1 = \{x \in \Sigma^* | g(x) \not\equiv 0 (\mathrm{mod}\, p^e)\}$ and $L_2 = \{x \in \Sigma^* | \binom{g(x)}{p^e} \not\equiv 0 (\mathrm{mod}\, p)\}$. By the induction hypothesis $L_1 \in \mathrm{Mod}_p\mathrm{L}$. Also it follows from Theorem 2.57 that $L_2 \in \mathrm{Mod}_p\mathrm{L}$. As a result $L_1 \cup L_2 \in \mathrm{Mod}_p\mathrm{L}$ which proves our result. □

LEMMA 3.13. *Let j and k be relatively prime. Then $L \in \mathrm{Mod}_{jk}\mathrm{L}$ if and only if there exists $L_j \in \mathrm{Mod}_j\mathrm{L}$ and $L_k \in \mathrm{Mod}_k\mathrm{L}$ such that $L = L_j \cup L_k$.*

PROOF. Let $L \in \mathrm{Mod}_{jk}\mathrm{L}$ using a NL-Turing machine M such that $f(x) \in \sharp\mathrm{L}$ denotes the number of accepting computation paths of $M(x)$, where $x \in \Sigma^*$ and Σ is the input alphabet. Also let $L_i = \{x | f(x) \not\equiv 0 (\mathrm{mod}\, i)\}$ for $i = j, k$. Clearly $L = L_j \cup L_k$.

Conversely, let $L_i \in \mathrm{Mod}_i\mathrm{L}$ using NL-Turing machine M_i such that $f_i(x)$ denotes the number of accepting computation paths of $M(x)$, where $x \in \Sigma^*$ and $i = j, k$. It follows from Proposition 2.33 that $\sharp\mathrm{L}$ is closed under addition and multiplication and so $f(x) = (kf_j(x) + jf_k(x)) \in \sharp\mathrm{L}$. Clearly, $x \in L_j \cup L_k$ if and only if $f(x) \not\equiv 0 (\mathrm{mod}\, jk)$ which proves the result. □

THEOREM 3.14. *Let $k = p_1^{e_1} p_2^{e_2} \cdots p_m^{e_m}$ be the prime factorization of $k \in \mathbb{N}$ and $k \geq 2$. $L \in \mathrm{Mod}_k\mathrm{L}$ if and only if there are languages $L_i \in \mathrm{Mod}_{p_i^{e_i}}\mathrm{L}$ such that $L = \cup_{i=1}^m L_i$, where $1 \leq i \leq m$. Especially the complexity class $\mathrm{Mod}_k\mathrm{L} = \mathrm{Mod}_{p_1 p_2 \cdots p_m}\mathrm{L}$.*

PROOF. By Lemma 3.12, $\mathrm{Mod}_{p^e}\mathrm{L} = \mathrm{Mod}_p\mathrm{L}$. Our theorem follows from Lemma 3.13 using induction on m. □

COROLLARY 3.15. $\mathrm{Mod}_k\mathrm{L}$ *is closed under union, where $k \in \mathbb{N}$ and $k \geq 2$.*

PROOF. It follows from Theorem 3.4(3) that $\mathrm{Mod}_p\mathrm{L}$ is closed under union if $p \in \mathbb{N}$ is a prime. Now using Lemma 3.12 and applying Theorem 3.14 we obtain our result. □

3.1. ModL: an extension of modulo

From now onwards, and the rest of this chapter, when we consider functions in GapL***, we do not necessarily assume that the computation tree of a function in*** GapL ***is a complete binary tree.***

For the purpose of the complexity class ModL which we study here, we first recollect Theorems 1.29 and 1.31 and basics of logarithmic space bounded computation from Section 1.2.1.

DEFINITION 3.16. Let Σ be the input alphabet. A language $L \subseteq \Sigma^*$ is in the complexity class ModL if there is a function $f \in \mathrm{GapL}$ and a function $g \in \mathrm{FL}$ such that on any input $x \in \Sigma^*$,

- $g(x) = 1^{p^e}$ for some prime p and a positive integer e, and
- $x \in L \Leftrightarrow f(x) \not\equiv 0 (\mathrm{mod}\ p^e)$.

DEFINITION 3.17. Let Σ be an input alphabet and let Γ be an output alphabet. We define $\mathrm{FL}^{\mathrm{GapL}}$ to be the complexity class of all functions $f : \Sigma^* \to \Gamma^*$, which are logspace Turing reducible to the complexity class of functions in GapL.

DEFINITION 3.18. Let Σ be an input alphabet and let Γ be an output alphabet. We define $\mathrm{FL}^{\mathrm{ModL}}$ to be the complexity class of all functions $f : \Sigma^* \to \Gamma^*$, which are logspace Turing reducible to the complexity class of languages in ModL.

The ***prime distribution function*** $\pi(n)$ specifies the number of primes that are less than or equal to n.

THEOREM 3.19. **(Prime Number Theorem)**

$$\lim_{n \to \infty} \frac{\pi(n)}{n/\ln n} = 1.$$

The approximation $n/\ln n$ gives reasonably accurate estimates of $\pi(n)$ even for small n.

THEOREM 3.20. **(Chinese Remainder Theorem)** *Let $n = n_1 n_2 \cdots n_k$, where the n_i are relatively prime. Consider the correspondence*

$$a \quad \leftrightarrow \quad (a_1, \ldots, a_k), \tag{3.1}$$

where $a \in \mathbb{Z}_n$, $a_i \in \mathbb{Z}_{n_i}$, and

$$a_i \equiv a (\mathrm{mod}\ n_i)$$

for $i = 1, 2, \ldots k$. Then, mapping (3.1) is a one-to-one correspondence (bijection) between $\mathbb{Z}_n$ and the Cartesian product $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$. Operations performed on the elements of $\mathbb{Z}_n$ can be equivalently performed on the corresponding

k-tuples by performing the operations independently in each coordinate position in the appropriate system. That is, if

$$\begin{aligned} a &\leftrightarrow (a_1, \ldots, a_k), \\ b &\leftrightarrow (b_1, \ldots, b_k), \end{aligned}$$

then

$$\begin{aligned} (a+b) \pmod n &\leftrightarrow ((a_1+b_1) \pmod{n_1}, \ldots, (a_k+b_k) \pmod{n_k}), \\ (a-b) \pmod n &\leftrightarrow ((a_1-b_1) \pmod{n_1}, \ldots, (a_k-b_k) \pmod{n_k}), \\ (ab) \pmod n &\leftrightarrow ((a_1 b_1) \pmod{n_1}, \ldots, (a_k b_k) \pmod{n_k}). \end{aligned}$$

LEMMA 3.21. $\mathrm{FL}^{\mathrm{ModL}} = \mathrm{FL}^{\mathrm{GapL}}$.

PROOF. For the forward inclusion it suffices to show that $\mathrm{ModL} \subseteq \mathrm{L}^{\mathrm{GapL}}$, since $\mathrm{L}^{\mathrm{L}^{\mathrm{GapL}}} = \mathrm{L}^{\mathrm{GapL}}$ from Proposition 2.54. Suppose $L \in \mathrm{ModL}$ is witnessed by a function $f \in \mathrm{GapL}$ and a function $g \in \mathrm{FL}$ that computes prime powers in unary. Now, a logarithmic space bounded deterministic Turing machine can retrieve the value of $f(x)$ for any input x to the GapL oracle as stated in Section 1.1.5. It is also easy to see that checking if $f(x) \not\equiv 0 (\mathrm{mod}\ |g(x)|)$ is also computable in logarithmic space. As a result it follows that $L \in \mathrm{L}^{\mathrm{GapL}}$. Therefore $\mathrm{L}^{\mathrm{ModL}} \subseteq \mathrm{L}^{\mathrm{GapL}}$.

For the reverse inclusion, let $L \in \mathrm{L}^{\mathrm{GapL}}$ be computed by a logarithmic space bounded deterministic Turing machine with access to a function $f \in \mathrm{GapL}$ as an oracle. (It follows from Theorem 2.55 that $\mathrm{L}^{\sharp\mathrm{L}} = \mathrm{L}^{\mathrm{GapL}}$. So we can assume that the function $f \in \sharp\mathrm{L}$ and that it is non-negative on all inputs). For $x \in \Sigma^n$, we have $\mathrm{size}(f(x)) \leq p(n)$, for some polynomial $p(n)$. By the Prime Number Theorem, the number of primes between 2 and $p^2(n)$ is $O(p^2(n)/\log n) > p(n)$, for sufficiently large n. Also the first $p(n)$ primes are of size $O(\log n)$. Furthermore, the product of the first $p(n)$ primes is greater than $f(x)$. Also it is easy to see using Corollary 1.30, that checking if a $O(\log n)$-bit integer is a prime can be done in logarithmic space. Furthermore, in logarithmic space, we can also compute the i^{th} prime for $1 \leq i \leq p(n)$. Let p_i denote the i^{th} prime. We define the function $g \in \mathrm{FL}$ as follows: $g(x, 0^{p(|x|)}, i) = 0^{p_i}$ if $i \leq p(|x|)$, and it is 0^2 otherwise.

We define the following language in ModL

$$L' = \{\langle x, 0^{p(|x|)}, i, k\rangle \mid f(x) \equiv k (\mathrm{mod}\ g(x, 0^{p(|x|)}, i)), i \leq p^2(|x|), k \leq p^2(|x|)\}.$$

In order to show that $L \in \mathrm{L}^{\mathrm{ModL}}$ we need to simulate the $\mathrm{L}^{\mathrm{GapL}}$ Turing machine for L with a $\mathrm{L}^{\mathrm{ModL}}$ computation. Clearly, it suffices to show that each GapL query $f(x)$ made by the base logarithmic space bounded deterministic Turing machine can be simulated in $\mathrm{L}^{\mathrm{ModL}}$. For each $1 \leq i \leq p(n)$, we can query L' for $\langle x, 0^{p(|x|)}, i, k\rangle$ for different values of $k \leq p^2(|x|)$ to find $f(x) (\mathrm{mod}\ p_i)$.

Now, by the Chinese Remainder Theorem, $f(x)$ is uniquely determined by $f(x) (\mathrm{mod}\ p_i)$, for $1 \leq i \leq p(n)$. Moreover, given these residues $f(x) (\mathrm{mod}\ p_i)$, for $1 \leq i \leq p(n)$, it follows from Theorem 1.29, that it is possible to compute

$f(x)$ in logarithmic space (in fact, given the residues of $f(x) (\mathrm{mod}\, p_i)$, for $1 \leq i \leq p(n)$, it is possible to compute $f(x)$ in DLOGTIME-uniform TC^0 which is a very restricted form of uniformity of the uniform TC^0 complexity class). Hence a logarithmic space bounded deterministic oracle Turing machine with access to the ModL oracle L' can recover $f(x)$ for each query x. As a result it follows easily that L is in $\mathrm{L}^{\mathrm{ModL}}$. □

3.1.1. Characterization of ModL. In this section we study the complexity class ModL and obtain a characterization of ModL that shows that a ♯L function f and a FL function g that outputs a prime number in unary representation are sufficient to decide if a given input x is in a language $L \in$ ModL, however under the assumption that $\mathrm{NL} = \mathrm{UL}$. More precisely we are able to show a characterization of ModL in Theorem 3.23 that if we assume $\mathrm{NL} = \mathrm{UL}$ and we have a language $L \subseteq \Sigma^*$ with $L \in$ ModL then we can decide whether an input $x \in \Sigma^*$ is in L using $f \in$ ♯L and $g \in$ FL where $g(x)$ is a prime number p that is output by g in the unary representation such that if $x \in L$ then $f(x) \equiv 1 (\mathrm{mod}\, p)$ and if $x \notin L$ then $f(x) \equiv 0 (\mathrm{mod}\, p)$. As an immediate consequence of Theorem 3.23, we show that ModL is closed under complement assuming $\mathrm{NL} = \mathrm{UL}$ in Corollary 3.24.

LEMMA 3.22. *Let $L \in$ ModL. There exists a function $f \in$ ♯L and a function $g \in$ FL such that on any input string x,*

- *$g(x) = 0^{p^e}$ for some prime p and a positive integer e, and*
- *$x \in L$ if and only if $f(x) \not\equiv 0 (mod\ p^e)$.*

PROOF. Let $L \in$ ModL be witnessed by functions $f' \in \mathrm{GapL}$ and $g \in$ FL as in Definition 3.16. Now given $f' \in \mathrm{GapL}$ there exists $f_1, f_2 \in$ ♯L such that on any input string x we have $f'(x) = f_1(x) - f_2(x)$. Consider $f(x) = f_1(x) + (p^e - 1)f_2(x)$. Since ♯L is closed under multiplication by a FL function that outputs a positive integer, and also under addition, we have $f(x) \in$ ♯L. Moreover on a given input x, we have $f'(x) \not\equiv 0 (\mathrm{mod}\, p^e)$ if and only if $f'(x) = f_1(x) - f_2(x) \not\equiv 0 (\mathrm{mod}\, p^e)$ if and only if $f(x) = f_1(x) + (p^e - 1)f_2(x) \not\equiv 0 (\mathrm{mod}\, p^e)$. As a result we can replace the GapL function f' by the ♯L function f to decide if any given input string x is in L. □

THEOREM 3.23. *Assume that $\mathrm{NL} = \mathrm{UL}$ and let $L \in$ ModL. There exists a function $f \in$ ♯L and a function $g \in$ FL such that on any input string x,*

- *$g(x) = 0^p$ for some prime $p > 0$, and,*
- *if $x \in L$ then $f(x) \equiv 1 (mod\ p)$,*
- *if $x \notin L$ then $f(x) \equiv 0 (mod\ p)$.*

PROOF. Let $L \in$ ModL. It follows from Lemma 3.22 that there exists $f' \in$ ♯L and $g' \in$ FL such that on any input string x we have $g'(x) = 0^{p^e}$ for some prime p and a positive integer e, and $x \in L$ if and only if $f'(x) \not\equiv 0 (\mathrm{mod}\, p^e)$. Let g be a FL function that outputs the prime p in unary when given the input x. Now assume that there exists a $f'' \in$ ♯L such that $f'(x) \not\equiv 0 (\mathrm{mod}\, p^e)$ if and only if $f''(x) \not\equiv 0 (\mathrm{mod}\, p)$. Then $x \in L$ if and only if $f''(x) \not\equiv 0 (\mathrm{mod}\, p)$. Define $f(x) = (f''(x))^{(p-1)}$. Using Fermat's Little Theorem we have, if $x \in L$ then

$f(x) \equiv 1 (\mathrm{mod}\ p)$. Otherwise if $x \notin L$ then $f(x) \equiv 0 (\mathrm{mod}\ p)$. We therefore prove the theorem statement if we define the function f'' such that $f'(x) \not\equiv 0 (\mathrm{mod}\ p^e)$ if and only if $f''(x) \not\equiv 0 (\mathrm{mod}\ p)$.

It is easy to see that we can compute the largest power of p that divides $p^e = |g'(x)|$ in FL (for this, we simply iteratively compute $\lfloor \frac{|g'(x)|}{p} \rfloor$ until this value is 0, which is in $\mathrm{U_L}$-NC^1 as stated in Theorem 1.31).

If $e = 1$ then we define $f'' = f'$ where f' is the $\sharp\mathrm{L}$ function in Lemma 3.22. It is clear that on an input string x we have $g(x) = g'(x) = 0^p$ for some prime p and $x \in L$ if and only if $f'(x) \not\equiv 0 (\mathrm{mod}\ p^e)$ which is true if and only if $f''(x) \not\equiv 0 (\mathrm{mod}\ p)$.

Otherwise $e \geq 2$. Here we have $g'(x) = 0^{p^e}$ and $g(x) = 0^p$. Now we use induction on $(e-1)$ to define f''. Inductively assume that for $1 \leq i \leq (e-1)$ we have functions $f_i \in \sharp\mathrm{L}$ such that $f'(x) \not\equiv 0 (\mathrm{mod}\ p^i)$ if and only if $f_i(x) \not\equiv 0 (\mathrm{mod}\ p)$. For the case when $(e-1) = 1$ we can have $f_{e-1} = f'$. Then it is clear that given an input string x, we have $f'(x)$ is divisible by p^e if and only if

(1) $f'(x)$ is divisible by p^{e-1}, and
(2) the coefficient of p^{e-1} in the base-p expansion of $f'(x)$ is zero.

Here, using Corollary 3.11 we get that, condition (2) is equivalent to $\binom{f'(x)}{p^{e-1}} \equiv 0 (\mathrm{mod}\ p)$. Therefore $f'(x) \not\equiv 0 (\mathrm{mod}\ p^e)$ if and only if $\{f'(x) \not\equiv 0 (\mathrm{mod}\ p^{e-1})$ or $\binom{f'(x)}{p^{e-1}} \not\equiv 0 (\mathrm{mod}\ p)\}$ which is true if and only if $\{f_{e-1}(x) \not\equiv 0 (\mathrm{mod}\ p)$ or $\binom{f'(x)}{p^{e-1}} \not\equiv 0 (\mathrm{mod}\ p)\}$. Let $f'_e(x) = (f_{e-1}(x))^{p-1} \binom{f'(x)}{p^{e-1}}^{p-1} + ((f_{e-1}(x))^{p-1} + p - 1) \binom{f'(x)}{p^{e-1}}^{p-1} + (f_{e-1}(x))^{p-1} (\binom{f'(x)}{p^{e-1}}^{p-1} + (p-1))$.

Now define $f_e(x) = (f'_e(x))^{p-1}$. Using Theorem 2.81 and closure properties of $\sharp\mathrm{L}$ it follows that $f_e \in \sharp\mathrm{L}$. Moreover on an input x, if $f'(x) \equiv 0 (\mathrm{mod}\ p^e)$ then $f_e(x) \equiv 0 (\mathrm{mod}\ p)$. On the other hand if $f'(x) \not\equiv 0 (\mathrm{mod}\ p^e)$ then we consider the following two cases.
case 1: $f'(x) \equiv 0 (\mathrm{mod}\ p^{e-1})$: Then $f_{e-1}(x) \equiv 0 (\mathrm{mod}\ p)$ and we also have $\binom{f'(x)}{p^{e-1}} \not\equiv 0 (\mathrm{mod}\ p)$. As a result from the definition of $f_e(x)$ we get $f_e(x) \equiv 1 (\mathrm{mod}\ p)$.
case 2: $f'(x) \not\equiv 0 (\mathrm{mod}\ p^{e-1})$: Then $f_{e-1}(x) \not\equiv 0 (\mathrm{mod}\ p)$ and we can have either $\binom{f'(x)}{p^{e-1}} \not\equiv 0 (\mathrm{mod}\ p)$ or $\binom{f'(x)}{p^{e-1}} \equiv 0 (\mathrm{mod}\ p)$. However in both of these cases we get $f_e(x) \equiv 1 (\mathrm{mod}\ p)$. As a result defining $f'' = f_e$ we complete the proof. □

COROLLARY 3.24. *Assume that* NL = UL. ModL *is closed under complement.*

PROOF. Let $L \in$ ModL. Then by Theorem 3.23 there exists $f \in \sharp\mathrm{L}$ and $g \in$ FL such that on any input x, we have $g(x) = 0^p$ for some prime p and if $x \in L$ then $f(x) \equiv 1 (\mathrm{mod}\ p)$. Otherwise if $x \notin L$ then $f(x) \equiv 0 (\mathrm{mod}\ p)$.

Let $h(x) = (f(x) + (p-1))^{(p-1)}$. Using closure properties of $\sharp\mathrm{L}$ it follows that $h(x) \in \sharp\mathrm{L}$. Using Fermat's Little Theorem we have, if $x \in L$ then $h(x) \equiv 0 (\mathrm{mod}\ p)$ and if $x \notin L$ then $h(x) \equiv 1 (\mathrm{mod}\ p)$. Clearly this shows $\overline{L} \in$ ModL or that ModL is closed under complement. □

3.2. Closure properties of ModL

As a consequence of the characterization of ModL that we have shown in Theorem 3.23, it is shown in Corollary 3.24 that ModL is closed under complement under the assumption that $\mathrm{NL} = \mathrm{UL}$. We continue to study the complexity class ModL and obtain its closure properties which follow due to the above stated characterization of ModL.

3.2.1. An unconditional closure property of ModL. We recall the definition of the $\vee$ (join) operation from Definition 3.5.

THEOREM 3.25. *If* $\{0,1\} \subseteq \Sigma$ *and* $L_1 \in \mathrm{ModL}$ *then* $L_1 \vee L_2 \in \mathrm{ModL}$.

PROOF. Let $L_i \in \mathrm{ModL}$ and assume that we decide if an input string $x \in \Sigma^*$ is in L_i using functions $f_i \in \mathrm{GapL}$ and $g_i \in \mathrm{FL}$ where $1 \leq i \leq 2$ respectively. We assume without loss of generality that the size of any input string x is at least 1. Let $x = x_1 x_2 \cdots x_{n+1}$ and let $y = x_1 \cdots x_n$ where $n \in \mathbb{Z}^+$. Also let

$$f(x) = \begin{cases} f_1(y) & \text{if } x_{n+1} = 1 \\ f_2(y) & \text{otherwise if } x_{n+1} = 0, \end{cases}$$

and

$$g(x) = \begin{cases} g_1(y) & \text{if } x_{n+1} = 1 \\ g_2(y) & \text{otherwise if } x_{n+1} = 0. \end{cases}$$

It is easy to note that $f \in \mathrm{GapL}$ and $g \in \mathrm{FL}$. Since $|g_1(x)|$ or $|g_2(x)|$ is always a prime power on any input string $x \in \Sigma^*$ it follows that $|g(x)|$ is also a prime power. Now $f(x) \not\equiv 0 (\mathrm{mod}\ |g(x)|)$ for any input $x = yx_{n+1}$ if and only if exactly one of the following is true: ($x_{n+1} = 1$ and $y \in L_1$) or ($x_{n+1} = 0$ and $y \in L_2$). Clearly this shows that $L_1 \vee L_2 \in \mathrm{ModL}$ and that we can decide if any input string $x \in \Sigma^*$ is in $L_1 \vee L_2$ using $f \in \mathrm{GapL}$ and $g \in \mathrm{FL}$ which completes the proof. □

In Theorem 3.31(1) we show that ModL is closed under logspace many-one reductions ($\leq_m^{\mathrm{L}}$) and similarly in Theorem 3.31(2) we show that ModL is closed under unambiguous logspace many-one reductions ($\leq_m^{\mathrm{UL}}$). Using Theorem 3.23 and Corollary 3.24 we show in Theorem 3.32(1) that ModL is closed under logspace truth-table reductions that make one query to the ModL oracle ($\leq_{1-\mathrm{tt}}^{\mathrm{L}}$) assuming $\mathrm{NL} = \mathrm{UL}$. We also show that if $\mathrm{NL} = \mathrm{UL}$ then ModL is closed under unambiguous logspace many-one reductions that make one query to the oracle ($\leq_{1-\mathrm{tt}}^{\mathrm{UL}}$) in Theorem 3.32(2). The following table lists the closure properties of ModL that are known.

As a consequence of these results we show in Corollary 3.33 that $\mathrm{UL}_{1-\mathrm{tt}}^{\mathrm{ModL}} = \mathrm{ModL}$ assuming $\mathrm{NL} = \mathrm{UL}$.

3.2.2. Reducibilities of ModL. We recall the Definitions 1.39, 1.40 and 2.21 stated in Section 2.2 of Chapter 2.

DEFINITION 3.26. Let Σ be the input alphabet and let $L_1, L_2 \subseteq \Sigma^*$. We say that L_1 is logspace 1-truth-table reducible to L_2 and denoted by $L_1 \leq_{1-\mathrm{tt}}^{\mathrm{L}} L_2$ if there exists a $O(\log n)$ space bounded Turing machine M^{L_2} that has access to L_2

Closure property	*Assumptions*	*Reference*
$\vee$ (join)	unconditional	Theorem 3.25
complement	NL = UL	Corollary 3.24
$\leq_m^{\mathrm{L}}$	unconditional	Theorem 3.31(1)
$\leq_m^{\mathrm{UL}}$	unconditional	Theorem 3.31(2)
$\leq_{1-\mathrm{tt}}^{\mathrm{L}}$	NL = UL	Theorem 3.32(1)
$\leq_{1-\mathrm{tt}}^{\mathrm{UL}}$	NL = UL	Theorem 3.32(2)

TABLE 3.1. Closure properties of ModL

as an oracle such that M^{L_2} decides if any given input $x \in \Sigma^*$ is in L_1 by making exactly one query to the oracle L_2, where $n = |x|$.

DEFINITION 3.27. Let Σ be the input alphabet and let $L_1, L_2 \subseteq \Sigma^*$. We say that $L_1 \leq_m^{\mathrm{UL}} L_2$ if there exists a FNL Turing machine M such that on any input $x \in \Sigma^*$ the number of accepting computation paths of $M(x)$ is at most 1. Also if $x \in L_1$ then there exists an accepting computation path of M on input x such that if we obtain $y \in \Sigma^*$ as the output at the end of the accepting computation path then $y \in L_2$. If $x \notin L_1$ then either M does not have any accepting computation path on input x or if there exists an accepting computation path of M on input x and $y \in \Sigma^*$ is the output at the end of this computation path then $y \notin L_2$.

DEFINITION 3.28. Let Σ be the input alphabet and let $L_1, L_2 \subseteq \Sigma^*$. We say that $L_1 \leq_{1-\mathrm{tt}}^{\mathrm{UL}} L_2$ if there exists a NL-Turing machine M^{L_2} with access to the oracle L_2 such that on any input $x \in \Sigma^*$ the number of accepting computation paths of $M^{L_2}(x)$ is at most 1. Also M^{L_2} submits exactly one query $q_x \in \Sigma^*$ to the L_2 oracle on any input $x \in \Sigma^*$ to decide if $x \in L_1$. We assume that the query is submitted by the NL-Turing machine M^{L_2} in a deterministic manner to the L_2 oracle according to the Ruzzo-Simon-Tompa oracle access mechanism. Here if $x \in L_1$ then there exists exactly one accepting computation path of M^{L_2} on input x. Otherwise if $x \notin L_1$ then M^{L_2} rejects the input x on all of its computation paths.

We recall the definiton of ModL stated in Definition 3.16.

DEFINITION 3.29. Let Σ be the input alphabet and let $L_1, L_2 \in \Sigma^*$. We define the complexity class $\mathrm{UL}_{1-\mathrm{tt}}^{\mathrm{ModL}} = \{L_1 | L_1 \leq_{1-\mathrm{tt}}^{\mathrm{UL}} L_2, \text{where } L_2 \in \mathrm{ModL}\}$.

LEMMA 3.30. *Let Σ be the input alphabet and let $g \in$ FL such that on any input $x \in \Sigma^*$, we have $g(x)$ to be a positive integer p in the unary notation that depends on the input x. There exists a* NL*-Turing machine M such that the lexicographically least $|g(x)|$ computation paths are exactly the only accepting computation paths in the computation tree of M.*

PROOF. Let us consider the following algorithm of a non-deterministic Turing machine given an input $x \in \Sigma^*$.

Algorithm 10 Lex-Least-accept-FL$_g$
Input: $x \in \Sigma^*$.
Output: *accept* or *reject.* The NL-Turing machine M that implements this algorithm is such that the lexicographically least $|g(x)|$ computation paths are the only accepting computation paths of $M(x)$ and so $acc_M(x) = |g(x)|$.
Complexity: NL.

1: Compute $p = |g(x)|$.
2: Compute k such that $2^{k-1} \leq p < 2^k$.
3: Non-deterministically choose k bits from $\{0, 1\}$ that describes a computation path of length k in the computation tree of depth k. Let m be the number describing the computation path that is chosen.
4: **if** $m \leq p$ **then**
5: accept and stop.
6: **else**
7: reject and stop.
8: **end if**

Since $g(x) = 0^p$ for some positive integer p and $k \in \mathbb{Z}^+$ such that $2^{k-1} \leq p < 2^k$, it is easy to see that $k, p \in O(\log n)$ As a result it is easy to note that the above algorithm can be implemented by a non-deterministic Turing machine that uses at most $O(\log n)$ space where $n = |x|$ and $x \in \Sigma^*$ is the input. Let M be the NL-Turing machine that simulates the above stated algorithm on input x. Clearly, the NL-Turing machine M non-deterministically makes k choices and accepts in a computation path if the non-deterministic choices form a binary string less than or equal to p. M rejects at the end of all other computation paths which proves our result. □

THEOREM 3.31. *Let Σ be the input alphabet and let $L_1, L_2 \subseteq \Sigma^*$. Also let $L_2 \in$ ModL. Now*

(1) *if $L_1 \leq_m^{\mathrm{L}} L_2$ then $L_1 \in$ ModL, and*
(2) *if $L_1 \leq_m^{\mathrm{UL}} L_2$ then $L_1 \in$ ModL.*

PROOF. (1) Let $L_2 \in$ ModL and let $f_2 \in$ GapL and $g_2 \in$ FL be functions using which we decide if an input $x \in \Sigma^*$ is in L_2. Also let $L_1 \leq_m^{\mathrm{L}} L_2$ using $f \in$ FL. If $x \in \Sigma^*$ is the input then $x \in L_1$ if and only if $f(x) \in L_2$. However $L_2 \in$ ModL and therefore $f(x) \in L_2$ if and only if $f_2(f(x)) \not\equiv 0 (\mathrm{mod}\ p^e)$ where $p^e = |g_2(f(x))| \in \mathbb{Z}^+$ where $p, e \in \mathbb{Z}^+$ with p being a prime and $e \geq 1$. Now it follows from Lemma 2.56 that $(f_2 \circ f) \in$ GapL and $(g_2 \circ f) \in$ FL. Clearly $g_2(f(x))$ is a prime power in the unary representation for any $x \in \Sigma^*$. As a result if $x \in \Sigma^*$ then $x \in L_1$ if and only if $f_2(f(x)) \not\equiv 0 (\mathrm{mod}\ p^e)$ where $p^e = |g_2(f(x))|$ which shows $L_1 \in$ ModL.

(2) We recall the definition $\leq_m^{\mathrm{UL}}$ from Definition 3.27. The proof of this assertion is similar to Theorem 3.31(1). We use Lemma 3.22 for $L_2 \in$ ModL and assume that there exists $f \in \sharp\mathrm{L}$ and $g \in$ FL such that on

any input $x \in \Sigma^*$ we have $g(x) = 1^{p^e}$ where $p, e \in \mathbb{Z}^+$, p is a prime, $e \geq 1$ and $x \in L_2$ if and only if $f(x) \not\equiv 0 (\text{mod } p^e)$. Let us denote the $\leq_m^{\text{UL}}$ reduction by h. Clearly, h has at most one accepting computation path. Let us now consider the following algorithm implemented by a non-deterministic Turing machine given an input $x \in \Sigma^*$.

Algorithm 11 UL-many-one-ModL
Input: $x \in \Sigma^*$.
Output: *accept* or *reject.* The NL-Turing machine that implements this algorithm obeys the property of ModL in Lemma 3.22.
Complexity: ModL.

1: Simulate the NL-Turing machine M_h corresponding to the FUL function h on the input x.
2: **if** the computation path of $M_h(x)$ ends in an accepting configuration **then**
3: Let $y = M_h(x)$.
4: **else**
5: Lex-Least-accept-FL$_g(x)$ and stop.
6: **end if**
7: Simulate $M(y)$ and stop.

It is easy to see that this algorithm can be implemented by a non-deterministic Turing machine that uses $O(\log n)$ space, where $n = |x|$. In the above algorithm, upon simulating M_h on input x, if the computation path that we obtain is an accepting computation path and $y \in \Sigma^*$ is the output at the end of the accepting computation path, then simulating the NL-Turing machine for $L_2 \in$ ModL with y as the input shows that the congruence relation based on the $\sharp$L function f and the FL function g that is used to decide if any input $x \in \Sigma^*$ is in L_2 can also be used for deciding if $x \in L_1$. Otherwise if the computation path of the $\leq_m^{\text{UL}}$ reduction that we obtain is a rejecting computation path then using the non-deterministic algorithm stated in Lemma 3.30, it follows that our NL-Turing machine can make sufficiently many non-deterministic choices so that the number of accepting computation paths that we obtain at the end of each of these rejecting computation paths is divisible by $|g(x)|$. This shows $L_1 \in$ ModL.

□

THEOREM 3.32. *Let Σ be the input alphabet and let $L_1, L_2 \subseteq \Sigma^*$. Also let $L_2 \in$ ModL and assume that* NL $=$ UL*.*

(1) *If $L_1 \leq_{1-\text{tt}}^{\text{L}} L_2$ then $L_1 \in$ ModL, and*
(2) *If $L_1 \leq_{1-\text{tt}}^{\text{UL}} L_2$ then $L_1 \in$ ModL.*

PROOF. (1) We recall the definition of $\leq_{1-\text{tt}}^{\text{L}}$ from Definition 3.26. Let $L_1 \leq_{1-\text{tt}}^{\text{L}} L_2$ using $f \in$ FL that makes exactly one query to the oracle

L_2. In other words, f correctly decides if an input $x \in \Sigma^*$ is in L_1 by making exactly one query to the oracle L_2. Also let $q_x \in \Sigma^*$ be the query string that is generated by f for the input $x \in \Sigma^*$. Clearly f decides if $x \in L_1$ based on x and the reply of the oracle L_2 when the oracle L_2 is given the query q_x as input. Therefore let $f' \in$ FL be the function that generates and outputs the query q_x generated by f when it is given the input x. Also let $g' \in$ FL be such that $g'(x, \chi_{L_2}(q_x)) = f(x)$ where χ_{L_2} is the characteristic function of L_2. It is clear that g' correctly decides whether any input $x \in L_1$ if the reply of the oracle L_2 is also given to it. Since $L_2 \in$ ModL and we have assumed that NL = UL, it follows from Theorem 3.23 that there exists $f_2 \in \sharp$L and $g_2 \in$ FL such that on any input $x \in \Sigma^*$ we have if $x \in L_2$ then $f_2(x) \equiv 1 (\text{mod } p)$ and if $x \notin L_2$ then $f_2(x) \equiv 0 (\text{mod } p)$, where $g_2(x) = 1^p$, $p \in \mathbb{Z}^+$ and p is a prime. Let us consider the function $(g_2 \circ f')$. Given an input $x \in \Sigma^*$, $g_2(f(x)) = 1^p$, for some prime $p \in \mathbb{Z}^+$ which depends on the query $f'(x)$ generated from the input x. It is clear that $(g_2 \circ f') \in$ FL. Consider the function $(g' \circ f_2 \circ f')$ and let M denote the NL-Turing machine corresponding to this function. The NL-Turing machine M considers the output of a computation path of the NL-Turing machine corresponding to $(f_2 \circ f')(x)$ as $\chi_{L_2}(q_x)$. Since $g' \in$ FL it follows that the output of M is the same at the end of all the accepting computation paths of the NL-Turing machine corresponding to $(f_2 \circ f')$. In other words, either M accepts at the end of all the accepting computation paths of the NL-Turing machine corresponding to $(f_2 \circ f')$ or M rejects at the end of all the accepting computation paths of the NL-Turing machine corresponding to $(f_2 \circ f')$. Similarly either M accepts at the end of all the rejecting computation paths of the NL-Turing machine corresponding to $(f_2 \circ f')$ or M rejects at the end of all the rejecting computation paths of the NL-Turing machine corresponding to $(f_2 \circ f')$. Let us now consider the following algorithm.

Algorithm 12 L-1-truth-table-ModL
Input: $x \in \Sigma^*$.
Output: *accept* or *reject.* The NL-Turing machine M' that implements this algorithm obeys the property of ModL in Theorem 3.23.
Complexity: ModL.

Compute $p = |g_2(f'(x))|$.
$counter = 0$.
$lex\_least\_flag =$ True.
while $counter < p - 1$ **do**
    Simulate the NL-Turing machine M on input x as follows.
    if $M(x)$ rejects at the end of its computation path **then**
        reject and stop.
    else if $M(x)$ accepts and $(f_2 \circ f')(x)$ rejects **then**
        if the computation path of $M(x)$ is not the lexicographically least **then**

Algorithm 12 L-1-truth-table-ModL (continued)

10: $lex\_least\_flag =$ False.
11: **end if**
12: **end if**
13: $counter = counter + 1$.
14: **end while**
15: **if** $lex\_least\_flag =$ True **then**
16: Lex-Least-accept-FL$_{g_2}(f'(x))$.
17: M' non-deterministically branches sufficiently many times on its computation path as in Corollary 3.24.
18: **end if**

Let $x \in \Sigma^*$ be the input and let $g_2(f'(x)) = 1^p$. In the above algorithm, if the output of M at the end of any of its computation paths is the same as the output of the computation path of the NL-Turing machine corresponding to $(f_2 \circ f')$ on the input x then M' makes $(p-1)$ times the simulation of M on input x and stops. In this case, for a given input $x \in \Sigma^*$, if $L_1 \leq^{\mathrm{L}}_{1-\mathrm{tt}} L_2$ then at the end of every computation path the reduction is similar to a logspace many-one reduction from L_1 to L_2. In this case using Fermat's Little Theorem, it follows from our observations that if $x \in L_1$ then $acc_{M'}(x) \equiv 1 (\mathrm{mod}\, p)$. However if $x \notin L_1$ then $acc_{M'}(x) \equiv 0 (\mathrm{mod}\, p)$. This shows $L_1 \in$ ModL. On the other hand if we have the output of M along any of its computation paths is the complement of the output of the computation path of the NL-Turing machine corresponding to $(f_2 \circ f')$ on a given input $x \in \Sigma^*$ then also M' makes $(p-1)$ simulations of M on input x. However in these $(p-1)$ simulations of M by M', along any of the computation paths of M' that is not the lexicographically least computation path we assume that M' stops with the output of M on input x. Along the lexicographically least computation path that we obtain in these $(p-1)$ simulations of M by M' we assume that M' makes sufficiently many non-deterministic choices at the end of this computation path as in Lemma 3.30 so that we obtain $acc_{M'}(x) = acc_M(x)^{p-1} + (p-1)$. In this case using Fermat's Little Theorem, it follows from our observations that if $x \in L_1$ then $acc_{M'}(x) \equiv 0 (\mathrm{mod}\, p)$ and if $x \notin L_1$ then $acc_{M'}(x) \not\equiv 0 (\mathrm{mod}\, p)$ which implies $\overline{L_1} \in$ ModL. We now use our assumption that NL = UL and so it follows from Corollary 3.24 that $L_1 \in$ ModL and this completes the proof.

(2) We recall the definition of $\leq^{\mathrm{UL}}_{1-\mathrm{tt}}$ from Definition 3.28. We follow the Ruzzo-Simon-Tompa oracle access mechanism stated in Section 1.1.6 and generate the query string $q_x \in \Sigma^*$ given the input $x \in \Sigma^*$ in a deterministic manner using $O(\log n)$ space. Our proof follows from our assumption that NL = UL and a case analysis similar to the proof of Theorem 3.32(1) depending on whether the $\leq^{\mathrm{UL}}_{1-\mathrm{tt}}$ reduction is similar to

a $\leq_m^{\mathrm{UL}}$ reduction or if the output of the computation paths of the NL-Turing machine that is used in the reduction is the complement of the output of the computation paths of the NL-Turing machine for the oracle L_2 on the query string q_x.

□

COROLLARY 3.33. *Assume that* $\mathrm{NL} = \mathrm{UL}$. $\mathrm{UL}_{1-\mathrm{tt}}^{\mathrm{ModL}} = \mathrm{ModL}$.

PROOF. Since we know that NL is closed under complement from Theorem 2.18, our assumption that $\mathrm{NL} = \mathrm{UL}$ implies $\mathrm{UL} =$ co-UL. Proof of our result now follows from Theorem 3.32(2). □

3.3. Relations among Modulo-based

DEFINITION 3.34. Let Σ be the input alphabet and let $L_1, L_2 \subseteq \Sigma^*$. We say that L_1 is logspace conjunctive truth-table reducible to L_2, denoted by $L_1 \leq_{\mathrm{ctt}}^{\mathrm{L}} L_2$, if there exists $f \in \mathrm{FL}$ such that given an input $x \in \Sigma^*$ of length n we have $f \in \mathrm{FL}$ such that $f(x) = \{y_1 \ldots, y_{p(n)}\}$ and $x \in L_1$ if and only if $y_i \in L_2$ for every $1 \leq i \leq p(n)$ where $p(n)$ is a polynomial in n.

DEFINITION 3.35. Let Σ be the input alphabet and let $L_1, L_2 \subseteq \Sigma^*$. We say that L_1 is logspace k-conjunctive truth-table reducible to L_2, denoted by $L_1 \leq_{k-\mathrm{ctt}}^{\mathrm{L}} L_2$, if there exists $f \in \mathrm{FL}$ such that given an input $x \in \Sigma^*$ of length n $f(x) = \{y_1 \ldots, y_k\}$ and $x \in L_1$ if and only if $y_i \in L_2$ for every $1 \leq i \leq k$.

DEFINITION 3.36. Let Σ be the input alphabet and let $L_1, L_2 \subseteq \Sigma^*$. We say that L_1 is logspace disjunctive truth-table reducible to L_2, denoted by $L_1 \leq_{\mathrm{dtt}}^{\mathrm{L}} L_2$, if there exists $f \in \mathrm{FL}$ such that given an input $x \in \Sigma^*$ of length n we have $f(x) = \{y_1, \ldots, y_{p(n)}\}$ and $x \in L_1$ if and only if $y_i \in L_2$ for at least one $1 \leq i \leq p(n)$ where $p(n)$ is a polynomial in n.

DEFINITION 3.37. Let Σ be the input alphabet and let $L_1, L_2 \subseteq \Sigma^*$. We say that L_1 is logspace k-disjunctive truth-table reducible to L_2 for some $k \in \mathbb{Z}^+$, $k \geq 1$, denoted by $L_1 \leq_{k-\mathrm{dtt}}^{\mathrm{L}} L_2$, if there exists $f \in \mathrm{FL}$ such that given an input $x \in \Sigma^*$ of length n we have $f(x) = \{y_1, \ldots, y_k\}$ and $x \in L_1$ if and only if $y_i \in L_2$ for at least one $1 \leq i \leq k$.

It follows from the definition of ModL that if $p \in \mathbb{Z}^+$ is a prime then $\mathrm{Mod}_p\mathrm{L} \subseteq \mathrm{ModL}$. However if $k \in \mathbb{Z}^+$ such that $k \geq 6$ is a composite number that has more than one distinct prime divisor then it is not known if $\mathrm{Mod}_k\mathrm{L} \subseteq \mathrm{ModL}$. We show that if ModL is closed under $\leq_{l-\mathrm{dtt}}^{\mathrm{L}}$ reductions for some $l \in \mathbb{Z}^+$ such that $l \geq 2$ then $\mathrm{Mod}_k\mathrm{L} \subseteq \mathrm{ModL}$ for all $k \in \mathbb{Z}^+$ such that $k \geq 6$ is a composite number that has at most l distinct prime divisors.

THEOREM 3.38. *If* ModL *is closed under* $\leq_{l-\mathrm{dtt}}^{\mathrm{L}}$ *reductions where* $l \in \mathbb{Z}^+$ *and* $l \geq 2$ *then* $\mathrm{Mod}_k\mathrm{L} \subseteq \mathrm{ModL}$ *for all* $k \in \mathbb{Z}^+$ *such that* $k \geq 6$ *is a composite number that has at least* 2 *and at most* l *distinct prime divisors.*

PROOF. Let us assume that ModL is closed under $\leq^{\mathrm{L}}_{l-\mathrm{dtt}}$ reductions for some $l \in \mathbb{Z}^+$ such that $l \geq 2$. Let Σ be the input alphabet and let $\sharp \notin \Sigma$. Also let $L \in \Sigma^*$ and assume that $L \in \mathrm{Mod}_k\mathrm{L}$, where $k = p_1^{e_1} \cdots p_m^{e_m}$ is a composite number such that p_i is a prime, $e_i \in \mathbb{Z}^+$ with $e_i \geq 1$ and $p_i \neq p_j$ for all $1 \leq i < j \leq m \leq l$.

Using Lemma 3.22, it follows from the definition of $\mathrm{Mod}_k\mathrm{L}$, GapL, $\sharp$L and Proposition 2.16 that determining if the number of st-paths in an instance (G, s, t) of SLDAG is $\not\equiv 0 (\mathrm{mod}\ k)$ is logspace many-one complete for $\mathrm{Mod}_k\mathrm{L}$. Let us denote this problem by Mod$_k st$Path. Also it is easy to infer that for any function $f \in \sharp\mathrm{L}$, we have $f(x) \not\equiv 0 (\mathrm{mod}\ k)$ if and only if $f(x) \not\equiv 0 (\mathrm{mod}\ p_i)$ for at least one of the primes $p_i | k$, where $1 \leq i \leq m$. Based on these observations let us define $L_k = \{\langle x\sharp 1^{p_i}\rangle | f(x) \not\equiv 0 (\mathrm{mod}\ |g(\langle x\sharp 1^{p_i}\rangle)|)\ \forall\ p_i | k$ where $g \in$ FL and $g(\langle x\sharp 1^{p_i}\rangle) = 1^{p_i}\}$. As a result, it follows from Definition 3.16 that $L_k \in$ ModL.

Therefore if $x \in \Sigma^*$ is the input then $x \in L$ if and only if $\langle x\sharp 1^{p_i}\rangle \in L_k$ for at least one prime p_i where $1 \leq i \leq m$. However it is easy to note that given x as input, a $O(\log |x|)$ space bounded Turing machine M can obtain the prime factorization of k and also output $\langle x\sharp 1^{p_i}\rangle$ for all primes $p_i | k$ where $1 \leq i \leq m$. If $m < l$ then we assume that M outputs the last query $\langle x\sharp 1^{p_j}\rangle$ that it generates with repetition sufficiently many times to output l strings where $1 \leq j \leq m \leq l$. Now $x \in L$ if and only if $\langle x\sharp 1^{p_j}\rangle \in L_k$ for at least one $1 \leq j \leq l$. However we have assumed that ModL is closed under $\leq^{\mathrm{L}}_{l-\mathrm{dtt}}$ reductions which implies $L \in$ ModL. This shows that $\mathrm{Mod}_k\mathrm{L} \subseteq$ ModL whenever $k \geq 6$ is a composite number such that the number of distinct prime divisors of k is at least 2 and at most l. □

THEOREM 3.39. *If* ModL *is closed under* $\leq^{\mathrm{L}}_{\mathrm{dtt}}$ *reductions then* co-C$_=$L $\subseteq$ ModL.

PROOF. Let us assume that ModL is closed under $\leq^{\mathrm{L}}_{\mathrm{dtt}}$ reductions. Recalling the definition of Mod$_k st$Path from the proof of Theorem 3.38, we know that Mod$_k st$Path is a canonical complete language for ModL under logspace many-one reductions. Also given $x \in \Sigma^*$ as an input, the problem of determining if $f(x) \neq 0$ is complete for co-C$_=$L under logspace many-one reductions, where f is a logspace many-one complete function for $\sharp$L. Using the Chinese Remainder Theorem we know that the $\det(A)$ is uniquely determined by its residues modulo all the primes from 2 and n^4. Therefore $f(x) \neq 0$ if and only if $f(x) \not\equiv 0 (\mathrm{mod}\ p)$ for some prime $2 \leq p \leq n^4$.

Similar to the proof of Lemma 3.38 it is easy to note that there exists a $O(\log n)$ space bounded Turing machine M that when given the input string x as input computes all the primes from 2 to n^4 and also outputs the pairs $\langle x\sharp 1^{p_i}\rangle$ for all of the primes from 2 to n^4. Clearly $f(x) \neq 0$ if and only if $\langle x\sharp 1^{p_i}\rangle \in \mathrm{Mod}_k st$Path for at least one of the primes $p \in \{2, \ldots, n^4\}$. However we have assumed that ModL is closed under $\leq^{\mathrm{L}}_{\mathrm{dtt}}$ reductions from which it follows that we can determine if $f(x) \neq 0$ in ModL and this implies co-C$_=$L $\subseteq$ ModL. □

COROLLARY 3.40. *Assume that* NL $=$ UL. *If* ModL *is closed under* $\leq^{\mathrm{L}}_{\mathrm{dtt}}$ *reductions then* C$_=$L $\subseteq$ ModL.

PROOF. Proof follows from Theorem 3.39 and 3.24. □

LEMMA 3.41. *Assume that* $\mathrm{NL} = \mathrm{UL}$. *If* ModL *is closed under* $\leq^{\mathrm{L}}_{l-\mathrm{dtt}}$ *reductions where* $l \in \mathbb{Z}^+$ *and* $l \geq 2$ *then* ModL *is closed under* $\leq^{\mathrm{L}}_{l-\mathrm{ctt}}$ *reductions.*

PROOF. Let Σ be the input alphabet and let $\sharp \notin \Sigma$. Also let $L_1, L_2 \subseteq \Sigma^*$ and let $L_2 \in$ ModL be such that $L_1 \leq^{\mathrm{L}}_{l-\mathrm{ctt}} L_2$ using a function $f \in$ FL. Therefore if $x \in \Sigma^*$ is the input then we have $f(x) = \langle y_1 \sharp \cdots \sharp y_l \rangle$ such that $x \in L_1$ if and only if $y_i \in L_2$ for all $1 \leq i \leq l$.

However this is equivalent to $x \notin L_1$ if and only if $y_i \in \overline{L_2}$ for at least one $1 \leq i \leq l$ where $\overline{L_2}$ denotes the complement of L_2. However since we have assumed $\mathrm{NL} = \mathrm{UL}$ it follows from Corollary 3.24 that $\overline{L_2} \in$ ModL. These observations show that $\overline{L_1} \leq^{\mathrm{L}}_{l-\mathrm{dtt}} \overline{L_2}$ and since we have assumed ModL is closed under $\leq^{\mathrm{L}}_{l-\mathrm{dtt}}$ we have $\overline{L_1} \in$ ModL. Once again from our assumption that $\mathrm{NL} = \mathrm{UL}$ using the result that ModL is closed under complement shown in Corollary 3.24 it follows that $L_1 \in$ ModL. □

THEOREM 3.42. *Assume that* $\mathrm{NL} = \mathrm{UL}$. *If* ModL *is closed under* $\leq^{\mathrm{L}}_{\mathrm{dtt}}$ *reductions then* ModL *is closed under* $\leq^{\mathrm{L}}_{\mathrm{ctt}}$ *reductions.*

PROOF. Proof of this result is similar to the proof of Lemma 3.41. We need to observe that the number of instances of L_2 that can be output by a $\leq^{\mathrm{L}}_{\mathrm{dtt}}$ reduction is at most a polynomial in the size of the input. Since we have assumed $\mathrm{NL} = \mathrm{UL}$ it follows from Corollary 3.24 that ModL is closed under complement and by using this property of ModL we obtain this result. □

Exercises

(1) Show that NL is closed under $\leq^{\mathrm{L}}_{\mathrm{ctt}}$ reductions without using Theorems 2.19 and 2.18 .
(2) Show that NL is closed under $\leq^{\mathrm{L}}_{\mathrm{dtt}}$ reductions without using Theorems 2.19 and 2.18.
(3) Let Σ be the input alphabet and let $A \subseteq \Sigma^*$. Define the complexity class ModL^A.
(4) Define the complexity class $\mathrm{ModL}^{\mathrm{UL}}$.
(5) Show that $\mathrm{ModL}^{\mathrm{UL}} = \mathrm{ModL}$ assuming $\mathrm{UL} = \text{co-UL}$.

Open problems

(1) Does there exist any containment relation between $\oplus$L and NL? Is $(\oplus \mathrm{L} \Delta \mathrm{NL})$ non-empty?
(2) Is $\mathrm{Mod}_k\mathrm{L} \subseteq$ ModL if $k > 0$ is a composite number?
(3) Is $\mathrm{Mod}_k\mathrm{LH} = \mathrm{Mod}_k\mathrm{L}$, where $k \in \mathbb{N}$ and k is a composite?

Notes

Modulo-based logarithmic space bounded counting classes $\mathrm{Mod}_k\mathrm{L}$, where $k \in \mathbb{Z}$ and $k \geq 2$, were by defined Gerhard Buntrock, Carsten Damm, Ulrich Hertrampf and Christoph Meinel in [**BDH$^+$92**]. All the results in this chapter prior to Section 3.1 have been shown in the polynomial-time setting by Richard Beigel and John Gill in [**BG92**] for complexity classes $\mathrm{Mod}_k\mathrm{P}$ and $\mathrm{Mod}_p\mathrm{P}$, where $k, p \in \mathbb{N}$, $k, p \geq 2$ and p is a prime. Theorems 3.7 and 3.8 are due to Ulrich Hertrampf, Steffan Reith and Heribert Vollmer [**HRV00**]. The remaining results starting from Proposition 3.9 to Corollary 3.15 are from [**BDH$^+$92**].

The modulo-based logarithmic space bounded counting class ModL was defined by V. Arvind and T. C. Vijayaraghavan in [**AV10, Vij08**] to tightly classify the complexity of the problem called LCON which is to solve a system of linear equations modulo a composite number k, where k is given in terms of its prime factorization such that every distinct prime power divisor that occurs in the prime factorization of k is given in the unary representation. In [**AV10**], it is shown that LCON$\in \mathrm{L}^{\mathrm{ModL}}/\mathrm{poly}$ and in BP.NC^2. Along with LCON, some more problems on linear congruences modulo a composite number k which is given as a part of the input in terms of its prime factorization, where every prime power is given in unary representation, and a host of problems on Abelian permutation groups have been shown to be in $\mathrm{L}^{\mathrm{ModL}}/\mathrm{poly}$ and in BP.NC^2. The complexity class ModL generalizes $\mathrm{Mod}_k\mathrm{L}$, for any $k \in \mathbb{Z}$ and $k \geq 2$ since the moduli varies with the input. A very useful observation on ModL which uses the Chinese Remainder Theorem is Lemma 3.21 and it is from [**AV10**, Lemma 3.3]. Given a number n as residues modulo polynomially many primes, the result that it is possible to compute or find out n from its residues in logspace-uniform NC^1 is due to Andrew Chiu, George Davida and Bruce Litow [**CDL01**]. This result was improved by W. Hesse, David A. Mix Barrington and Eric Allender in [**HAB02**] wherein it is shown that the same operation of computing the number n from its residues modulo polynomially many primes is in DLOGTIME-TC^0.

Results on ModL in Section 3.1.1 are from [**Vij22**]. An important consequence of the characterization of ModL shown in Theorem 3.23 is that, assuming $\mathrm{NL} = \mathrm{UL}$, we are able to show that ModL is the logspace analogue of the polynomial time counting complexity class ModP defined by J. Köbler and Seinosuke Toda in [**KT96**]. Closure properties of ModL shown in Section 3.2 and results on reducibilities of ModL shown in Section 3.2.2 is from [**Vij10**].

CHAPTER 4

Probabilistic Logarithmic space bounded counting class: PL

From now onwards for the rest of this chapter, when we consider functions in GapL***, we do not necessarily assume that the computation tree of a function in*** GapL ***is a complete binary tree.***

4.1. Closure properties of PL

We recall the definition of PL from Definition 2.31.

PROPOSITION 4.1. *Let* Σ *be the input alphabet. The class* PL *consists of those languages* $L \subseteq \Sigma^*$ *such that for some* GapL *function* f *and all* $x \in \Sigma^*$

- *if* $x \in L$ *then* $f(x) > 0$
- *if* $x \notin L$ *then* $f(x) < 0$.

PROOF. For a GapL function f, the GapL function $2f(x) - 1$ has the same sign as $f(x)$ for positive and negative values of $f(x)$ and takes value -1 for $f(x) = 0$. □

COROLLARY 4.2. PL *is closed under complement.*

LEMMA 4.3. *Define*

$$P_n(x) = (x-1)\prod_{i=1}^{n}(x-2^i)^2.$$
$$A_n(x) = P_n(-x))^{\frac{n}{2}+1} - (P_n(x))^{\frac{n}{2}+1}.$$
$$B_n(x) = P_n(-x))^{\frac{n}{2}+1} + (P_n(x))^{\frac{n}{2}+1}.$$
$$S_n(x) = \tfrac{A_n}{B_n}.$$

Then, for $n \geq 1$,

(1) *If* $1 \leq x \leq 2^n$ *then* $0 \leq 4P_n(x) < -P_n(-x)$.
(2) *If* $1 \leq x \leq 2^n$ *then* $1 \leq S_n < \frac{5}{3}$.
(3) *If* $-2^n \leq x \leq -1$ *then* $-\frac{5}{3} < S_n(x) \leq -1$.

PROOF. (1) Since $x \geq 1$, we have that $P_n(x) \geq 0$. Clearly, $x-1 < x+1$, and $(x-2^i)^2 < (-x-2^i)^2$ for $i = 1, \ldots, n$. Also, if $2^{k-1} \leq x < 2^k$ then $4(x-2^k)^2 \leq 4.2^{2k-2} = 2^{2k} < (-x-2^k)^2$. Together these imply that $4P_n(x) < -P_n(-x)$.

(2) If $P_n(x) = 0$ then $S_n(x) = 1$. If $P_n(x) \neq 0$, then we can write

$$S_n(x) = 1 + \frac{2}{\left(\left(\frac{-P_n(-x)}{P_n(x)}\right) - 1\right)}.$$

Simple algebra and part (1) yield the desired result.

(3) This follows from (2) and the fact that $S_n(x)$ is an odd function, i.e., $S_n(-x) = -S_n(x)$.

□

THEOREM 4.4. PL *is closed under union.*

PROOF. Fix two languages D and E in PL. We will show that $D \cup E$ is also in PL.

Let f_D and f_E be the functions given by Proposition 4.1 for D and E, respectively. By Lemma 2.44, let $n = n(|x|)$ be a polynomial such that 2^n bounds the maximum absolute value of $f_D(x)$ and $f_E(x)$. Define $A_D(x) = A_n(f_D(x))$. Similarly, define $B_D(x), S_D(x), A_E(x), B_E(x)$, and $S_E(x)$. Note that $A_D(x), B_D(x)$, $A_E(x)$, and $B_E(x)$ are GapL functions.

Let $H(x) = S_D(x) + S_E(x) + 1$. By Lemma 4.3, we have

(1) If $x \in D$ and $x \in E$ then $H(x) \geq 3$.
(2) If $x \in D$ and $x \notin E$ then $H(x) \geq \frac{1}{3}$.
(3) If $x \notin D$ and $x \in E$ then $H(x) \geq \frac{1}{3}$.
(4) If $x \notin D$ and $x \notin E$ then $H(x) \leq -1$.

Thus we have that $x \in D \cup E$ if and only if $H(x) > 0$.

We would be finished if H were a GapL function. However unfortunately it may even take nonintegral values. We do have

$$H(x) = \frac{A_D(x)}{B_D(x)} + \frac{A_E(x)}{B_E(x)} + 1 = \frac{A_D(x)B_E(x) + A_E(x)B_D(x) + B_D(x)B_E(x)}{B_D(x)B_E(x)}.$$

Note that for nonzero integers p and q we have $p/q > 0$ if and only if $pq > 0$. Then we define

$$H'(x) = (A_D(x)B_E(x) + A_E(x)B_D(x) + B_D(x)B_E(x))(B_D(x)B_E(x)).$$

We have that

(1) $H'(x)$ is GapL function.
(2) For all $x \in \Sigma^*$, $H(x) > 0$ if and only if $H'(x) > 0$.
(3) $x \in D \cup E$ if and only if $H'(x) > 0$.

Finally applying Proposition 4.1 to H', we have $D \cup E \in$ PL. □

COROLLARY 4.5. PL *is closed under intersection.*

4.2. PL is closed under PL-Turing reductions

Similar to Definition 2.21, we define the complexity class $\mathrm{PL}^{\mathrm{PL}}$ as follows.

DEFINITION 4.6. Let Σ be the input alphabet. We define $\mathrm{PL}^{\mathrm{PL}}$ to be the complexity class of all languages $L \subseteq \Sigma^*$ for which there exists a $O(\log n)$ space bounded non-deterministic Turing machine M that has oracle access to a language $A \in \mathrm{PL}$ such that for any $x \in \Sigma^*$, the number of accepting computation paths of $M(x) \geq 2^{p(n)-1}$ if and only if $x \in L$, where $n = |x|$, $A \subseteq \Sigma^*$ and $2^{p(n)}$ is the number of accepting computation paths of M on any input of size n for some polynomial $p(n)$. Here we assume that M submits queries to the oracle A according to the Ruzzo-Simon-Tompa oracle access mechanism.

In Theorem 4.11 of this section, we show that PL is closed under PL-Turing reductions. In other words, we show that $\mathrm{PL}^{\mathrm{PL}} = \mathrm{PL}$.

DEFINITION 4.7. **(Low-Degree Polynomials to Approximate the Sign Function)** Let m and r be positive integers. Define:

$$P_m(z) = (z-1)\prod_{1\leq i\leq m}(z-2^i)^2. \tag{4.1}$$

$$Q_m(z) = -P_m(z) - P_m(-z). \tag{4.2}$$

$$A_{m,r}(z) = (Q_m(z))^{2r}. \tag{4.3}$$

$$B_{m,r}(z) = (Q_m(z))^{2r} + (2P_m(z))^{2r}. \tag{4.4}$$

$$R_m, r(z) = \left(\frac{2P_m(z)}{Q_m(z)}\right)^{2r}. \tag{4.5}$$

$$S_{m,r}(z) = (1 + R_{m,r}(z))^{-1}. \tag{4.6}$$

$R_{m,r}(z)$ and $S_{m,r}(z)$ are two auxiliary functions that will help us understand the properties of $A_{m,r}(z)$ and $B_{m,r}(z)$.

LEMMA 4.8. (1) *For all positive integers m and r, $S_{m,r}(z) = \frac{A_{m,r}(z)}{B_{m,r}(z)}$.*

(2) *For every positive integers m and r, both $A_{m,r}(z)$ and $B_{m,r}(z)$ are polynomials in z of degree $O(rm)$.*

(3) *For all integers $m, r \geq 1$ and every integer z,*
   (a) *if $1 \leq z \leq 2^m$, then $1 - 2^{-r} \leq S_{m,r}(z) \leq 1$, and*
   (b) *if $-2^m \leq z \leq -1$, then $0 \leq S_{m,r}(z) \leq 2^{-r}$.*

PROOF. The proofs of (1) and (2) are by routine calculation. We leave them to the reader. To prove (3), let m and r be positive integers. First consider the case when $1 \leq z \leq 2^m$. In this case $P_m(z) \geq 0$ and $P_m(-z) < 0$. We prove the following claim.

CLAIM 4.9. *If $1 \leq z \leq 2^m$, then $0 \leq P_m(z) < -\frac{P_m(-z)}{9}$.*

Proof of Claim. The claim clearly holds for $z = 1$. So suppose that $1 \leq z \leq 2^m$. There is a unique i, $1 \leq i \leq m$, such that $2^i \leq z < 2^{i+1}$. Let t be that i. Then (i) $2^t \leq z$ and (ii) $z/2 < 2^t$. By combining (i) and (ii) we get $0 \leq (z - 2^t) < \frac{z}{2}$, and from (ii) we get $\frac{z}{2} < |-z - 2^t|/3$. By combining the two inequalities, we have $(z - 2^t)^2 < \frac{(-z-2^t)^2}{9}$. Note that $z - 1 < z + 1$ and $|z - 2^i| \leq z + 2^i$ for every i, $1 \leq i \leq m$. Thus, in light of the definition of P_m, $P_m(z) < -\frac{P_m(-z)}{9}$.

Now by the above claim $0 \leq P_m(z) < -\frac{P_m(-z)}{9}$. Combining this with $P_m(-z) \leq 0$ yields $Q_m(z) > -\frac{8P_m(-z)}{9} > 0$. Thus

$$0 \leq R_{m,r}(z) < \left(\frac{2\left(\frac{-1}{9}\right)P_m(-z)}{-\frac{8}{9}P_m(-z)}\right)^{2r} = \left(\frac{1}{4}\right)^{2r} < 2^{-r}.$$

Since $R_{m,r}(z) \geq 0$ and since for ever $\delta \geq 0$, $(1+\delta) > 0$ and $(1+\delta)(1\delta) = 1-\delta^2 \leq 1$, we have

$$1 \geq S_{m,r}(z) = \frac{1}{1+R_{m,r}(z)} > 1 - R_{m,r}(z) > 1 - 2^{-r}.$$

Hence (3a) holds.

Next consider the case when $-2^m \leq z \leq -1$. For this range of values of z, in light of Claim 4.9 we have $0 \leq P_m(-z) < -\frac{P_m(z)}{9}$. This implies $0 < Q_{m,r}(z) < -P_m(z)$. Thus

$$R_{m,r}(z) \geq \left(\frac{2P_m(z)}{-P_m(z)}\right)^{2r} = 4^r > 2^r.$$

Hence (3b) holds. □

LEMMA 4.10. *For each $L \in$ PL and each polynomial r, there exists* GapL *functions $g : \Sigma^* \to \mathbb{N}$ and $h : \Sigma^* \to \mathbb{N}$ such that, for all $x \in \Sigma^*$,*

(1) *if $\chi_L(x) = b$, then $1 - 2^{-r(|x|)} \leq \frac{g(\langle x,b\rangle)}{h(x)} \leq 1$, and*
(2) *if $\chi_L(x) \neq b$, then $0 \leq \frac{g(\langle x,b\rangle)}{h(x)} \leq 2^{-r(|x|)}$.*

PROOF. Let $L \in$ PL and f be a GapL function witnessing, in the sense of Proposition 4.1, the membership of L in PL. Then, for every $x \in \Sigma^*$, the absolute value of $f(x)$ is at least one. Let m be a polynomial such that, for every $x \in \Sigma^*$, the absolute value of $f(x)$ is at most $2^{m(|x|)}$. Let r be an arbitrary polynomial. Define:

$$\begin{aligned} h(x) &= B_{m(|x|),r(|x|)}(f(x)), \\ g(\langle x,1\rangle) &= A_{m(|x|),r(|x|)}(f(x)), \textit{and} \\ g(\langle x,0\rangle) &= B_{m(|x|,r(|x|)}(f(x)) - A_{m(|x|),r(|x|)}(f(x)). \end{aligned}$$

Then, for every $x \in \Sigma^*$, $g(\langle x,0\rangle) + g(\langle x,1\rangle) = h(x)$. For every $x \in \Sigma^*$, by Part 1 of Lemma 4.8, we have $S_{m(|x|),r(|x|)}(f(x)) = \frac{g(\langle x,1\rangle)}{h(x)}$ and since $g(\langle x,0\rangle) + g(\langle x,1\rangle) = h(x)$, $1 - S_{m(|x|),r(|x|)}(f(x)) = \frac{g(\langle x,0\rangle)}{h(x)}$. So, by Lemma 4.8, it follows that $1 - 2^{-r(|x|)} \leq \frac{g(\langle x,1\rangle)}{h(x)} \leq 1$ if $f(x) > 0$, and $2^{-r(|x|)} \geq \frac{g(\langle x,1\rangle)}{h(x)} \geq 0$ if $f(x) < 0$. Since $\frac{g(\langle x,0\rangle)}{h(x)} = 1 - \frac{g(\langle x,1\rangle)}{h(x)}$, we have $1 - 2^{-r(|x|)} \leq \frac{g(\langle x,0\rangle)}{h(x)} \leq 1$ if $f(x) < 0$, and $2^{-r(|x|)} \geq \frac{g(\langle x,0\rangle)}{h(x)} \geq 0$ if $f(x) > 0$. Now it remains to prove that both g and h are in GapL, which is easy because of the closure properties of GapL under addition and multiplication of up to polynomial length. We leave that verification as an exercise for the reader. □

THEOREM 4.11. $\mathrm{PL}^{\mathrm{PL}} = \mathrm{PL}$.

PROOF. Let $L \in \mathrm{PL}^{\mathrm{PL}}$ such that N is a non-deterministic $O(\log n)$-space bounded oracle Turing machine which obeys the Ruzzo-Simon-Tompa oracle access mechanism as described in Section 1.1.6 in Chapter 1 and which makes oracle queries to a language $A \in \mathrm{PL}$ to correctly decide if an input string $x \in \Sigma^*$ is in L or not, where Σ is the input alphabet. We use $\Gamma_N^A(x)$ to denote the answer sequence that the oracle A provides to N on input x. Let p be a polynomial bounding the running time of N. Let q be a polynomial such that, for every $x \in \Sigma^*$,

- $acc_{N^A}(x) \leq 2^{q(|x|)}$, and
- $x \in L \Leftrightarrow acc_{N^A}(x) \geq 2^{q(|x|)-1}$.

Let m be a polynomially bounded function, as defined above, that maps each integer n to the number of queries that N makes on each input of length n. Then $m(n) \leq p(n)$ for all n. For each $x \in \Sigma^*$ and w, $|w| = m(|x|)$, let $\alpha(x, w)$ denote the number of accepting computation paths that M on input x would have if for every i, $1 \leq i \leq m(|x|)$, the oracle answer to the i^{th} query of N on input x is taken to be yes if the i^{th} bit of w is a 1, and it is no if the i^{th} bit of w is a 0. Then, for every $x \in \Sigma^*$, $\alpha(x, \Gamma_N^A(x)) = acc_{N^A}(x)$, and for every $w \in \Sigma^{m(|x|)}$, $\alpha(x, w) \leq 2^{q(|x|)}$. We now define a GapL function d such that on any given input string $x \in \Sigma^*$, we have $x \in L$ if and only if $d(x) \geq 0$.

By Lemma 4.10, there exists non-negative function $g \in \mathrm{GapL}$ and a strictly positive function $h \in \mathrm{GapL}$ such that, for all $x \in \Sigma^*$ and $b \in \{0, 1\}$,

$$\begin{aligned} 1 - 2^{-r(|x|)} \leq \frac{g(\langle x, b\rangle)}{h(x)} &\leq 1 \text{ if } \chi_A(x) = b, \text{ and} \\ 0 \leq \frac{g(\langle x, b\rangle)}{h(x)} &\leq 2^{-r(|x|)}, \text{ otherwise.} \end{aligned}$$

Define, for each $x \in \Sigma^*$,

$$s(x) = \sum_{|w|=m(|x|)} \alpha(x, w) \prod_{1 \leq i \leq m(|x|)} g(\langle y_{x,i}, w_i\rangle)$$

and

$$t(x) = \prod_{1 \leq i \leq m(|x|)} h(y_{x,i}).$$

We claim that for every $x \in \Sigma^*$,

$$x \in L \Leftrightarrow \frac{s(x)}{t(x)} \geq 2^{q(|x|)-1} - \frac{1}{4}.$$

To prove the claim let $x \in \Sigma^*$. First suppose that $x \in L$. For $w = \Gamma_N^A$, the fraction

$$\kappa(x, w) = \frac{\prod_{1 \leq i \leq m(|x|)} g(\langle y_{x,i}, w_i\rangle)}{t(x)}$$

is at least

$$1 - m(|x|)2^{-min\{r(|y_{x,1}|),\ldots,r(|y_{x,m(|x|)}|)\}}.$$

Because m is bounded by p, because r is a natural polynomial, and because the machine N is a length-increasing query generator, the above amount is at least

$$1 - p(|x|)2^{-p(|x|)-q(|x|)-1} > 1 - 2^{-q(|x|)-1}.$$

Note that in the summation expression of $s(x)$, there exists a computation path which corresponds to some w such that the bits of w are consistent with the sequence of oracle replies $\Gamma_N^A(x)$. For other every other w, we always add non-negative numbers to $s(x)$ irrespective of whether $w = \Gamma_N^A(x)$ or not. Moreover, w is a non-deterministically chosen string of polynomial length, and so we get that $s(x) \in \mathrm{GapL}$. Thus the fraction $\frac{s(x)}{t(x)}$ is at least

$$2^{q(|x|)-1}(1 - 2^{-q(|x|)-1}) = 2^{q(|x|)-1} - \frac{1}{4}.$$

Next suppose that $x \notin L$. For $w = \Gamma_N^A$, $\kappa(x, w) \leq 1$ and $\alpha(x, w) = acc_{N^A}(x) \leq 2^{q(|x|)} - 1$. For other w of length $m(|x|)$, $\alpha(x, w) \leq 2^{q(|x|)}$ and

$$\kappa(x, w) \leq 2^{-min\{r(y_{x,1}),\ldots,r(|y_{x,m(|x|)}|)\}}.$$

Because m is bounded by p, because r is a natural polynomial, and because the machine N is a length-increasing query generator, the above number is at most $2^{-p(|x|)-q(|x|)-1}$. Since the number of w, $|w| = m(|x|)$, such that $w \neq \Gamma_N^A(x)$ is $2^{m(|x|)} - 1 < 2^{p(|x|)}$, $\frac{s(x)}{t(x)}$ is less than

$$2^{q(|x|)-1} - 1 + 2^{p(|x|)}2^{-p(|x|)-q(|x|)-1} = 2^{q(|x|)-1} - 1 + 2^{-q(|x|)-1} \leq 2^{q(|x|)-1} - \frac{1}{2}.$$

Thus the claim holds.

Now define

$$d(x) = 4s(x) - (2^{q(|x|+1} - 1)t(x).$$

Then, for every $x \in \Sigma^*$, $x \in L$ if and only if $d(x) \geq 0$. Clearly $d \in \mathrm{GapL}$ which proves our theorem. □

4.2.1. An alternate proof of NL ⊆ PL.

THEOREM 4.12. $\mathrm{C_{=}L} \subseteq \mathrm{PL}$.

PROOF. Let $L \subseteq \Sigma^*$ and let $L \in \mathrm{C_{=}L}$. Let $f \in \mathrm{GapL}$ be such that for any input $x \in \Sigma^*$ we have $x \in L$ if and only if $f(x) = 0$. Let $L_1 = \{x \in \Sigma^* | f(x) > 0\}$ and let $L_2 = \{x \in \Sigma^* | f(x) < 0\}$. It is easy to note that $x \in L$ if and only if both of the following two conditions are false:

1. is $f(x) > 0$, equivalently $x \in L_1$,
2. is $f(x) < 0$, equivalently $x \in L_2$.

Clearly $L_1 \in \mathrm{PL}$. Determining if $f(x) > 0$ is therefore possible using a logarithmic space bounded deterministic Turing machine that has access to L_1 as an oracle. Next to determine if $f(x) < 0$ let us consider the function $h(x) = -(2f(x) + 1)$. Using Theorem 4.11 it follows that $h \in \mathrm{GapL}$. We now observe that if $x \in L_2$ then $f(x) < 0$ and so $h(x) > 0$. Conversely, if $x \notin L_2$ then $f(x) \geq 0$ and so $h(x) < 0$. As a result we get $L_2 \in$ co-PL. Using Corollary 4.2 it follows that $L_2 \in \mathrm{PL}$. Consider $L' = L_1 \cup L_2$. Then it follows from Theorem 4.4

that $L' \in$ PL. Now a logarithmic space bounded deterministic Turing machine that has access to L' as an oracle can therefore determine if the input $x \in \Sigma^*$ is such that $f(x) = 0$. This shows that $\mathrm{C}_{=}\mathrm{L} \subseteq \mathrm{L}^{\mathrm{PL}}$. However $\mathrm{L}^{\mathrm{PL}} \subseteq \mathrm{PL}^{\mathrm{PL}} = \mathrm{PL}$ by Theorem 4.11 from which we get $\mathrm{C}_{=}\mathrm{L} \subseteq$ PL. □

Notes

The complexity class PL was defined by John Gill in [**Gil77**]. The definition of PL which we have stated in Definition 2.31 is drastically different from the definition given by John Gill in [**Gil77**]. We state a definition of PL from [**AO96**] as follows: PL is defined to be a complexity class of languages A for which there exists a probabilistic Turing machine (in the sense of [**Gil77**]; that is, a Turing machine with access to a source of unbiased random bits), such that on input x the machine never uses more than $\log |x|$ space, and $x \in A$ if and only if the probability that the machine reaches an accepting configuration is greater than one half. As mentioned in [**AO96**], the source of the difficulty in analyzing PL is because probabilistic logspace machines can perform useful work after exponentially many computation steps. However H. Jung in [**Jun85**] has shown that at least in the unbounded error model (which defined the class PL), the polynomial time restriction causes no loss of power. In other words, H. Jung has shown that the probabilistic Turing machine associated with any language $A \in$ PL can be assumed to be running in time which is polynomial in the size of the input. Eric Allender and Mitsunori Ogihara in [**AO96**] have given a simpler proof of this Jung's Theorem in [**AO96**, Theorem 6]. Also our definition of PL stated in Definition 2.31 is shown to be equivalent to the conventional definition of PL using Jung's Theorem by Eric Allender and Mitsunori Ogihara in [**AO96**, Proposition 3]. As a result we use their proposition to define the complexity class PL. [**AO96**] also show some more interesting closure properties of PL such as the closure of PL under logspace conjunctive truth-table reductions and logspace disjunctive truth-table reductions which use low degree polynomials defined by [**BRS95**].

Results shown in Section 4.1 that PL is closed under complement, union and intersection are derived from the results shown in the polynomial time setting for the complexity class PP in [**For97**, Section 4.4] by Lance Fortnow. The framework of low-degree polynomials to approximate the sign of a GapL function used by Mitsunori Ogihara in [**Ogi98**] which leads us to prove in Theorem 4.11 that the complexity class PL is closed under PL-Turing reductions is obtained from the polynomial time counting classes setting and it is due to Richard Beigel, Nick Reingold and Daniel Spielman in [**BRS95**].

CHAPTER 5

Complete problems and Hierarchies

In this chapter we list the set of problems that are logspace many-one complete for various logarithmic space bounded counting classes we have studied in Chapters 1 to 4.

5.1. Problems logspace many-one complete for NL

2SAT (Satisfiability of 2-CNF Boolean formulae)
INSTANCE: Boolean formula ϕ in the 2-CNF.
QUESTION: Is there an assignment of Boolean values for the variables of ϕ that is a satisfying assignment for ϕ?
REFERENCE: Theorem 2.25.

DSTCON (Directed st-connectivity)
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s, t \in V$.
QUESTION: Is there a directed path from s to t in G?
REFERENCE: Theorem 2.7.

SLDAGSTCON (Simple Layered Directed Acyclic Graph st-connectivity)
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s, t \in V$ where the vertices in G are arranged as a square matrix such that there are n rows and every row has n vertices. Any edge in this graph is from a vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2$. Also s is a vertex in the first row and t is a vertex in the last row of G.
QUESTION: Is there a directed path from s to t in G?
REFERENCE: Theorem 2.10.

5.2. Problems logspace many-one complete for $\mathbf{C_{=}L}$

ExactDSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct.
QUESTION: Is the number of directed paths from s_1 to t_1 equal to the number of directed paths from s_2 to t_2 in G?
REFERENCE: Follows from Definition 2.32 and the definition of GapDSTCON

given below.

ExactSLDAGSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where the vertices in G are arranged as a square matrix such that there are n rows and every row has n vertices. Any edge in this graph is from a vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2$. Also s_1, s_2 are vertices in the first row and t_1, t_2 are vertices in the last row of G such that at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct.
QUESTION: Is the number of directed paths from s_1 to t_1 equal to the number of directed paths from s_2 to t_2 in G?
REFERENCE: Follows from Definition 2.32 and the definition of GapSLDAGSTCON given below.

5.3. Problems logspace many-one complete for ♯L

♯DSTCON (sharp Directed st-connectivity)
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s, t \in V$.
QUESTION: Count the number of directed paths from s to t in G?
REFERENCE: Proposition 2.15.

♯SLDAGSTCON (sharp Simple Layered Directed Acyclic Graph st-connectivity)
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s, t \in V$, where the vertices in G are arranged as a square matrix such that there are n rows and every row has n vertices. Any edge in this graph is from a vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2$. Also s is a vertex in the first row and t is a vertex in the last row of G.
QUESTION: Count the number of directed paths from s to t in G?
REFERENCE: Proposition 2.16.

5.4. Problems logspace many-one complete for GapL

GapDSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct.
QUESTION: Find the number of directed paths from s_1 to t_1 minus the number of directed paths from s_2 to t_2 in G?
REFERENCE: Follows from ♯DSTCON defined above and Lemma 2.49.

GapSLDAGSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where the vertices in G are arranged as a square matrix

such that there are n rows and every row has n vertices. Any edge in this graph is from a vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2$. Also s_1, s_2 are vertices in the first row and t_1, t_2 are vertices in the last row of G such that at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct.
QUESTION: Find the number of directed paths from s_1 to t_1 minus the number of directed paths from s_2 to t_2 in G?
REFERENCE: Follows from $\sharp$SLDAGSTCON defined above and Lemma 2.49.

5.5. Problems logspace many-one complete for $\mathbf{Mod}_k\mathbf{L}$

$\mathbf{Mod}_k st$GapDSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct.
QUESTION: Is the (number of directed paths from s_1 to t_1 minus the number of directed paths from s_2 to t_2 in G) not divisible by k, where $k \in \mathbb{N}$ and $k \geq 2$?
REFERENCE: Follows from Definition 3.1, $\sharp$DSTCON, Proposition 2.33 and using elementary modulo arithmetic as in Lemma 3.22.

$\mathbf{Mod}_k st$GapSLDAGSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where the vertices in G are arranged as a square matrix such that there are n rows and every row has n vertices. Any edge in this graph is from a vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2$. Also s_1, s_2 are vertices in the first row and t_1, t_2 are vertices in the last row of G such that at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct
QUESTION: Is the (number of directed paths from s_1 to t_1 minus the number of directed paths from s_2 to t_2 in G) not divisible by k, where $k \in \mathbb{N}$ and $k \geq 2$?
REFERENCE: Follows from Definition 3.1, $\sharp$SLDAGSTCON, Proposition 2.33 and using elementary modulo arithmetic as in Lemma 3.22.

$\mathbf{Mod}_k st$Path
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s, t \in V$.
QUESTION: Is the number of directed paths from s to t in G not divisible by k, where $k \in \mathbb{N}$ and $k \geq 2$?
REFERENCE: Follows from Definition 3.1, $\sharp$DSTCON and the proof of Theorem 3.38.

$\mathbf{Mod}_k st$SLDAGSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s, t \in V$, where the vertices in G are arranged as a square matrix such that there are n rows and every row has n vertices. Any edge in this graph is from a

vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2$. Also s is a vertex in the first row and t is a vertex in the last row of G.
QUESTION: Is the number of directed paths from s to t in G not divisible by k, where $k \in \mathbb{N}$ and $k \geq 2$?
REFERENCE: Follows from Definition 3.1, ♯SLDAGSTCON. Similar to the logspace many-one completeness of Mod$_k st$Path for $\mathrm{Mod}_k\mathrm{L}$.

5.6. Problems logspace many-one complete for ModL

ModDSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s, t \in V$.
QUESTION: Is the number of directed paths from s to t in G not divisible by k, where $k = |g(x)|$ for some $g \in$ FL and $x = (G, s, t)$?
REFERENCE: Follows from Definition 3.16, ♯DSTCON and Lemma 3.22.

ModGapDSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct.
QUESTION: Is the (number of directed paths from s_1 to t_1 minus the number of directed paths from s_2 to t_2 in G) not divisible by k, where $k = |g(x)|$ for some $g \in$ FL and $x = (G, s, t)$?
REFERENCE: Follows from Definition 3.16 and ♯DSTCON.

ModGapSLDAGSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where the vertices in G are arranged as a square matrix such that there are n rows and every row has n vertices. Any edge in this graph is from a vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2$. Also s_1, s_2 are vertices in the first row and t_1, t_2 are vertices in the last row of G such that at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct.
QUESTION: Is the (number of directed paths from s_1 to t_1 minus the number of directed paths from s_2 to t_2 in G) not divisible by k, where $k = |g(x)|$ for some $g \in$ FL and $x = (G, s, t)$?
REFERENCE: Follows from Definition 3.16 and ♯DSTCON.

ModSLDAGSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s, t \in V$, where the vertices in G are arranged as a square matrix such that there are n rows and every row has n vertices. Any edge in this graph is from a vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2$. Also s is a vertex in the first row and t is a vertex in the last row of G.
QUESTION: Is the number of directed paths from s to t in G not divisible by k,

where $k = |g(x)|$ for some $g \in$ FL and $x = (G, s, t)$?
REFERENCE: Follows from Definition 3.16, ♯SLDAGSTCON and Lemma 3.22.

5.7. Problems logspace many-one complete for PL

ProbDSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct.
QUESTION: Is the number of directed paths from s_1 to t_1 minus the number of directed paths from s_2 to t_2 in G greater than 0?
REFERENCE: Follows from Definition 2.31.

ProbSLDAGSTCON
INSTANCE: Directed graph $G = (V, E)$ which is given in terms of its adjacency matrix, $s_1, t_1, s_2, t_2 \in V$, where the vertices in G are arranged as a square matrix such that there are n rows and every row has n vertices. Any edge in this graph is from a vertex in the i^{th} row to a vertex in the $(i+1)^{st}$ row, where $1 \leq i \leq (n-1)$ and $n \geq 2$. Also s_1, s_2 are vertices in the first row and t_1, t_2 are vertices in the last row of G such that at least three vertices in $\{s_1, s_2, t_1, t_2\}$ should be necessarily distinct.
QUESTION: Is the number of directed paths from s_1 to t_1 minus the number of directed paths from s_2 to t_2 in G greater than 0?
REFERENCE: Follows from Definition 2.31.

5.8. Closure properties of logarithmic space bounded counting classes

We summarize the most important closure properties of language based logarithmic space bounded counting classes shown in Chapters 2 to 4 as Table 5.1.

5.9. Logarithmic space bounded counting class hierarchies

In exploring the power and limitations of logarithmic space bounded counting classes, it is a practice to define logarithmic space bounded counting class hierarchies for every logarithmic space bounded counting class.

DEFINITION 5.1. Let Σ be the input alphabet and let $\text{NLH}_1 = \text{NL}$. For $i \geq 2$, we define NLH_i as the class of all languages $L \subseteq \Sigma^*$ such that there exists a $O(\log n)$ space bounded non-deterministic oracle Turing machine M^A that has access to a language $A \in \text{NLH}_{i-1}$ as an oracle and we have $x \in L$ if and only if $acc_{M^A}(x) > 0$, where n denotes the size of the input. Here we assume that M^A submits queries to the oracle A according to the Ruzzo-Simon-Tompa oracle access mechanism. The non-deterministic logspace hierarchy, denoted by NLH, is defined as

$$\text{NLH} = \cup_{i \geq 1} \text{NLH}_i$$

Complexity class of languages $\mathcal{C}$	Closure property				
	union $\cup$	complement $\overline{L}$	intersection $\cap$	logspace Turing reduction	$\mathcal{C}$-Turing reduction
NL	✓	✓	✓	✓	✓
C$_=$L	✓	unknown	unknown	unknown	unknown
Mod$_p$L, $p \in \mathbb{N}$ is a prime.	✓	✓	✓	✓	✓
Mod$_k$L, $k \in \mathbb{N}$ is a composite.	✓	unknown	unknown	unknown	unknown
ModL	unknown	✓ assuming NL = UL	unknown	unknown	unknown
PL	✓	✓	✓	✓	✓

TABLE 5.1. Fundamental closure properties of logarithmic space bounded counting classes. Here, ✓ denotes that we have shown that property for the corresponding complexity class in Chapters 1 to 4.

DEFINITION 5.2. Let Σ be the input alphabet and let $\text{Mod}_p\text{LH}_1 = \text{Mod}_p\text{L}$, where $p \in \mathbb{N}$ and p is a prime. For $i \geq 2$, we define Mod_pLH_i as the class of all languages $L \subseteq \Sigma^*$ such that there exists a $O(\log n)$ space bounded non-deterministic oracle Turing machine M^A that has access to a language $A \in \text{Mod}_p\text{L}$ as an oracle with $A \in \text{Mod}_p\text{LH}_{i-1}$ and we have $x \in L$ if and only if $acc_{M^A}(x) \not\equiv 0 (\text{mod } p)$, where $p \in \mathbb{N}$, p is a prime and n denotes the size of the input. Here we assume that M^A submits queries to the oracle A according to the Ruzzo-Simon-Tompa oracle access mechanism. The modulo-prime logspace hierarchy, denoted by Mod_pLH is defined as,

$$\text{Mod}_p\text{LH} = \cup_{i \geq 1} \text{Mod}_p\text{LH}_i,$$

where $p \in \mathbb{N}$ and p is a prime.

DEFINITION 5.3. Let Σ be the input alphabet and let $\text{PLH}_1 = \text{PL}$. For $i \geq 2$, we define PLH_i as the class of all languages $L \subseteq \Sigma^*$ such that there exists a $O(\log n)$ space bounded non-deterministic oracle Turing machine M^A that has access to a language $A \in \text{PL}$ as an oracle with $A \in \text{PLH}_{i-1}$ and we have $x \in L$ if and only if $acc_{M^A}(x) > 0$, where n denotes the size of the input. Here we assume that M^A submits queries to the oracle A according to the Ruzzo-Simon-Tompa oracle access mechanism. The probabilistic logspace hierarchy, denoted by PLH is defined as,

$$\text{PLH} = \cup_{i \geq 1} \text{PLH}_i$$

DEFINITION 5.4. Let Σ be the input alphabet and let $\text{Mod}_k\text{LH}_1 = \text{Mod}_k\text{L}$, where $k \in \mathbb{N}$ and $k \geq 2$. For $i \geq 2$, we define Mod_kLH_i as the class of all languages $L \subseteq \Sigma^*$ such that there exists a $O(\log n)$ space bounded non-deterministic oracle Turing machine M^A that has access to a language $A \in \text{Mod}_k\text{LH}_{i-1}$ and we have $acc_{M^A}(x) \not\equiv 0 (\text{mod } k)$, where $k \in \mathbb{N}$, $k \geq 2$ and n denotes the size of the input. Here we assume that M^A submits queries to the oracle A according to the Ruzzo-Simon-Tompa oracle access mechanism. The modulo-k logspace hierarchy, denoted by Mod_kLH is defined as,

$$\text{Mod}_k\text{LH} = \cup_{i \geq 1} \text{Mod}_k\text{LH}_i,$$

where $k \in \mathbb{N}$ and $k \geq 2$.

THEOREM 5.5. *Let $\mathcal{C}$ be any of the following logarithmic space bounded counting classes:* NL *or* Mod_pL*, where $p \in \mathbb{N}$ and p is a prime, or* PL*.*

$$\mathcal{C}\text{H} = \cup_{i \geq 1} \mathcal{C}\text{H}_i = \mathcal{C}.$$

PROOF. Let $\mathcal{C}$ be NL or $\text{C}_=$L or Mod_pL, where $p \in \mathbb{N}$ and p is a prime, or PL. We know from the results shown in Chapters 1 to 3 which is summarized in Table 5.1 that each of these complexity classes denoted by $\mathcal{C}$ is closed under $\mathcal{C}$-Turing reductions. It is trivial from the definition of $\mathcal{C}$H that $\mathcal{C}\text{H}_1$=$\mathcal{C}$. We now use induction on $i \geq 1$ and assume that $\mathcal{C}\text{H}_i = \mathcal{C}$ and consider $\mathcal{C}\text{H}_{i+1}$. The remaining part of this proof easily follows from the definition of $\mathcal{C}$H using the result that $\mathcal{C}$ is closed under $\mathcal{C}$-Turing reductions and from our inductive assumption that $\mathcal{C}\text{H}_i$=$\mathcal{C}$. □

Since it is not known whether Mod_kL is closed under the Mod_kL-Turing reductions, where $k \in \mathbb{N}$ and $k \geq 2$ is a composite number, we cannot show that $\text{Mod}_k\text{LH} = \text{Mod}_k\text{L}$ in Theorem 5.5.

5.10. Hierarchies and Boolean circuits with oracle gates

We recall the definition of standard unbounded fan-in basis from Definition 1.21, admissible encoding scheme from Definition 1.23, and family of L-uniform circuits from Definition 1.24.

DEFINITION 5.6. We define $\mathcal{B}_1(\mathcal{C}) = \{\neg, (\wedge^n)_{n \in \mathbb{N}}, (\vee^n)_{n \in \mathbb{N}}, \mathcal{C}\}$ as the standard unbounded fan-in basis with oracle gates for $\mathcal{C}$, a complexity class of languages.

DEFINITION 5.7. Let $\mathcal{B}_1(\mathcal{C})$ be a standard unbounded fan-in basis with oracle gates for the complexity class of languages $\mathcal{C}$, and let $s, d : \mathbb{N} \to \mathbb{N}$. We define the complexity class, SIZE-DEPTH$_{\mathcal{B}_1(\mathcal{C})}(s, d)$ as the class of all sets $A \subseteq \{0,1\}^*$ for which there is a circuit family $(C_n)_{n \in \mathbb{N}}$ over the basis $\mathcal{B}_1(\mathcal{C})$ of size $O(s)$ and depth $O(d)$ that accepts A.

We recall the definition of U_L-AC^0 from Definition 1.25.

DEFINITION 5.8. Let $\mathcal{C}$ be any of the following logarithmic space bounded counting classes: NL or Mod_pL, where $p \in \mathbb{N}$ and p is a prime, or PL. We define L-uniform AC$^0(\mathcal{C}, i)$, denoted by U_L-AC$^0(\mathcal{C}, i)$, as the class of all languages

computable by circuits, L-uniform SIZE-DEPTH$_{\mathcal{B}_1(\mathcal{C})}(q(n), O(1))$ over the basis $\mathcal{B}_1(C)$ such that there are no more than i oracle gates for $\mathcal{C}$ in any directed path from an input gate to an output gate in any circuit, where $q(n)$ is a polynomial in n and n is the size of the input. We define $\mathrm{U_L}$-AC$^0(\mathcal{C}) = \cup_{i\geq 0}\mathrm{U_L}$-AC$^0(\mathcal{C}, i)$.

It is easy to note that $\mathrm{U_L}$-AC$^0 \subseteq \mathrm{U_L}$-AC$^0(\mathcal{C}, 0) \subseteq \mathrm{U_L}$-AC$^0(\mathcal{C})$ and $\mathcal{C} \subseteq \mathrm{U_L}$-AC$^0(\mathcal{C}, 1) \subseteq \mathrm{U_L}$-AC$^0(\mathcal{C})$ follow from Definitions 1.25 and 5.8.

THEOREM 5.9. *Let $\mathcal{C}$ be any of the following logarithmic space bounded counting classes:* NL *or* Mod$_p$L*, where $p \in \mathbb{N}$ and p is a prime, or* PL. $\mathrm{U_L}$-AC$^0(\mathcal{C}) = \mathcal{C}$H. *In particular, for $i \geq 1$ we have* $\mathrm{U_L}$-AC$^0(\mathcal{C}, i) = \mathcal{C}\mathrm{H}_i$.

PROOF. We know from Theorem 5.5 that $\mathcal{C}\mathrm{H} = \mathcal{C}$. Also, it follows from Definition 5.8 that $\mathcal{C} \subseteq \mathrm{U_L}$-AC$^0(\mathcal{C})$ which proves that $\mathcal{C}\mathrm{H} \subseteq \mathrm{U_L}$-AC$^0(\mathcal{C})$.

Conversely, we know that $\mathrm{U_L}$-AC$^0 \subseteq \mathcal{C}$. Now, consider the class $\mathrm{U_L}$-AC$^0(\mathcal{C}, 1)$. We want to show that $\mathrm{U_L}$-AC$^0(\mathcal{C}, 1) \subseteq \mathcal{C}\mathrm{H}_1$.

Let $L \in \mathrm{U_L}$-AC$^0(\mathcal{C}, 1)$ and let $(C_n)_{n\in\mathbb{N}}$ be a L-uniform circuit family accepting L. Therefore the admissible encoding scheme $\overline{C_n}$ of C_n is computable in space $O(\log n)$ when 1^n is given as an input. Clearly it follows from the Definition 5.8 that there is at most 1 oracle gate $\mathcal{C}$ on any directed path from an input gate to an output gate in C_n, where $n \geq 1$.

It is easy to note that given an admissible encoding scheme $\overline{C_n}$ of C_n we can assume without loss of generality that any two sub-circuits of C_n with the $\mathcal{C}$ oracle gate as the root do not have any gate in common. This is easy to observe since C_n is a constant depth circuit and a $O(\log n)$ space bounded deterministic Turing machine can obtain $\overline{C_n}$ when given 1^n as input and also make sufficiently many copies of Boolean gates and assign predecessors from the successive levels using $\overline{C_n}$ such that this property is preserved for all sub-circuits with the $\mathcal{C}$ oracle gate as the root. Using this assumption it follows that a $O(\log n)$ space bounded deterministic Turing machine can also compute inputs to the oracle gate and it writes the input on the oracle tape of a $\mathcal{C}$ oracle. We assume that the deterministic $O(\log n)$ Turing machine similarly submits all the queries to the $\mathcal{C}$ oracle that are possible and it also gets the membership replies from $\mathcal{C}$ oracle. Once again based on the L-uniform circuit description, and using oracle replies it is possible for a $O(\log n)$ space bounded deterministic Turing machine to compute the output of C_n on the given input. Since the entire computation of evaluating a $\mathrm{U_L}$-AC$^0(\mathcal{C}, 1)$ circuit that decides if an input string of length n is in L is in L$^{\mathcal{C}}$ and $\mathcal{C}$ is closed under logspace Turing reductions (Table 5.1), it follows that $\mathrm{U_L}$-AC$^0(\mathcal{C}, 1) \subseteq$ C.

We now use induction on i and assume for $i \geq 1$ that $\mathrm{U_L}$-AC$^0(\mathcal{C}, i) \subseteq \mathcal{C}\mathrm{H}_i$ and consider the class $\mathrm{U_L}$-AC$^0(\mathcal{C}, i + 1)$. We want to show that $\mathrm{U_L}$-AC$^0(\mathcal{C}, i + 1) \subseteq \mathcal{C}\mathrm{H}_{i+1}$. The proof for this step is similar to the base case. It uses the inductive assumption and the property of $\mathcal{C}$ that $\mathcal{C}\mathrm{H}_i = \mathcal{C}$ which follows from the collapse of $\mathcal{C}\mathrm{H}$ to $\mathcal{C}$ shown in Theorem 5.5, which in turn is implied by the property of $\mathcal{C}$ that it is closed under $\mathcal{C}$-Turing reductions (Table 5.1). Thus we get the proof. □

COROLLARY 5.10. *Let $\mathcal{C}$ be any of the following logarithmic space bounded counting classes:* NL *or* $\mathrm{Mod}_p\mathrm{L}$*, where $p \in \mathbb{N}$ and p is a prime, or* PL. $\mathrm{U_L}$-$\mathrm{AC}^0(\mathcal{C}) = \mathcal{C}$.

Exercises

(1) Define the ModL hierarchy, denoted by ModLH.
(2) Define the $\sharp$L hierarchy, denoted by $\sharp$LH and show that ModLH $= \sharp$LH.

Notes

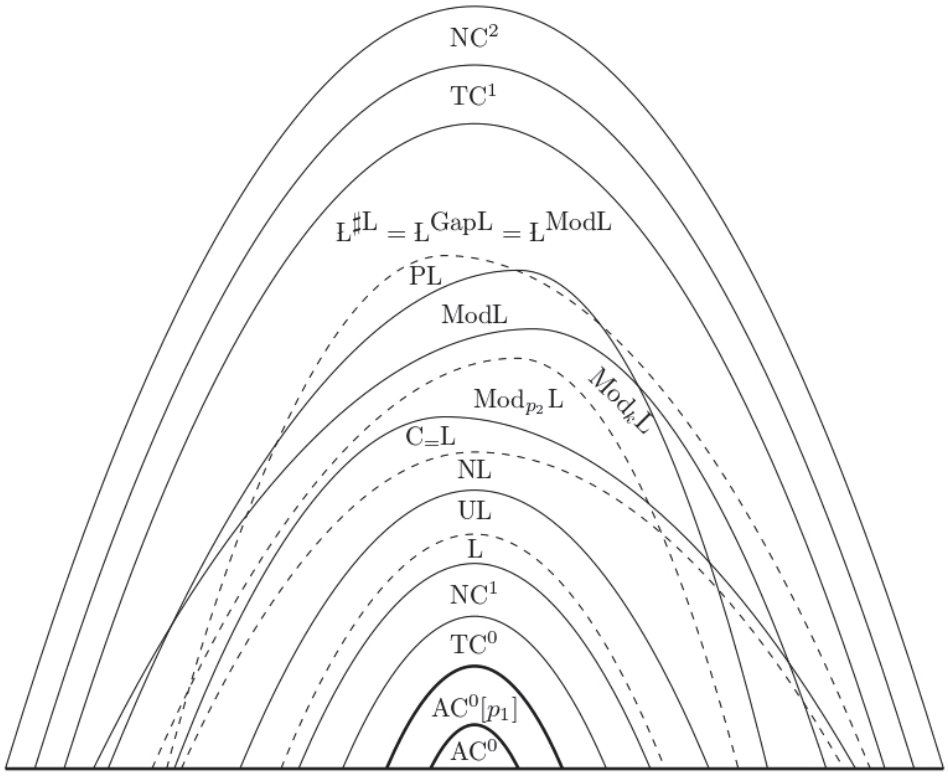

FIGURE 5.1. *The landscape of complexity classes contained in* NC^2. *In this figure, thick line between two complexity classes $\mathcal{C}_1$ and $\mathcal{C}_2$ denotes that $\mathcal{C}_1 \subset \mathcal{C}_2$. Dashed line above a complexity class $\mathcal{C}$ denotes that $\mathcal{C} =$co-$\mathcal{C}$ (in fact $\mathcal{C}$ is closed under $\mathcal{C}$-Turing reductions). Here $p_1, p_2 \geq 2$ are primes and $k \geq 6$ is a composite number that has more than one distinct prime divisor and $p_2 | k$. We assume that the circuit complexity classes are* L*-uniform circuit complexity classes.*

In exploring the power and limitations of logarithmic space bounded counting classes, it is a routine exercise to define logarithmic space bounded counting class hierarchies for every logarithmic space bounded counting class that has been defined. These hierarchies are similar to the well known Polynomial Hierarchy (PH) which is defined based on the complexity class NP. Our definitions and collapse results on hierarchies for $\mathrm{C_{=}L}$, PL and $\mathrm{Mod}_p\mathrm{L}$ in Section 5.9 is from [**AO96, Ogi98, ABO99, HRV00**]. Theorem 5.9 was shown for $\mathcal{C} = \mathrm{PL}$ and $\mathcal{C} = \mathrm{C_{=}L}$ in [**AO96**]. We note that using [**Vol99**, Theorems 2.18 and 2.32] it is possible to show that we can evaluate $\mathrm{U_L}$-NC^1 circuits in L.

Pause to ponder. Now what happens if instead, we define the $\sharp$L hierarchy or a complexity class of logspace uniform constant depth Boolean circuits that also contains oracle gates for $\sharp$L functions? This has also been well studied by Eric Allender and Mitsunori Ogihara in [**AO96**]. Let us now define $\sharp$LH which is based on functions in $\sharp$L [**AO96**]:

Let Σ be the input alphabet and let $\sharp\mathrm{LH}_1 = \sharp\mathrm{L}$. For $i \geq 2$, we define $\sharp\mathrm{LH}_i$ to be the class of all functions $f : \Sigma^* \to \mathbb{Z}^+$ such that there exists a $O(\log n)$-space bounded non-deterministic oracle Turing machine M^g that has access to a function $g : \Sigma^* \to \mathbb{Z}^+$ as an oracle with $g \in \sharp\mathrm{LH}_{i-1}$ and $f(x) = acc_{M^g}(x)$, where n denotes the size of the input. Here we assume that M^g submits queries to the oracle function g according to the Ruzzo-Simon-Tompa oracle access mechanism. The counting logspace hierarchy, denoted by $\sharp$LH is defined as,

$$\sharp\mathrm{LH} = \cup_{i \geq 1} \sharp\mathrm{LH}_i$$

An interesting question is, does the $\sharp$LH collapse? Few points with regard to this question are as follows:

(1) In [**Coo85**], Stephen A. Cook proves that the following four linear algebraic problems of computing the determinant of an integer matrix, computing the entries of the powers of an integer matrix, computing the entries of the iterated product of a set of integer matrices, and computing the entries of the inverse of an integer matrix are all L-uniform NC^1-Turing equivalent (refer to [**Coo85**] for the definition of L-uniform NC^1-Turing reduction).
(2) Eric Allender and Mitsunori Ogihara have shown that L-uniform $\mathrm{AC}^0(\sharp\mathrm{L}) = \sharp\mathrm{LH}$ in [**AO96**].
(3) Let us now assume that L-uniform $\mathrm{NC}^1(\sharp\mathrm{L}) =$ L-uniform $\mathrm{AC}^0(\sharp\mathrm{L})$.
(4) As we will see in Corollary 6.48 in Chapter 6 of this monograph, it is well known that computing the determinant of integer matrices is logspace many-one complete for GapL.
(5) So shall we expect all the four problems introduced by S. A. Cook in [**Coo85**] to be logspace many-one equivalent which will imply that these four problems are also logspace many-one complete for GapL? If so, then as a consequence of results shown by Eric Allender, it implies that L-uniform $\mathrm{NC}^1(\sharp\mathrm{L}) = \mathrm{GapL}$ which will mean that $\sharp$LH collapses to GapL.

CHAPTER 6

The complexity of computing the determinant

6.1. Permutations

Let $\Omega = \{1, \ldots, n\}$. A permutation on Ω is a one-to-one and onto mapping from Ω to itself. The set of all permutations of Ω is denoted by S_n, and it is easy to see that $|S_n| = n!$.

PROPOSITION 6.1. *S_n forms a group under the operation of composition of mappings, which we call as multiplication of permutations, and we say that S_n is the symmetric group of degree n on Ω.*

Given $\theta \in S_n$ and $s \in \Omega$, let $s\theta$ denote an element $s' \in \Omega$ such that $\theta(s) = s'$. Similarly, let $s\theta^i$ denote the element $s'' \in \Omega$ such that $\theta^i(s) = s''$, where $i \geq 1$.

PROPOSITION 6.2. *If $\theta \in S_n$ and $s \in \Omega$, then there is the smallest positive integer t depending on θ and s such that $s\theta^t = s$.*

DEFINITION 6.3. Given θ and $s \in \Omega$ such that $s\theta^t = s$, let $x_i = s\theta^i$, where $1 \leq i \leq t$. The set $\{x_1, \ldots, x_t\}$ is called the orbit of s under θ.

PROPOSITION 6.4. *If $\theta \in S_n$, $s, s' \in \Omega$ and A is the orbit of s under θ and A' is the orbit of s' under θ, then either $A = A'$ or $A \cap A' = \emptyset$.*

PROOF. $s' \in A$ if and only if $s \in A'$. Therefore, we have either $A = A'$ or $A \cap A' = \emptyset$. □

DEFINITION 6.5. Given θ and $s \in \Omega$ such that $s\theta^t = s$, let us consider the orbit $A = \{x_1, x_2, \ldots, x_t\}$ of s under θ, where $x_i = s\theta^i$. Let $\{y_1, y_2, \ldots, y_t\}$ be elements of A arranged in the increasing order. We define the ***orbicycle*** containing s under the permutation θ to be $(y_1\theta\ y_2\theta\ \cdot\ \cdot\ \cdot\ y_t\theta)$.

LEMMA 6.6. *Every permutation $\theta \in S_n$ can be uniquely expressed as a product of its orbicycles.*

PROOF. Since orbicycles are formed from orbits, it is clear from Proposition 6.4 that given two orbicycles, either they are equal or there does not exist any $x \in \Omega$ which is in two different orbicycles.

If we consider an orbit $A = \{x_1, x_2, \ldots, x_t\}$ of s under θ, where $x_i = s\theta^i$, $1 \leq i \leq t$, and its orbicycle $(y_1\theta\ y_2\theta\ \cdot\ \cdot\ \cdot\ y_t\theta)$ as a permutation in S_n then it permutes elements of A and point-wise fixes every element of Ω which is not in A. As a result, considering orbicycles of θ as individual permutations, if we multiply all orbicycles of θ we get θ. By re-ordering orbicycles of θ based on the least

element in each orbicycle we get that every permutation $\theta \in S_n$ can be uniquely expressed as a product of its orbicycles. □

EXAMPLE 6.7. (Even permutation) Let $\Omega = \{1, \ldots, 9\}$. Consider the permutation,

$$\theta = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 5 & 4 & 8 & 7 & 6 & 1 & 3 & 2 & 9 \end{pmatrix}.$$

Orbit of 1 under $\theta = \{5, 6, 1\}$, orbit of 2 under $\theta = \{4, 7, 3, 8, 2\}$ and orbit of 9 under $\theta = \{9\}$. Orbicycle containing 1 under $\theta = (5\ 6\ 1)$, orbicycle containing 2 under $\theta = (4\ 8\ 7\ 3\ 2)$ and orbicycle containing 9 under $\theta = (9)$. It is easy to see that $\theta = (5\ 6\ 1)(4\ 8\ 7\ 3\ 2)(9)$.

EXAMPLE 6.8. (Odd permutation) Let $\Omega = \{1, \ldots, 9\}$. Consider the permutation,

$$\theta' = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 8 & 7 & 4 & 6 & 2 & 5 & 3 & 9 \end{pmatrix}.$$

Orbit of 1 under $\theta = \{1\}$, orbit of 2 under $\theta = \{8, 3, 7, 5, 6, 2\}$, orbit of 4 under $\theta = \{4\}$ and orbit of 9 under $\theta = \{9\}$. Orbicycle containing 1 under $\theta = (1)$, orbicycle containing 2 under $\theta = (8\ 7\ 6\ 2\ 5\ 3)$, orbicycle containing 4 under $\theta = (4)$ and orbicycle containing 9 under $\theta = (9)$. It is easy to see that $\theta' = (1)(8\ 7\ 6\ 2\ 5\ 3)(4)(9)$.

DEFINITION 6.9. We say that an orbicycle is a ***2-orbicycle*** if $s\theta \neq s$, $s\theta^2 = s$ for some $s \in \Omega$, and $s'\theta = s'$, for all $s' \in \Omega$ such that $s \neq s'$ and $s\theta \neq s'$.

THEOREM 6.10. *Every orbicycle can be uniquely expressed as a product of* 2*-orbicycles. Every permutation can be uniquely expressed as a product of* 2*-orbicycles.*

PROOF. Consider an orbit $A = \{x_1, x_2, \ldots, x_t\}$ of s under θ, where $x_i = s\theta^i$, $1 \leq i \leq t$, and its orbicycle $(y_1\theta\ y_2\theta\ \cdots\ y_t\theta)$. Then, θ is the product of 2-orbicycles $(y_1\theta\ y_1)\ (y_1\theta^2\ y_1)\ \cdots\ (y_1\theta^{t-1}\ y_1)$ in this order.

Since orbits of any two elements in Ω is either the same or both are disjoint, using Lemma 6.6, it is easy to see that every permutation can be uniquely expressed as a product of 2-orbicycles, maintaining their order. □

COROLLARY 6.11. *Every orbicycle can be uniquely expressed as a product of* $(l-1)$ 2*-orbicycles, where* l *is the number of elements of* Ω *in the orbit of the orbicycle. Every permutation* θ *can be uniquely expressed as a product of* $\sum_{i=1}^{m}(l_i - 1)$ 2*-orbicycles, where* l_i *denotes the number of elements in the orbit of the* i^{th} *orbicycle, and* m *denotes the number of orbicycles of* θ*.*

DEFINITION 6.12. A permutation is said to be an even permutation if it is a product of an even number of 2-orbicycles.

DEFINITION 6.13. A permutation is an odd permutation if it is not an even permutation.

LEMMA 6.14. *A permutation can be expressed as the product of an even number of* 2*-orbicycles if and only if it cannot be expressed as the product of an odd number of* 2*-orbicycles.*

PROOF. Consider the polynomial in n variables

$$p(x_1, \ldots, x_n) = \prod_{i<j}(x_i - x_j).$$

If $\theta \in S_n$, let θ act on the polynomial $p(x_1, \ldots, x_n)$ by

$$\theta : p(x_1, \ldots, x_n) = \prod_{i<j}(x_i - x_j) \to \prod_{i<j}(x_{\theta(i)} - x_{\theta(j)}).$$

It is clear that $\theta : p(x_1, \ldots, x_n) \to \pm p(x_1, \ldots, x_n)$. For instance, in S_5, if $\theta = (3\ 4\ 1)(5\ 2)$, then θ takes

$$\begin{aligned} p(x_1, \ldots, x_5) \quad = \quad & (x_1 - x_2)(x_1 - x_3)(x_1 - x_4)(x_1 - x_5) \\ & \times (x_2 - x_3)(x_2 - x_4)(x_2 - x_5) \\ & \times (x_3 - x_4)(x_3 - x_5) \\ & \times (x_4 - x_5) \end{aligned}$$

into

$$(x_3 - x_5)(x_3 - x_4)(x_3 - x_1)(x_3 - x_2)(x_5 - x_4)(x_5 - x_1)(x_5 - x_2)(x_4 - x_1) \\ \times (x_4 - x_2)(x_1 - x_2),$$

which can be easily verified to be $-p(x_1, \ldots, x_n)$.

In particular, if θ is a 2-orbicycle, then $\theta : p(x_1, \ldots, x_n) \to -p(x_1, \ldots, x_n)$. Extending this observation, since a permutation which is a 2-orbicycle is an odd permutation, if θ is an odd permutation, then $\theta : p(x_1, \ldots, x_n) \to -p(x_1, \ldots, x_n)$.

Thus, if a permutation θ can be written as a product of an even number of 2-orbicycles, then θ must leave $p(x_1, \ldots, x_n)$ fixed. So any other way of writing θ as a product of 2-orbicycles must be such that it leaves $p(x_1, \ldots, x_n)$ fixed; which implies that θ is always a product of an even number of 2-orbicycles. □

Based on Lemma 6.14 we obtain the following.

LEMMA 6.15.
(1) *The product of two even permutations is an even permutation.*
(2) *The product of an even permutation and an odd permutation is an odd permutation.*
(3) *The product of two odd permutations is an even permutation.*

EXAMPLE 6.16. Let $\Omega = \{1, \ldots, 9\}$ and let θ be the permutation in Example 6.7. It is easy to see that the orbicycle containing 1 under $\theta = (5\ 6\ 1) = (5\ 1)(6\ 1)$ and therefore this orbicycle is an even permutation. Similarly, the orbicycle containing 2 under $\theta = (4\ 8\ 7\ 3\ 2) = (4\ 2)(7\ 2)(3\ 2)(8\ 2)$ and therefore this orbicycle is also an even permutation. The orbicycle containing 9 under θ has zero 2-orbicycles and therefore this orbicycle is also an even permutation. It is easy to

see that $\theta = (5\ 1)(6\ 1)(4\ 2)(7\ 2)(3\ 2)(8\ 2)$. Since θ is a product of only even permutations it follows that θ is also an even permutation.

EXAMPLE 6.17. Let $\Omega = \{1, \ldots, 9\}$ and let θ' be the permutation in Example 6.8. It is easy to see that the orbicycle containing 1 under θ has zero 2-orbicycles and so this orbicycle is an even permutation. Similarly the orbicycle containing 4 is an even permutation and the orbicycle containing 9 is also an even permutation. However the orbicycle containing 2 under $\theta' = (8\ 7\ 6\ 2\ 5\ 3) = (8\ 2)(3\ 2)(7\ 2)(5\ 2)(6\ 2)$ which means that this orbicycle is an odd permutation. It is easy to see that $\theta' = (8\ 2)(3\ 2)(7\ 2)(5\ 2)(6\ 2)$. Since in writing θ' as a product of its orbicycles, we have odd number of odd permutations it follows that, θ' is an odd permutation.

THEOREM 6.18. *Let A_n denote the set of all even permutations in S_n. A_n is a normal subgroup of S_n of index 2 under multiplication of permutations, and it is called as the* alternating group *of degree n.*

PROOF. A_n is a group since the product of two even permutations is an even permutation.

Let W be the group of real numbers -1 and 1 under multiplication. Define $\psi : S_n \to W$ by $\psi(\theta) = 1$ if θ is an even permutation, $\psi(\theta) = -1$ if θ is an odd permutation. Using Lemma 6.15 it is easy to see that ψ is a homomorphism onto W. The kernel of ψ is precisely A_n. Therefore A_n is a normal subgroup of S_n. As a consequence, $\left(\frac{S_n}{A_n}\right) \approx W$ which implies, $o(W) = 2 = o\left(\frac{S_n}{A_n}\right) = \frac{o(S_n)}{o(A_n)}$. This completes the proof. □

DEFINITION 6.19. Given a permutation $\theta \in S_n$, let p denote the number of 2-orbicycles in θ. We define the sign of θ, denoted by $sgn(\theta)$, as $(-1)^p$.

PROPOSITION 6.20. *Let θ be an even permutation. $sgn(\theta) = 1$. Let σ be an odd permutation. $sgn(\sigma) = -1$.*

DEFINITION 6.21. Given an orbicycle $(y_1\theta\ y_2\theta\ \cdots\ y_t\theta)$ of a permutation θ on $\Omega = \{1, \ldots, n\}$, we say that a pair $(y_i\theta, y_j\theta)$ is an inversion if $y_i\theta > y_j\theta$ whenever $y_i < y_j$. The number of inversions in θ is equal to the sum of the number of inversions in its orbicycles.

THEOREM 6.22.
(1) *Given a permutation $\theta \in S_n$, let i denote the number of inversions in θ. $sgn(\theta) = (-1)^i$.*
(2) *Given a permutation $\theta \in S_n$, $sgn(\theta)$ is equal to the determinant of the permutation matrix of θ.*

PROOF. (1) Consider the proof of Lemma 6.14 and $p(x_1, \ldots, x_n)$ defined therein. Clearly, if a permutation θ is an even permutation, since it is a product of an even number of 2-orbicycles it leaves $p(x_1, \ldots, x_n)$ fixed. Otherwise if θ is an odd permutation, then $\theta : p(x_1, \ldots, x_n) \to -p(x_1, \ldots, x_n)$. In other words, if θ is an even permutation, then θ contains even number of inversions among elements of Ω when it is written as a product of its orbicycles due to which θ maps

$p(x_1, \ldots, x_n)$ to itself. On the other hand, if θ is an odd permutation, then θ contains odd number of inversions among elements of Ω when it is written as a product of its orbicycles due to which it maps $p(x_1, \ldots, x_n)$ to $-p(x_1, \ldots, x_n)$. This shows that $sgn(\theta) = (-1)^{i(\mathrm{mod}\ 2)} = (-1)^i$, where i is the number of inversions in θ.

(2) Follows from the definition of the determinant of a matrix in equation (6.1) given below, and since the matrix is only a permutation matrix whose entries are only 0 and 1. □

EXAMPLE 6.23. In continuation with Example 6.16, $sgn(\theta) = 1$. Also the number of inversions in $\theta = (2 + 8) = 10$, which is an even number. It is easy to see that $sgn(\theta) = (-1)^{10} = 1$. If we consider the permutation matrix of θ, it is

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Clearly, $\det(A) = 1$.

EXAMPLE 6.24. In continuation with Example 6.17, $sgn(\theta') = -1$. Also the number of inversions in $\theta' = 13$, which is an odd number. It is easy to see that $sgn(\theta') = (-1)^{13} = -1$. If we consider the permutation matrix of θ', it is

$$A' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Clearly, $\det(A') = -1$.

6.2. Mahajan-Vinay's Theorems on the Determinant

We recall the definition of the determinant of a matrix. Let $A \in \mathbb{R}^{n \times n}$. We define the *determinant* of A as:

$$\det(A) \quad = \quad \sum_{\sigma \in S_n} sgn(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}. \tag{6.1}$$

In this chapter, given a matrix $A \in \mathbb{R}^{n\times n}$, we view A as a directed graph such that the weight of the directed edge (i,j) is equal to the entry $A(i,j)$, where $1 \leq i,j \leq n$. If $A(i,j) = 0$, then the directed edge (i,j) does not exist in the directed graph.

PROPOSITION 6.25. *Let $A \in \mathbb{R}^{n\times n}$ and let*

$$B = \begin{bmatrix} 0 & -A \\ I_n & 0 \end{bmatrix}.$$

Then, $\det(A) = \det(B)$.

Note that if $B \in \mathbb{R}^{2n\times 2n}$ in Proposition 6.25 is viewed as a directed graph, then there does not exist self-loop on any vertex in the directed graph represented by B. We need this important property to prove our results.

DEFINITION 6.26. Let G be a directed graph on n vertices. A cycle is a sequence of k distinct vertices $C = (v_1, v_2, \ldots, v_k)$ such that $(v_i, v_{i+1}), (v_k, v_1) \in E(G)$, where $1 \leq i \leq k-1$ and $k \leq n$.

DEFINITION 6.27. Let G be a directed graph on n vertices. Given a cycle $C = (v_1, v_2, \ldots, v_k)$ in G, we define the length of C as k and it is denoted by $l(C)$.

DEFINITION 6.28. Let G be a weighted directed graph on n vertices such that $w : E(G) \to \mathbb{R}$ is a weight function on the edges of G. Given a cycle $C = (v_1, v_2, \ldots, v_k)$ in G, we define the weight of the cycle as $w(C) = \prod_{i=1}^{k} w(e_i)$, where $e_i = (v_i, v_{i+1})$, for $1 \leq i \leq k-1$ and $e_k = (v_k, v_1)$.

DEFINITION 6.29. Let G be a directed graph on n vertices. Given a cycle $C = (v_1, v_2, \ldots, v_k)$ in G such that $v_1 \leq v_i$ for all $1 \leq i \leq k$, we say that v_1 is the head of the cycle C and it is denoted by $h(C)$.

DEFINITION 6.30. Let $G = (V, E)$ be a directed graph on n vertices. A walk $W = (v_1, v_2, \ldots, v_l)$ in G is a clow (abbreviation for "closed-walk") starting from the least numbered vertex v_1 and ending at the same vertex such that there is exactly one incoming edge $(v_l, v_1) \in E(G)$ for v_1 in the clow and exactly one outgoing edge $(v_1, v_2) \in E(G)$ for v_1 in the clow, and $(v_i, v_{i+1}) \in E(G)$, where $1 \leq i \leq k-1$.

DEFINITION 6.31. Let $G = (V, E)$ be a directed graph on n vertices and let $W = (v_1, v_2, \ldots, v_l)$ be a clow in G. We define the length of the clow W as the number of edges in the clow, and it is denoted by $l(W)$.

DEFINITION 6.32. Let G be a weighted directed graph on n vertices such that $w : E(G) \to \mathbb{R}$ is a weight function on the edges of G. Given a clow $W = (v_1, v_2, \ldots, v_k)$ in G, we define the weight of the clow as $w(W) = \prod_{i=1}^{k} w(e_i)$, where $e_i = (v_i, v_{i+1})$, for $1 \leq i \leq k-1$ and $e_k = (v_k, v_1)$.

DEFINITION 6.33. Let $G = (V, E)$ be a directed graph on n vertices and let $W = (v_1, v_2, \ldots, v_l)$ be a clow in G such that the vertex v_1 is the least numbered vertex in W. We define v_1 as the head of the clow W and it is denoted by $h(W)$.

In a directed graph, it is easy to see that every cycle is a clow.

PROPOSITION 6.34. *Let $G = (V, E)$ be a directed graph on n vertices and let $W = (v_1, v_2, \ldots, v_l)$ be a clow in G. Then v_1 occurs exactly once in the clow.*

DEFINITION 6.35. Let $G = (V, E)$ be a directed graph. A clow sequence $\mathcal{W}$ of G is a sequence of clows $\mathcal{W} = (C_1, \ldots, C_k)$ such that the sequence is ordered based on heads of clows, that is head$(C_1) <$ head$(C_2) < \cdots <$ head(C_k), and every vertex of G is in at least one C_i, where $1 \leq i \leq k$.

A directed graph can have infinitely many clow sequences; for example, see Figure 6.1.

DEFINITION 6.36. Let $G = (V, E)$ be a directed graph and let $\mathcal{W} = (C_1, \ldots, C_k)$ be a clow sequence of G. We define the length of the clow sequence $\mathcal{W}$ as the sum of the lengths of the clows in $\mathcal{W}$ and it is denoted by $l(\mathcal{W})$.

DEFINITION 6.37. Let $G = (V, E)$ be a weighted directed graph such that $w : E \to \mathbb{R}$ is a weight function on the edges of G. The weight of a clow sequence $\mathcal{W} = (C_1, \ldots, C_m)$ is defined as $w(\mathcal{W}) = \prod_{i=1}^{m} w(C_i)$.

DEFINITION 6.38. Let $G = (V, E)$ be a directed graph. A clow sequence $\mathcal{W} = (C_1, \ldots, C_k)$ is a cycle cover of G if every C_i is a cycle and there does not exist any vertex which is common to C_i and C_j, where $1 \leq i, j \leq k$ and $i \neq j$.

DEFINITION 6.39. Let G be a directed graph on n vertices. Let $x, y_1, \ldots, y_t \in \{1, \ldots, n\}$ and $\sigma \in S_n$ such that $(y_1\sigma\ y_2\sigma\ \cdots\ y_t\sigma)$ is the orbicycle of x under σ. We say that the orbicycle of x under σ satisfies the graph G if $(y_i, y_i\sigma) \in E(G)$ for all $1 \leq i \leq t$.

DEFINITION 6.40. Let G be a directed graph on n vertices. A permutation σ in S_n is called an orbicycle cover of G if every orbicycle of σ satisfies G.

LEMMA 6.41. *Let G be a directed graph on n vertices. If a clow sequence $\mathcal{W} = (C_1, \ldots, C_k)$ is a cycle cover of G then we can obtain $\sigma \in S_n$ from $\mathcal{W}$ which is an orbicycle cover of G. Conversely, if $\sigma \in S_n$ is an orbicycle cover of G then we can obtain a clow sequence $\mathcal{W}$ from σ which is also a cycle cover of G.*

There exists a one-to-one and onto mapping between cycle covers of G and permutations in S_n which are orbicycle covers of G.

PROOF. Let $\mathcal{W} = (C_1, \ldots, C_k)$ be a clow sequence which is also a cycle cover of G. $\mathcal{W}$ is a collection of cycles such that head$(C_1) <$ head$(C_2) < \cdots <$ head(C_k). Let i_l denote $l(C_i)$. $\sum_{l=1}^{k} i_l = n$ and $1 \leq i_l \leq n$, where $1 \leq i \leq k$.

Let $C_i = (u_{i,1}, \ldots, u_{i,i_l})$, where $u_{i,1} =$ head(C_i), $u_{i,j} \in V$, $1 \leq u_{i,j} \leq n$, $1 \leq i \leq k$ and $1 \leq j \leq i_l \leq n$, $1 \leq l \leq k$.

Define $\sigma : V \to V$ such that for $1 \leq i \leq k$, $\sigma(u_{i,j}) = u_{i,j+1}$ if $1 \leq j < i_l$ and $\sigma(u_{i,i_l}) = u_{i,1}$. Since $\mathcal{W}$ is a cycle cover, individual clows of $\mathcal{W}$ are cycles

and each vertex of G is in exactly one cycle in $\mathcal{W}$. Therefore, σ is a permutation of $V(G)$. It is easy to see from the definition of σ that if we write σ as a product of orbicycles then every orbicycle satisfies G. This implies that $\sigma \in S_n$ is an orbicycle cover of G which we have obtained from $\mathcal{W}$.

Conversely if σ is an orbicycle cover of G, then it follows from the definition of an orbicycle cover (Definition 6.40) that every orbicycle of σ satisfies G. We therefore obtain cycles in G corresponding to each orbicycle. By arranging vertices in each cycle with the head of the cycle as the least numbered vertex, and ordering the collection of cycles based on their heads, we in turn obtain a clow sequence which is also a cycle cover of G.

It is easy to see that this mapping between cycle covers of G and permutations in S_n which are orbicycle covers of G is in fact a one-to-one and onto mapping. □

DEFINITION 6.42. Let G be a directed graph on n vertices.

(1) Let $\mathcal{W} = (C_1, \ldots, C_k)$ be a cycle cover in G and let $\sigma \in S_n$ be the permutation of $V(G)$ which we obtain from $\mathcal{W}$ such that σ is an orbicycle cover of G. We define the sign of the cycle cover $\mathcal{W}$ as $sgn(\sigma)$.
(2) We define the sign of a clow sequence $\mathcal{W} = (C_1, \ldots, C_k)$ which is not a cycle cover, called the clowsign of $\mathcal{W}$ and denoted by $clowsgn(\mathcal{W})$, as $(-1)^m$ where $0 \leq m \leq k$ is the number of components of $\mathcal{W}$ which does not include fixed point cycles that are clows in $\mathcal{W}$. We say that $\mathcal{W}$ is a clow sequence that has positive sign if m is even; otherwise m is odd and $\mathcal{W}$ is called as a clow sequence that has negative sign.

If $\mathcal{W}$ is a cycle cover then $\mathcal{W}$ is also a clow sequence. Therefore, without loss of generality we denote the sign of a cycle cover $\mathcal{W}$ also as $clowsgn(\mathcal{W})$.

THEOREM 6.43. *Let $A \in \mathbb{R}^{n \times n}$ be the adjacency matrix of a weighted directed graph G. Then,*

$$\det(A) = \sum_{\mathcal{C}: a\ cycle\ cover} clowsgn(\mathcal{C}) w(\mathcal{C}). \tag{6.2}$$

PROOF. It follows from the definition of determinant of A given in equation 6.1, the definition of the weight of a cycle cover (Definition 6.32) and the definition of the clowsign of a cycle cover as the sign of the permutation of the vertices of G obtained from the cycle cover (Definition 6.42). □

THEOREM 6.44. (**Mahajan-Vinay, Theorem A**) *Let $A \in \mathbb{R}^{n \times n}$ be the adjacency matrix of a weighted directed graph G such that $A(i,i) = 0$ for all $1 \leq i \leq n$. Now,*

$$\det(A) = \sum_{\mathcal{W}: a\ clow\ sequence} clowsgn(\mathcal{W}) w(\mathcal{W}). \tag{6.3}$$

PROOF. Given the matrix $A \in \mathbb{R}^{n \times n}$, our assumption in the statement of this theorem that $A(i,i) = 0$, for all $1 \leq i \leq n$, is very important for its proof which follows below. We prove this theorem by showing that the contribution of clow sequences that are not cycle covers is zero on the right-hand side of the above

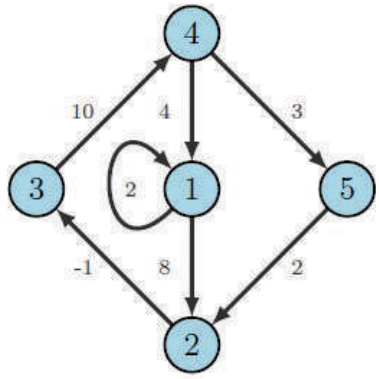

FIGURE 6.1. This figure is a directed graph $G_1 = (V_1, E_1)$. The following are clow sequences of G_1: $\mathcal{C}_1 = ((1,2,3,4),(2,3,4,5))$, $\mathcal{C}_2 = (1,2,3,4,5,2,3,4)$, $\mathcal{C}_3 = ((1),(2,3,4,5))$. Note that $\mathcal{C}_3$ is a cycle cover of G_1. It is easy to note the following facts about clows of these clow sequences in G_1: $l((1,2,3,4)) = l((2,3,4,5)) = 4$, $l((1,2,3,4,5,2,3,4)) = 9$ and $l((1)) = 1$. Here $w(\mathcal{C}_1) = w(\mathcal{C}_2) = 19200$ and $w(\mathcal{C}_3) = -120$. Since $\mathcal{C}_3$ is a cycle cover we can obtain from $\mathcal{C}_3$, the following orbicycle cover permutation denoted by σ, of the vertices of G_1 based on the traversal of vertices of the individual cycles of $\mathcal{C}_3$: $(1)(3\ 4\ 5\ 2)$. It is easy to note that if σ is written as a product of 2-orbicycles then $\sigma = (3\ 2)(4\ 2)(5\ 2)$. Since there are an odd number of 2-orbicycles it follows that $sgn(\mathcal{C}_3) = -1$. Clearly there are 3 inversions in σ $(3 > 2, 4 > 2, 5 > 2)$. So we once again infer that $sgn(\sigma) = -1$. Also we note the permutation matrix corresponding to σ has determinant is -1. We get $clowsgn(\mathcal{C}_1) = 1$ while $clowsgn(\mathcal{C}_2) = -1$ and $clowsgn(\mathcal{C}_3) = -1$. The determinant of the adjacency matrix of G_1=120. Note that G_1 has infinitely many clow sequences, each of which can be obtained by starting from the vertex 1 and iteratively including edges from the cycle $(2,3,4,5)$ without ending a clow using the edge $(4,1)$.

equation. Consequently, only cycle covers contribute to the summation yielding exactly the determinant of A using Theorem 6.43.

We define a mapping called an involution (denoted by φ) on the set of clow sequences of G. An involution φ on a set is a bijection with the property that φ^2 is the identity map on the set. The involution that we define has the domain to be the set of all clow sequences. Also φ^2 is the identity mapping on the set of all clow sequences, φ maps a cycle cover to itself and otherwise maps a clow sequence $\mathcal{W}$

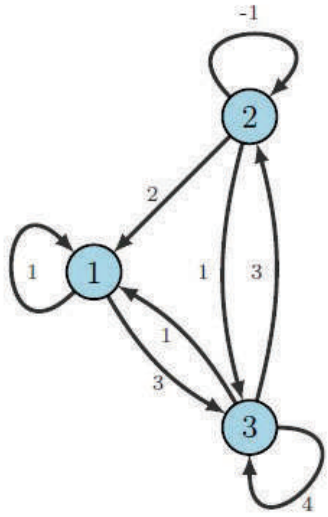

FIGURE 6.2. This figure is a directed graph $G_2 = (V_2, E_2)$. The following are clow sequences, which are cycle covers of G_2: $\mathcal{C}_1 = ((1),(2),(3))$, $\mathcal{C}_2 = ((1),(2,3))$, $\mathcal{C}_3 = ((1,3,2))$, and $\mathcal{C}_4 = ((1,3),(2))$. Examples of clow sequences which are not cycle covers of G_2: $\mathcal{C}_5 = (1,3,2,3,2)$ and $\mathcal{C}_6 = (1,3,2)(2,3)$. Here $l(\mathcal{C}_1) = l(\mathcal{C}_2) = l(\mathcal{C}_3) = l(\mathcal{C}_4) = 3$ and $l(\mathcal{C}_5) = l(\mathcal{C}_6) = 5$. Also $w(\mathcal{C}_1) = -4$, $w(\mathcal{C}_2) = 3$, $w(\mathcal{C}_3) = 18$, $w(\mathcal{C}_4) = -3$, and $w(\mathcal{C}_5) = w(\mathcal{C}_6) = 54$. Permutations that we obtain from $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$, and $\mathcal{C}_4$ are $(1)(2)(3)$, $(1)(3\ 2)$, $(3\ 1\ 2)$, and $(3\ 2\ 1)$ respectively. Therefore, $clowsgn(\mathcal{C}_1) = 1$, $clowsgn(\mathcal{C}_2) = -1$, $clowsgn(\mathcal{C}_3) = 1$ and $clowsgn(\mathcal{C}_4) = -1$. Also, $clowsgn(\mathcal{C}_5) = -1$ and $clowsgn(\mathcal{C}_6) = 1$. The determinant of the adjacency matrix of G_2 is 14 and it satisfies the equation 6.2 in Theorem 6.43. Also note that equation 6.3 in Theorem 6.44 is also satisfied.

to another clow sequence $\varphi(W)$ such that $\mathcal{W}$ and $\varphi(\mathcal{W})$ have the same weight but opposite clowsigns as follows.

Let $\mathcal{W} = (C_1, \ldots, C_m)$ be a clow sequence. Now, choose the smallest $1 \leq i \leq m$ such that in $\mathcal{W}$, we have $(C_{i+1}, \ldots, C_m)$ is a set of vertex disjoint cycles in G. If $i = 0$ then any such clow sequence $\mathcal{W}$ is a cycle cover and the involution φ maps $\mathcal{W}$ to itself. Otherwise, if such an $1 \leq i \leq m$ exists, then we traverse the clow C_i starting from the head until at least one of the following two possibilities occur:

(1) we visit a vertex that touches one of the cycles in $(C_{i+1}, \ldots, C_m)$,
(2) we visit a vertex that is a part of a cycle within C_i.

Without loss of generality, we consider the first such vertex and let it be v. It is not difficult to note that some vertex v might satisfy both of these above stated

possibilities simultaneously. If the clow C_i has such a vertex v we employ case 1 that follows when considering any such vertex v. We now consider these two possibilities:

Case 1: (merging of clows) Suppose v touches C_j. We map $\mathcal{W}$ to a clow sequence

$$\mathcal{W}' = (C_1, \ldots, C_{i-1}, C_i', C_{i+1}, \ldots, C_{j-1}, C_{j+1}, \ldots, C_m).$$

The modified clow sequence C_i' is obtained by merging C_i and C_j as follows: insert the cycle C_j into C_i at the first occurrence (from the head) of v. For example, let $C_i = (8, 11, 10, 14)$ and $C_j = (9, 10, 12)$. Then the new clow is $(8, 11, 10, 12, 9, 10, 14)$. Figure 6.3 illustrates the mapping.

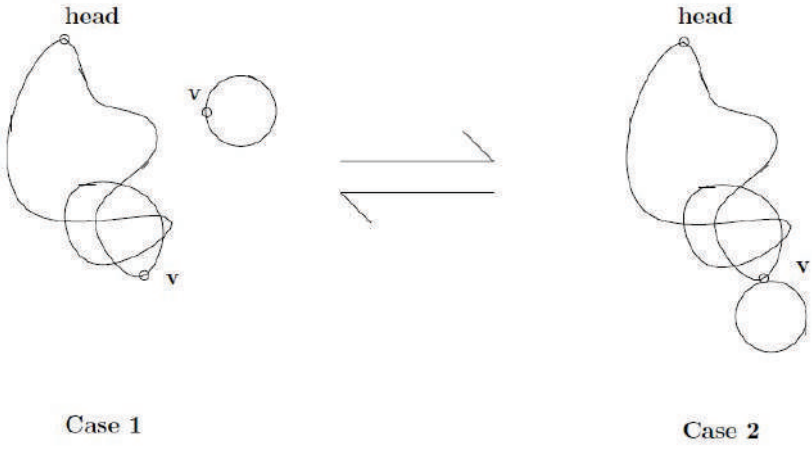

FIGURE 6.3. Pairing clow sequences of opposite signs

The head of C_i' is clearly the head of C_i. The new sequence has the same multiset of edges and hence the same weight as the original sequence. It also has one component less than the original sequence $\mathcal{W}$.

Case 2: (splitting of clows) Suppose v completes a cycle C in C_i.

We now modify the sequence $\mathcal{W}$ by deleting C from C_i and introducing C as a new clow in an appropriate position, depending on the minimum labeled vertex in C, which we make the head of C. For example, let $C_i = (8, 11, 10, 12, 9, 10, 14)$. Then C_i changes to $(8, 11, 10, 14)$ and the new cycle $(9, 10, 12)$ is inserted in the clow sequence.

To show that the modified sequence continues to be a clow sequence, note that the head of C is greater than the head of C_i; hence C occurs after C_i. Also the head of C is distinct from the heads of C_j $(i < j \leq m)$ since v does not touch any of the clows C_j $(i < j \leq m)$ for otherwise we would have merged clows as mentioned in case 1. In fact, C is disjoint from all cycles C_j $(i < j \leq m)$. Further the new sequence has the same multiset of edges and hence the same weight as the original sequence. It also has exactly one component more than the original

sequence. Figure 6.3 illustrates the mapping. In the new clow sequence, vertex v in cycle C_i' would have been chosen by our traversal, and it satisfies case 1.

In both of the above cases, the new clow sequence constructed maps back to the original clow sequence. Therefore the mapping is a weight preserving involution. Furthermore, the number of clows in the original sequence and in the new sequence in both these cases differ by one and there are no length 1 cycles in G. So they have opposite clowsigns. As a result the contribution of the weight of clow sequences which are not cycle covers on the right hand side in the statement of this theorem get canceled. This completes the proof. □

6.2.1. Constructing a special weighted layered directed acyclic graph H. Given a $n \times n$ matrix A, we define a weighted layered directed acyclic graph H with three special vertices s, t_+ and t_- that will satisfy the following property:

$$\det(A) \quad = \quad \sum_{\rho: s \rightsquigarrow t_+} w(\rho) - \sum_{\eta: s \rightsquigarrow t_-} w(\eta). \tag{6.4}$$

Here the weight of a walk is simply the product of the weights of the edges appearing in it. The idea is that there exists a one-to-one mapping from the set of all cycle covers of positive sign of the directed graph G represented by the matrix A to the set of all walks $s \rightsquigarrow t_+$ in H. Similarly there exists a one-to-one mapping from the set of all cycle covers of negative sign of the directed graph G represented by the matrix A to the set of all walks $s \rightsquigarrow t_-$ in H. To prove the above statement, given the matrix $A \in \mathbb{R}^{n \times n}$, we first obtain the matrix $B \in \mathbb{R}^{2n \times 2n}$ using Proposition 6.25 such that $B(i,i) = 0$, where $1 \leq i \leq 2n$. Therefore, without loss of generality for the rest of this section, we assume that the input matrix $A \in \mathbb{R}^{n \times n}$ and $A(i,i) = 0$, for all $1 \leq i \leq n$. In other words, if G denotes the directed graph represented by A then there are no self-loops on vertices in G.

We can view the directed graph H in 3-dimensions as follows: there exists a vertex s above and in between the two 3-dimensional cubes of dimension $n \times n \times n$ each, and two other vertices t_- and t_+ below the two cubes. The cube above the vertex t_+ is called as the 0-cube and the cube above the vertex t_- is called as the 1-cube. Every vertex in the two 3-dimensional cubes of this graph H is represented by a 4-tuple: (p, i, h, u). The 1^{st} component p of the 4-tuple (p, i, h, u) of a vertex denotes if a path which starts from s is in the 0-cube or the 1-cube. The 2^{nd} component i of the 4-tuple (p, i, h, u) of a vertex denotes the layer in which the path that is traversed starting from s is present. The 3^{rd} component h of the 4-tuple (p, i, h, u) of a vertex denotes the head of the clow that is traversed in G. The 4^{th} component u of the 4-tuple (p, i, h, u) of a vertex denotes the vertex of the clow in G with head h that is traversed. The set of vertices of H is therefore $\{s, t_+, t_-\} \cup \{[p, i, h, u] | p \in \{0, 1\}, h, u, i \in \{1, \ldots, n\}\}$.

We will traverse walks that are clow sequences in the directed graph G represented by A. This is mapped to directed paths in H starting from s. If a clow sequence in G has a clow with head h and the clow we are traversing is in vertex u then it is mapped to a vertex (p, i, h, u) in H. The first n edges of any such clow

sequence is mapped to paths from s in H and these paths have either t_- or t_+ as the destination vertex.

The edge set of H consists of the following edges:

(1) $(s, [0, 1, h, h])$ for $h \in \{1, \ldots, n\}$; this edge has weight 1,
(2) $([p, i, h, u], [\overline{p}, i + 1, h, v])$ if $i < n$ and $v > h$; this edge has weight $A(u, v)$,
(3) $([p, i, h, u], [p, i + 1, h', h'])$ if $i < n$ and $h' > h$; this edge has weight $A(u, h)$,
(4) $([1, n, h, u], t_-)$. Here, if $u > h$ and the directed edge (u, h) exists in the directed graph G represented by A then this edge exists and it has weight $A(u, h)$, and
(5) $([0, n, h, u], t_+)$. Here, if $u > h$ and the directed edge (u, h) exists in the directed graph G represented by A then this edge exists and it has weight $A(u, h)$.

In this definition of H, if we have not created a directed edge from a vertex a to a vertex b or that we have not mentioned the weight of the directed edge (a, b) then the weight of the directed edge (a, b) in H is assumed to be 0. Given the original matrix $A \in \mathbb{R}^{n \times n}$, after reducing A to B as in Proposition 6.25 and obtaining H, we have the number of vertices in H is $2(2n)^3 + 3 = 16n^3 + 3$ and the number of edges in H is at most $4n^4$. Without loss of generality, in this weighted layered directed acyclic graph H, we say that there are $(n + 2)$ layers, vertex s is in the layer 0, vertices t_+ and t_- are in the layer $n + 1$, and the vertex $[p, i, h, u]$ is in the layer i, where $1 \leq i \leq n$. Since H is a weighted layered directed acyclic graph, it follows that any walk from s to t_+ in H is actually a directed path. Similarly any walk from s to t_- in H is actually a directed path.

6.2.2. Facts about directed paths from s in the special directed graph H.

(1) Any directed path from s always starts from a vertex (represented by the 4-tuple $[0, 1, h, h]$) in layer 1 of the 0-cube. In the starting vertex of the cubes, (h, h) denotes that we are starting to traverse a walk in G which is in fact a clow sequence and that h is the head of the clow and we are at present in h.
(2) The second component of a vertex $[p, i, h, u]$ also denotes the layer of the cube we are at present in. It also denotes (1+the number of edges that we have traversed in the clow sequence of G).
(3) If we are traversing a clow sequence of G, then it is first mapped to a directed path in H starting from s. Also we traverse the edges of the directed graph H for only the first n edges of the clow sequence of G. This is important since every cycle cover of G has exactly n edges and such a mapping of the first n edges of clow sequences to directed paths from s suffices for us to prove Theorem 6.45.
(4) In the mapping from traversing a clow sequence in G to a directed path from s in H, we trace a directed path in H connecting vertices along edges which alternate between the vertices of the 0-cube and the 1-cube of H. This is due to premise (2) in the definition of H in Section 6.2.1.

(5) If a clow of a clow sequence ends then we start the next clow of the clow sequence in the next layer of the same cube of the directed graph H as defined in the premise (3) in the definition of H in Section 6.2.1.
(6) Premises $\{(4)$ and $(5)\}$ deal with the layer n of the cubes and state those edges that have non-zero weight which connect from the n^{th} layer of the cubes to vertices t_+ and t_-.

6.2.3. Determinant, the special directed graph H and GapL.

THEOREM 6.45. (**Mahajan-Vinay, Theorem B**) *Let $A \in \mathbb{R}^{n\times n}$ and let* H *be the weighted layered directed acyclic graph described above. Then,*

$$\det(A) \quad = \quad \sum_{\rho:s\rightsquigarrow t_+} w(\rho) - \sum_{\eta:s\rightsquigarrow t_-} w(\eta). \tag{6.5}$$

PROOF. To prove this theorem, as mentioned in Section 6.2.1, given the matrix $A \in \mathbb{R}^{n\times n}$, we first obtain the matrix $B \in \mathbb{R}^{2n\times 2n}$ using Proposition 6.25 such that $B(i,i) = 0$, where $1 \leq i \leq 2n$. Therefore, without loss of generality for the rest of this proof, we assume that the input matrix $A \in \mathbb{R}^{n\times n}$ and $A(i,i) = 0$, for all $1 \leq i \leq n$. In other words, if G denotes the directed graph represented by A then there are no self-loops on vertices in G.

Case 1: We first show that there exists a one-to-one mapping from the set of all cycle covers of positive clowsign in the directed graph G represented by the matrix A into the set of all paths $s \rightsquigarrow t_+$ in H. Similarly there exists a one-to-one mapping from the set of all cycle covers of negative clowsign in the directed graph G represented by the matrix A into the set of all paths $s \rightsquigarrow t_-$ in H.

Let $\mathcal{W} = (C_1, \ldots, C_k)$ be a cycle cover of G. Clearly, C_i are a collection of vertex disjoint cycles such that $h(C_1) < h(C_2) < \cdots < h(C_k)$. Using Lemma 6.41, it follows that cycle covers of G whose weight is non-zero yield permutations of vertices of G whose weight is non-zero and the same weight respectively. Clearly the clowsign of a cycle cover is the sgn of the permutation which we obtain from the cycle cover. It follows from the definition of H that from s we move to the vertex $[0, 1, h_1, h_1]$ and the weight of this edge is 1. In the sequence of vertices visited along this path, the first component denotes the cube we are at present in. The second component is the layer of the cube we are present in. It also denotes 1+the number of edges that we have traversed in the clow sequence of G and it is monotonically increasing. Also the third component is the head of the clow that is traversed in G and this component is also monotonically non-decreasing and takes, say k distinct values $h_1, \ldots, h_k$. Consider the maximal segment of the directed path in this clow with the third component h_i. The fourth component on this maximal segment is a vertex of the clow with the head h_i in G. When this clow is completely traversed, a new clow with a larger head must be started. Depending on whether we have the clowsign of the cycle cover is positive or negative, we end the directed path which started from s in t_+ or in t_- respectively. This is precisely modeled by the edges of H. Clearly, the weight of the path from s to t_+ or to t_- is equal to the weight of the cycle cover which is the product of the weights of the edges in the cycle cover. Also no two distinct cycle covers of the same clowsign

will yield the same directed path from s to vertices t_- or t_+. Since it is not possible to either merge clows (which are in fact pair of individual disjoint cycles) in cycle covers or to split clows (individual cycles) in cycle covers, we get this mapping to be a one-to-one mapping.

Case 2: We consider the case of a clow sequence $\mathcal{W} = (C_1, \ldots, C_k)$ which is not a cycle cover. Such clow sequences are not fully traversed by the directed path that starts from s since any directed path in H is of length only $n + 1$ and its first edge from s to a vertex in layer 1 of the 0-cube is not a part of the clow sequence. As a result only the first n edges of the clow sequence are traversed using edges of, say a $s \rightsquigarrow t_+$ path in H. We observe here also using the same argument as in the case of cycle covers that no two partial clow sequences, which are different and form only the first n edges of the actual clow sequences of G, will result in the same directed path from s to t_+ or from s to t_-.

We note that if a clow of a clow sequence which has length $> n$ does not end a clow at the n^{th} edge, then the weight of the clow sequence is 0. This property is preserved due to the definition of H and its edges from the layer n to vertices t_+ or t_-. Based on if k is even or odd, and hence the sign of $\mathcal{W}$ is positive or negative respectively, we will demonstrate that there exists a directed path from s to t_+ or from s to t_-, respectively in H.

Now, without loss of generality, let us consider a clow sequence $\mathcal{W}$ of negative clowsign which is not a cycle cover and the partial clow sequence $\mathcal{W}'$ obtained using the first n edges of this clow sequence $\mathcal{W}$ such that the n^{th} edge of $\mathcal{W}$ ends a clow in $\mathcal{W}$. Using the merge operation on a pair of clows or the split operation on a clow of $\mathcal{W}'$, we can therefore obtain another partial clow sequence $\mathcal{W}''$ such that $w(\mathcal{W}') = w(\mathcal{W}'')$ and $clowsgn(\mathcal{W}')$ and $clowsgn(\mathcal{W}'')$ are opposite to each other.

Claim: We claim that the number of negative partial clow sequences that we can obtain starting from any such partial clow sequence $\mathcal{W}'$ containing exactly n edges of a clow sequence $\mathcal{W}$ is equal to the number of positive partial clow sequences that we can obtain starting from the same partial clow sequence $\mathcal{W}'$ of the clow sequence $\mathcal{W}$.

Assuming that the above claim is true, it is easy to see that all these partial clow sequences, irrespective of their sign, have the same weight. As a result an equal number of positive partial clow sequences and negative partial clow sequences will cancel each other in the right-hand side of the equation in the theorem statement as in the proof of Theorem 6.44, yielding the fact that any non-zero contribution to the equation is always because of the sign and weight of cycle covers only. Our theorem therefore follows from Theorem 6.43.

Proof of claim: To prove our statement regarding partial clow sequences that contains exactly n edges, we first define a relation $\sim$ on such partial clow sequences such that any two partial clow sequences $\mathcal{W}_1$ and $\mathcal{W}_2$ are related if and only if we can obtain $\mathcal{W}_2$ from $\mathcal{W}_1$ using a merge operation on a pair of clows of $\mathcal{W}_1$ or a split operation on a clow of $\mathcal{W}_1$. It is clear that $\sim$ is symmetric. Also $\mathcal{W}_1$ and $\mathcal{W}_2$ have the same number of edges n and the same weight. However their signs are opposite to each other.

We now define a bipartite graph G' whose set of vertices is the set of all such partial clow sequences that are related by the relation $\sim$ such that partial clow sequences of positive clowsign are in one partition U and partial clow sequences of negative clowsign are in the other partition V. Also we put a directed edge from a vertex $u \in U$ to a vertex $v \in V$ or vice-versa if the two partial clow sequences represented by these two vertices u and v are related according to $\sim$. It is clear that since there are no self-loops in the directed graph represented by the input matrix A, we do not obtain a partial clow sequence of the same clowsign following a merge operation of a pair of clows of a partial clow sequence or a split operation of a clow of a partial clow sequence. As a result any directed edge in G' is always from a vertex in one partition to a vertex in the other partition. So our graph G' is bipartite. Note that $\sim$ is symmetric and so whenever there exists a directed edge between two vertices in G', there exists a back edge also. Therefore we can treat this bipartite graph G' as an undirected bipartite graph. Also this undirected bipartite graph G' is connected since any partial clow sequence represented by a vertex is always related to any other partial clow sequence by a sequence of merge or split operations and so there exists an undirected path connecting any two vertices in G'.

Now, for any vertex $u \in U$, let m denote the number of pairs of clows in the clow sequence represented by u that can be merged. Similarly let s denote the number of clows in the clow sequence represented by u that can be split. Then degree of u is $m + s$. Let $p = m + s$. It is easy to observe that if $v \in V$ is a neighbour of u in G' then the degree of v is also p. In fact the undirected bipartite graph G' is (p, p)-regular. Using a simple counting argument on the number of edges in G' along its partitions U and V, we can show that any such (p, p)-regular undirected bipartite graph G' has the same number of vertices in both the partitions U and V. This shows that the number of partial clow sequences of positive sign and the number of partial clow sequences of negative sign that can be obtained from any such partial clow sequence $\mathcal{W}'$ are equal from which the claim follows. □

THEOREM 6.46. *Let $A \in \mathbb{Z}^{n\times n}$. Computing $\det(A)$ is in* GapL.

PROOF. It is easy to see that given matrix A as input, we can obtain the adjacency matrix of the weighted layered directed acyclic graph H defined in Section 6.2.1 in using space which is logarithmic in the size of A. As a result it follows that the proof of Theorem 6.45 actually shows that the problem of computing $\det(A)$ is logspace many-one reducible to the problem GapDSTCON which is shown to be complete for GapL in Chapter 4. This completes the proof. □

THEOREM 6.47. *Let $A \in \mathbb{Z}^{n\times n}$. Computing $\det(A)$ is logspace many-one hard for* GapL.

PROOF. In Chapter 4, we have shown that the problem GapSLDAGSTCON is logspace many-one complete for GapL. Let (G, s, t_+, t_-) be an input instance of GapSLDAGSTCON. Let n denote the number of layers in G. We assume without loss of generality that G is not an empty graph. We have to compute the quantity D, which is number of directed paths from s to t_+ minus the number of directed paths

from s to t_- in G. We show that it is possible to obtain a matrix A in $O(\log N)$ space such that $\det(A) = D$, where N is the size of the input instance.

For this, first we obtain a new graph G' from G as follows. We first replace each directed edge in G which exists between vertices of adjacent layers in G by a directed path of length 2 such that both these edges have weight 1. We then add a vertex x and the following edges of weight 1: $\{(t_+, s), (t_-, x), (x, s)\}$, and self-loops having weight 1 on all vertices of G' except s. Clearly G' contains $(2n-1)$ layers, where the $(2n-1)^{th}$ layer contains $(n+1)$ vertices including x. G' contains $(n^2 + |E(G)| + 1)$ vertices. Also G' contains $(2|E(G)| + 3)$ ordinary edges, $(n^2 + |E(G)|)$ self-loops. Therefore G' contains $(n^2 + 3|E(G)| + 3)$ edges. Since the graph G' is still a weighted layered directed graph, we assume that the vertices in G' are labeled by a pair of indices (i, j) such that the vertex (i, j) is the j^{th} vertex in layer i, where $1 \leq i \leq 2(n-1)$ and $1 \leq j \leq n$. The vertex s is in the first layer of G', and vertices t_+ and t_- are in the last layer of G'. We also assume an ordering of the vertices first based on the layers and then from the left-most vertex to the right-most vertex in each layer.

It follows from the definition of G' that each $s \rightsquigarrow t_+$ path in G corresponds to a cycle of odd length in G' which contains s and t_+. Due to self-loops on all vertices of G' except s, using self-loop on any vertex which is not in this directed cycle, we therefore get a cycle cover in G'. Using Lemma 6.41 it follows that every such cycle cover in turn yields an even permutation of vertices of G'. As a result the sign of every such permutation is positive. Similarly, each $s \rightsquigarrow t_-$ path in G corresponds to a cycle cover of even length in G' which in turn yields a permutation of the vertices of G' of negative sign in G'. Conversely, we note that any cycle cover in G' should have a cycle containing s and since there is no self-loop on s, any cycle which contains s should contain the vertex t_+ and the directed edge (t_+, s), or the cycle should contain both vertices x and t_- and directed edges (t_-, x) and (x, s). We have therefore shown that there exists a directed path from s to t_+ in G if and only if there exists a cycle cover in G' containing s and t_+. Similarly, there exists a directed path from s to t_- in G if and only if there exists a cycle cover in G' containing s and vertices x and t_-.

Let A denote $adj(G')$, the adjacency matrix of G'. It is also easy to note that the weight of the cycle cover in G' is equal to the product of edge weights which is the product of entries of A. Since any newly added edge to G used to obtain G' has weight 1, it follows from the definition of the determinant (equation 6.1), Theorem 6.43 and Theorem 6.45 that $\det(A)$ equals D. Since we can obtain the matrix A from the input instance of GapSLDAGSTCON using at most $O(\log n)$ space, where n is the size of the input instance, we have proved our theorem. □

COROLLARY 6.48. *Let $A \in \mathbb{Z}^{n \times n}$. Computing $\det(A)$ is logspace many-one complete for* GapL.

COROLLARY 6.49. *Let $A \in \mathbb{Z}^{n \times n}$. Computing $\det(A)$ is logspace many-one hard for* ♯L.

PROOF. The proof of this result is similar to Theorem 6.47. In Chapter 4, we have shown that the problem ♯SLDAGSTCON is logspace many-one complete for

$\sharp$L. We show that this problem is logspace many-one reducible to the problem of computing the determinant of an integer matrix.

Let (G, s, t) be an input instance of $\sharp$SLDAGSTCON. We show that it is possible to obtain a matrix A in $O(\log n)$ space such that $\det(A) = N$, where n is the size of the input instance and N is the number of directed paths from s to t in G. We first replace each directed edge in G which exists between vertices of adjacent layers in G by a directed path of length 2 such that both these edges have weight 1. We then add the following edges of weight 1: (t, s) and self-loops having weight 1 on all vertices except s. Let the resulting directed graph be G'. Now any $s \rightsquigarrow t$ path in G yields a cycle cover in this new graph which further yields a permutation of the vertices of G'. As in Theorem 6.47, here we note that permutations which are obtained from cycle covers have positive sign. Also there are no permutations which we obtain that have negative sign. Therefore, as in Theorem 6.47, it follows from the definition of the determinant (equation 6.1), Theorem 6.43 and Theorem 6.45 that the number of $s \rightsquigarrow t$ paths in G is equal to the $\det(adj(G'))$, where $adj(G')$ denotes the adjacency matrix of G'. Since we can obtain A from the input instance of $\sharp$SLDAGSTCON using at most $O(\log n)$ space, where n is the size of the input instance, our theorem follows. □

6.3. Applications of computing the Determinant

In this section, we assume familiarity with basic notions of vector spaces over a field such as linear independence and linear dependence of vectors. We get the following results on linear algebraic problems.

THEOREM 6.50. (1) *Let $A \in \mathbb{Z}^{n\times n}$. Determining if $\det(A) = 0$ is logspace many-one complete for* $\mathrm{C_{=}L}$.

(2) *Given a set of vectors $S = \{\vec{\mathbf{v}}_1, \ldots, \vec{\mathbf{v}}_k\}$ with integer entries over the field of rationals $\mathbb{Q}$, determining if a column in S is in the lexicographically least set of linearly independent vectors in S over $\mathbb{Q}$ is in $\mathrm{L}^{\mathrm{C_{=}L}}$.*

(3) *Let $A \in \mathbb{Z}^{n\times n}$. Computing the $rank(A)$ is in $\mathrm{L}^{\mathrm{C_{=}L}}$.*

(4) *Given $(A, \vec{\mathbf{b}})$, where $A \in \mathbb{Z}^{m\times n}$ and $\vec{\mathbf{b}} \in \mathbb{Z}^{m\times 1}$, determining if there exists $\vec{\mathbf{x}} \in \mathbb{Q}^{n\times 1}$ such that $A\vec{\mathbf{x}} = \vec{\mathbf{b}}$ is in $\mathrm{L}^{\mathrm{C_{=}L}}$ and it is logspace many-one hard for* $\mathrm{C_{=}L}$.

PROOF. (1) It follows from Corollary 6.48 and Definition 2.32.

(2) We use the greedy method of iteratively picking columns from S in a lexicographically least manner such that the rank of the sub-matrix formed continues to increase. We simultaneously determine if the column that we want is in the lexicographically least subset of linearly independent columns S. Clearly, using a $O(\log n)$-space bounded deterministic Turing machine with access to a $\mathrm{C_{=}L}$ oracle, we can therefore obtain the lexicographically least subset of columns in S.

(3) Since we know that the $rank(A)$ is the cardinality of the lexicographically least subset of linearly independent columns of $A \in \mathbb{Z}^{n\times n}$ over $\mathbb{Q}$

and finding the size of this subset of columns of A is a $\leq_{\mathrm{T}}^{\mathrm{L}}$ reduction to $\mathrm{C}_{=}\mathrm{L}$.

(4) Since the system of linear equations $A\vec{\mathbf{x}} = \vec{\mathrm{b}}$ has a solution if and only if matrices A and $[A; \vec{\mathbf{b}}]$ have the same rank, it follows from Theorem 6.50(3) that this problem is in $\mathrm{L}^{\mathrm{C}_{=}\mathrm{L}}$.

It follows from Theorem 6.50(1) that given $A \in \mathbb{Z}^{n\times n}$, the problem of determining if $\det(A) = 0$ is logspace many-one complete for $\mathrm{C}_{=}\mathrm{L}$. This is equivalent to determining if the system of linear equations $AX = 0$ does not have any solution $X \in \mathbb{Z}^{n\times n}$, where 0 denotes the $n \times n$ matrix containing only zeros. As a result the problem of determining if a system of linear equations does not have any solution is logspace many-one hard for $\mathrm{C}_{=}\mathrm{L}$.

We now show that the problem of determining if a system of linear equations has a solution is logspace many-one reducible to its complement problem of determining if a system of linear equations does not have a solution. We claim that the system $A\vec{\mathbf{x}} = \vec{\mathbf{b}}$ is feasible if and only if there exists a $\vec{\mathbf{y}}$ such that $A^T\vec{\mathbf{y}} = 0$ and $\vec{\mathbf{b}}^T\vec{\mathbf{y}} = 1$. Let W be the subspace spanned by the columns of A. The system is feasible if and only if $\vec{\mathbf{b}} \in W$. From elementary linear algebra we know that $\vec{\mathbf{b}}$ can be uniquely written as $\vec{\mathbf{b}} = \vec{\mathrm{v}} + \vec{\mathbf{w}}$, where $\vec{\mathbf{v}}$ is perpendicular to W (i.e., $\vec{\mathbf{v}}^T A = 0$) and $\vec{\mathbf{w}} \in W$. If $\vec{\mathbf{v}} \neq 0$, then since $\vec{\mathbf{v}}^T\vec{\mathbf{w}} = 0$, we have $\vec{\mathbf{v}}^T\vec{\mathbf{b}} = \vec{\mathbf{v}}^T\vec{\mathbf{v}} > 0$, and we may let $\vec{\mathbf{y}} = \frac{1}{\vec{\mathbf{v}}^T\vec{\mathbf{v}}}\vec{\mathbf{v}}$. Thus if $A\vec{\mathbf{x}} = \vec{\mathbf{b}}$ is infeasible, then there exists $\vec{\mathbf{y}}$ such that $A^T\vec{\mathbf{y}} = \vec{\mathbf{0}}$ and $\vec{\mathbf{b}}^T\vec{\mathbf{y}} = 1$. Conversely, if such a $\vec{\mathbf{y}}$ exists then $A\vec{\mathbf{x}} = \vec{\mathbf{b}}$ is infeasible. This shows that the problem of determining if a system of linear equations has a solution is logspace many-one hard for $\mathrm{C}_{=}\mathrm{L}$.

□

COROLLARY 6.51. *Let $p \in \mathbb{N}$ such that p is a prime.*

(1) *Let $A \in \mathbb{Z}^{n\times n}$. Determining is $\det(A) \not\equiv 0 (\mathrm{mod}\ p)$ is logspace many-one complete for* $\mathrm{Mod}_p\mathrm{L}$.
(2) *Let $A \in \mathbb{Z}^{n\times n}$. Computing the $rank(A)$ over $\mathbb{Z}_p$ is in* $\mathrm{Mod}_p\mathrm{L}$.
(3) *Given a set of vectors $S = \{\vec{\mathbf{v}}_1, \ldots, \vec{\mathbf{v}}_k\}$ with integer entries over the field $\mathbb{Z}_p$, determining if a column in S is in the lexicographically least set of linearly independent vectors in S over $\mathbb{Z}_p$ is in* $\mathrm{Mod}_p\mathrm{L}$.
(4) *Given $(A, \vec{\mathbf{b}})$, where $A \in \mathbb{Z}^{m\times n}$ and $\vec{\mathbf{b}} \in \mathbb{Z}^{m\times 1}$, determining if there exists $x \in \mathbb{Q}^{n\times 1}$ such that $A\vec{\mathrm{x}} = \vec{\mathbf{b}} (\mathrm{mod}\ p)$ is logspace many-one complete for* $\mathrm{Mod}_p\mathrm{L}$.

It is unknown if GapL is closed under division. However it is possible to use Theorem 6.50(3) and show the following result concerning division of two functions in GapL and on $rank(A)$, where $A \in \mathbb{Z}^{n\times n}$.

PROPOSITION 6.52. *Let $A \in \mathbb{Z}^{n\times n}$. Then there exists functions $g, h \in \mathrm{GapL}$ such that $rank(A) = \frac{g(A)}{h(A)}$.*

PROOF. It follows from Theorem 6.50(3) that determining if $rank(A) = r$ is in $\mathrm{C}_{=}\mathrm{L}$. As a result there exists a function $f \in \mathrm{GapL}$ such that

$$rank(A) = r \iff f(A, r) = 0.$$

Define functions

$$\begin{aligned} g(A) &= \sum_{r=0}^{n} r f(A, r), \\ h(A) &= \sum_{r=0}^{n} f(A, r). \end{aligned}$$

Then we have $g, h \in \mathrm{GapL}$ and $rank(A) = \frac{g(A)}{h(A)}$. □

6.3.1. Matrix problems reducible to computing the determinant. Let us consider the following problem.

POWERELEMENT
INPUT: $A \in \mathbb{R}^{n \times n}$, and $i, j, m \in \mathbb{N}$ such that $1 \leq i, j, m \leq n$.
OUTPUT: $(A^m)_{i,j}$, the $(i, j)^{th}$ entry of A^m.

PROPOSITION 6.53. POWERELEMENT$\leq_m^{\mathrm{L}}$Determinant.

PROOF. Given an input instance (A, i, j, m) of POWERELEMENT, we can without loss of generality assume that $i = 1$ and $j = n$ since it is possible to use elementary row and column transformations to obtain a $n \times n$ matrix M from the matrix A such that the $(i, j)^{th}$ element of A is the $(1, n)^{th}$ element of M, and $M^m_{1,n} = A^m_{i,j}$. So let $i = 1$ and $j = n$.

We now construct a matrix B such that $A^m_{i,j} = \det(B)$. We interpret A as representing a directed bipartite graph on $2n$ nodes. That is, the nodes of the bipartite graph are arranged in two columns of n nodes each. In both columns, nodes are numbered from 1 to n. If entry $a_{k,l}$ of A is not zero, then there is an edge labeled $a_{k,l}$ from node k in the first column to node l in the second column. Now, take m copies of this graph, put them in a sequence and identify each second column of nodes with the first column of the next graph in the sequence. Call the resulting graph as G'. The directed graph G' has $m + 1$ columns of nodes. The weight of a path in G' is the product of all labels on the edges of the path. The crucial observation now is that the entry at position $(1, n)$ in A^m is the sum of the weights of all paths in G' from node 1 in the first column to node n in the last column. Call these two nodes as s and t respectively.

The graph G' is further modified: for each edge (k, l) with label $a_{k,l}$, introduce a new node u and replace the edge by two edges (k, u) with label 1 and (u, l) with label $a_{k,l}$. Now all paths from s to t have even length, but still the same weight. Add an edge labeled 1 from t to s. Finally, add self-loops labeled 1 to all nodes, except t. Call the resulting graph G.

Let B be the adjacency matrix of G. The determinant of B can be expressed as the sum over all weighted cycle covers of G. However, every cycle cover of

G consists of a directed path from s to t, (due to the extra edge from t to s) and self-loops for the remaining nodes. The single nontrivial cycle in each cover has odd length, and thus corresponds to an even permutation. Therefore, $\det(B)$ is precisely the sum over all weighted directed paths from s to t in G'. We conclude that $\det(B) = (A^m)_{1,n}$ as desired. □

Let us consider the following problem.

ITMATPROD
INPUT: a set of n matrices $A_1, \ldots, A_n$ of dimension $n \times n$ each, indexes $1 \leq i, j \leq n$.
OUTPUT: the $(i,j)^{th}$ element of the product $\prod_{1 \leq i \leq n} A_i$.

PROPOSITION 6.54. *We have the following.*
(1) POWERELEMENT$\leq_m^{\mathrm{L}}$ ITMATPROD*, and*
(2) ITMATPROD$\leq_m^{\mathrm{L}}$ POWERELEMENT.

In other words, POWERELEMENT *and* ITMATPROD *are logspace many-one equivalent, denoted by* POWERELEMENT$\equiv_m^{\mathrm{L}}$ ITMATPROD.

PROOF. It is obvious that POWERELEMENT$\leq_m^{\mathrm{L}}$ITMATPROD. Conversely, let $A_1 \ldots, A_n$ be $n \times n$ matrices, and let B be the $(n^2+n) \times (n^2+n)$ matrix consisting of $n \times n$ blocks which are all zero except for $A_1, \ldots, A_n$ appearing above the diagonal of zero blocks. Then B^n has the product $A_1 A_2 \cdots A_n$ in the upper right corner. □

Let us consider the following problem.

MATINV
INPUT: $A \in \mathbb{R}^{n \times n}$ and $i, j \in \mathbb{N}$ such that $1 \leq i, j \leq n$.
OUTPUT: the $(i,j)^{th}$ element of A^{-1} in the form $(numerator, denominator)$ which is $((-1)^{i+j} \det(A(j|i)), \det(A))$, where $(-1)^{i+j} \det(A(j|i))$ denotes the $(i,j)^{th}$ entry in the co-factor matrix of A.

PROPOSITION 6.55. POWERELEMENT$\leq_m^{\mathrm{L}}$MATINV

PROOF. Let N be the $n^2 \times n^2$ matrix consisting of $n \times n$ blocks which are all zero except for $n-1$ copies of A above the diagonal of zero blocks. Then $N^n = 0$ and $(I_n - N)^{-1} = I_n + N + N^2 + \cdots N^{n-1} =$

$$\begin{bmatrix} I_n & A & A^2 & \cdots & A^{n-1} \\ 0 & I_n & A & \cdots & A^{n-2} \\ \vdots & & & & \vdots \\ 0 & & & \cdots & I_n \end{bmatrix},$$

where I_n denotes the $n \times n$ identity matrix. □

PROPOSITION 6.56. MATINV$\leq_m^{\mathrm{L}}$Determinant

PROOF. All the entries of the co-factor matrix of the input matrix A are determinants of minors of A with appropriate sign. □

THEOREM 6.57. POWERELEMENT, ITMATPROD, MATINV $\in$ GapL.

THEOREM 6.58. POWERELEMENT, ITMATPROD, MATINV *are logspace many-one hard for* ♯L.

6.4. Logarithmic space bounded counting classes and Boolean circuits

We recall Definitions 1.27 and 1.28 from Chapter 1. It can be shown using Theorem 6.46 that computing the determinant of an integer matrix is in the circuit-based complexity class $\mathrm{U_L}$-TC^1 as follows. Observing the proof of Theorem 6.46, it follows that given a matrix $A \in \mathbb{Z}^{n\times n}$, the graph H_A can be output in $\mathrm{U_L}$-AC^0. To complete the proof of the above assertion, we show that POWERELEMENT$\in$ $\mathrm{U_L}$-TC^1 first.

We need the following standard and basic functions involving integers:

ADD
INPUT: two n bitnumbers $a = a_{n-1}\cdots a_0$ and $b = b_{n-1}\cdots b_0$.
OUTPUT: $s = s_n \cdots s_0$, where $s =_{\mathrm{def}} a + b$.

ITADD
INPUT: n numbers in binary with n bits each.
OUTPUT: the sum of the input numbers in binary.

MULT
INPUT: two n bitnumbers $a = a_{n-1}\cdots a_0$ and $b = b_{n-1}\cdots b_0$.
OUTPUT: the product of the input numbers in binary.

ITMULT
INPUT: n numbers in binary with n bits each.
OUTPUT: the product of the input numbers in binary.

We can without loss of generality assume that the functions that we have defined are length respecting, which means the following. Let $f : \{0,1\}^* \to \{0,1\}^*$. We say that f is length-respecting if whenever $|x| = |y|$ then also $|f(x)| = |f(y)|$. Also corresponding to any length-respecting function $f : \{0,1\}^* \to \{0,1\}^*$, the notion of computing the bits of the output of the function f on any given input in $\{0,1\}^*$ is defined as follows.

DEFINITION 6.59. Let $f : \{0,1\}^* \to \{0,1\}^*$, $f = (f^n)_{n\in\mathbb{N}}$, be length-respecting. Let for every n and $|x| = n$ the length of $f^n(x)$ be $r(n)$. Let $f_i^n : \{0,1\}^n \to \{0,1\}$ be the Boolean function that computes the i^{th} bit of f^n, i.e., if $f^n(x) = a_1a_2\cdots a_{r(n)}$ then $f_i^n(x) = a_i$, where $1 \le i \le r(n)$. Then, $\mathrm{bits}(f)$ denotes the class of all those functions, i.e., $\mathrm{bits}(f) =_{\mathrm{def}} \{f_i^n | n, i \in \mathbb{N}, i \le r(n)\}$.

Let FSIZE-DEPTH$_{\mathcal{B}_1\cup\mathrm{bits(g)}}$ $(n^{O(1)}, 1)$ denote the complexity class of all length-respecting functions $f : \{0,1\}^* \to \{0,1\}^*$ such that computing $\mathrm{bits}(f)$ is in the complexity class SIZE-DEPTH$_{\mathcal{B}_1\cup\mathrm{bits(f)}}(n^{O(1)}, 1)$. We also need the definition of *constant-depth reductions*.

DEFINITION 6.60. Let $f, g : \{0,1\}^* \to \{0,1\}^*$ be length-respecting. We say that f is constant-depth reducible to g if $f \in$ FSIZE-DEPTH$_{\mathcal{B}_1 \cup \mathrm{bits(g)}}(n^{O(1)}, 1)$ and it is denoted by $f \leq_{\mathrm{cd}} g$. Here, a gate v for a function in $\mathrm{bits(g)}$ contributes to the size of its circuit with k, where k is the fan-in of v. We say that f is constant-depth equivalent to g and we write $f \equiv_{\mathrm{cd}} g$ if $f \leq_{\mathrm{cd}} g$ and $g \leq_{\mathrm{cd}} f$.

It is a standard result in circuit complexity theory that MAJ$\equiv_{\mathrm{cd}}$ ITADD$\equiv_{\mathrm{cd}}$ MULT$\equiv_{\mathrm{cd}}$ ITMULT. As a result MULT, ITADD$\in \mathrm{U_L}$-TC^0. Therefore it follows that given $\vec{\mathbf{u}}, \vec{\mathbf{v}} \in \mathbb{Z}^{n \times n}$, the inner product of $\vec{\mathbf{u}}$ and $\vec{\mathbf{v}}$, denoted by $\langle \vec{\mathbf{u}}, \vec{\mathbf{v}} \rangle$ is computable in $\mathrm{U_L}$-TC^0. As a result it is clear that it is possible to compute the $(i,j)^{th}$ entry in the product of an input pair of square integer matrices in $\mathrm{U_L}$-TC^0. As a consequence if we are given a set of n square integer matrices as input where every entry in each of these matrices is of size n then we can compute the entries in product of these matrices by combining these TC^0 circuits in a pairwise manner to obtain a Boolean circuit that contains $\neg$ gates, unbounded fan-in $\vee$,$\wedge$ and MAJ gates and whose size is a polynomial in n and depth $O(\log n)$. Clearly this shows that the problem of computing the $(i,j)^{th}$ entry of the powers of an input matrix that has entries in $\mathbb{Z}$ is in $\mathrm{U_L}$-TC^1 which implies POWERELEMENT$\in \mathrm{U_L}$-TC^1. It is also a standard fact that $\mathrm{U_L}$-AC^0 $\subseteq \mathrm{U_L}$-TC^1. As a result, combining the $\mathrm{U_L}$-AC^0 circuit that many-one reduces the problem of computing the determinant of an integer matrix to computing the difference of the $(i,j)^{th}$ entries of powers of the adjacency matrix of the layered directed acyclic graph H_A, it follows that we can compute the $\det(A)$ in $\subseteq \mathrm{U_L}$-TC^1. We therefore get the following theorem.

THEOREM 6.61. *Let $A \in \mathbb{Z}^{n \times n}$. Computing $\det(A) \in \mathrm{U_L}$-TC^1.*

This result also shows the following.

COROLLARY 6.62. $\mathrm{GapL} \subseteq \mathrm{U_L}$-TC^1.

COROLLARY 6.63. $\sharp$LH $\subseteq \mathrm{U_L}$-TC^1.

PROOF. It follows from Corollary 6.62 that $\sharp\mathrm{L} \subseteq \mathrm{U_L}$-TC^1. We therefore get $\sharp\mathrm{LH}_1 = \sharp\mathrm{L} \subseteq \mathrm{U_L}$-TC^1. We can now prove our result by showing that for $i \geq 2$, $\sharp\mathrm{LH}_i \subseteq \mathrm{U_L}$-TC^1 using induction on the number of levels of $\sharp$LH. □

Exercises

(1) Show that computing the permanent of a square integer matrix modulo 2 is logspace many-one complete for $\oplus$L.
(2) Show that determining if any two linear representations M_1 and M_2 of linearly representable matroids over $\mathbb{Z}_2$ represent the same matroid is logspace many-one complete for $\oplus$L.
(3) Let $A \in \mathbb{Q}^{n \times m}$ be the input rational matrix. Show that computing a maximal set of linearly independent columns of A over $\mathbb{Q}$ is in $\mathrm{FL}^{\mathrm{C_=L}}$.
(4) Let $A \in \mathbb{Q}^{n \times m}$ be the input rational matrix. Show that determining whether a column of A is in the lexicographically least set of linearly independent columns of A over $\mathbb{Q}$ is in $\mathrm{L}^{\mathrm{C_=L}}$.

(5) Define the GapL hierarchy, denoted by GapLH. Show that GapLH $=$ $\sharp$LH. Also show that GapLH $= \mathrm{U_L}$-$\mathrm{AC}^0(\mathrm{GapL})$.
(6) Let $A \in \mathbb{Q}^{m \times n}$ and $\mathbf{b} \in \mathbb{Q}^n$. Assume that the system of linear equations $A\mathbf{x} = \mathbf{b}$ has a solution. Show that it is possible to compute a solution $\mathbf{x}$ to $A\mathbf{x} = \mathbf{b}$ in GapLH_3. More precisely, show that a logspace machine with access to a function in GapLH_2 as an oracle can obtain a solution $\mathbf{x}$ to $A\mathbf{x} = \mathbf{b}$.
(7) Let $f(x), g(x) \in \mathbb{Q}[x]$ be monic polynomials given as input in terms of a vector of its coefficients. Show that the $\deg(\gcd(f(x), g(x)))$ can be computed in PL.
(8) Let $f(x), g(x), h(x) \in \mathbb{Q}[x]$ be monic polynomials given as input in terms of a vector of its coefficients. Show that we can test if $h(x) = \gcd(f(x), g(x))$ in $\mathrm{L}^{\mathrm{C_=L}}$.
(9) Let $f(x), g(x) \in \mathbb{Q}[x]$ be monic polynomials given as input in terms of a vector of its coefficients. Show that the coefficients of the $\gcd(f(x), g(x))$ can be computed in GapLH_5. More precisely, show that a logspace machine with access to a function in GapLH_4 as an oracle can compute the coefficients of the $\gcd(f(x), g(x))$.
(10) Let $A \in \mathbb{Q}^{n \times m}$ be the input rational matrix. Show that it is possible to compute the coefficients of the minimal polynomial of A in GapLH. What is the precise complexity level of GapLH to compute the coefficients of the minimal polynomial of A?

Notes

Basic facts about permutations included in Section 6.1 of this chapter is based on [**Her75**, 2.10]. We refer to [**DM96, HJ13, Str06**] for supplementary material on permutations and matrices.

The notation and terminology for all the definitions and results shown in Section 6.2 are based on the exposition and results due to Meena Mahajan and V. Vinay in [**MV97**]. The necessary background on Combinatorial Matrix Theory needed to follow results in Section 6.2 can be found in [**BR91**, Chapter 9]. We have brought about significant modifications to the definition of cycle cover, clow, clow sequence, weight of a clow, weight of a clow sequence and the sign of a clow sequence in [**MV97**]. Our proof of Theorem 6.44 is based on [**MV97**]. Theorem 6.43 has not been stated explicitly in [**MV97**] even though it is very useful to complete the proof of Theorem 6.45. Theorem 6.45 is stated in [**MV97**]. Our proof of Theorem 6.45 is substantially different from [**MV97**] in view of the use of Proposition 6.25. The 4-tuple used to represent a vertex in the definition of the special weighted layered directed acyclic graph $\mathbf{H}$ and the definition of $\mathbf{H}$ is slightly different from the vertex definition in [**MV97**]. Once again Theorems 6.46 and 6.47, Corollaries 6.48 and 6.49 are not stated in [**MV97**] explicitly even though the proof of these results have been explained.

Results shown in Section 6.3 on applications of computing the determinant are based on problems defined and studied in [**ABO99, BDH$^+$92, HT05, Coo85**].

The results on linear algebraic problems shown in this monograph is not exhaustive. Many more results on classifying the complexity of computing the coefficients of the minimal polynomial of an integer matrix [**HT03**], computing the inertia of an integer matrix [**HT05, HT10**], computing the gcd of the coefficients of any two uni-variate polynomials with integer coefficients [**Vij08, HT10, AV11**] have been shown to be contained in logarithmic space bounded counting classes, especially the $\sharp$LH.

We show in Section 6.4 that essentially all the logarithmic space bounded counting classes are contained in the Boolean circuit complexity class TC^1. The notion of length-respecting functions is from [**Vol99**, pp. 11]. The definition of computing the bits of a function $f : \{0,1\}^* \to \{0,1\}^*$ is from [**Vol99**, pp. 18]. The standard and basic functions defined in this section involving integers such as ADD, ITADD, MULT and ITMULT have been shown to be equivalent under constant-depth reductions in [**Vol99**, Corollary 1.45]. Exercise problems 3, 4 and 6 to 10 of this chapter are used in [**AV11**] to tightly classify the complexity of a linear algebraic problem called the orbit problem using logarithmic space bounded counting classes.

Yet another very interesting problem in computational complexity is the problem of testing if two given undirected graphs are isomorphic. In other words, this problem known as the Graph Isomorphism problem, is about determining if there exists a bijection ϕ between the set of vertices of a pair of undirected graphs (G_1, G_2) given as input such that given any two vertices $v_1, v_2 \in V(G_1)$ we have $(v_1, v_2) \in E(G_!)$ if and only if $(\phi(v_1), \phi(v_2)) \in E(G_2)$. There are many results known that classify the computational complexity of the Graph Isomorphism problem and its restricted variants using logarithmic space bounded counting classes. To mention a few references that contain these results, see [**JKM$^+$03, Tor04, AKV05, JKM$^+$06, Wag07, Tor08**].

APPENDIX A

Mathematical prerequisites

A.1. Number Theory

THEOREM A.1. **(Fundamental Theorem of Arithmetic)** *Every integer $n > 1$ can be uniquely represented as a product of prime powers.*

DEFINITION A.2. Given integers $a_1, a_2, \ldots, a_n$ all different from 0, the least of the positive common multiples is called the least common multiple, and it is denoted by $[a_1, a_2, \ldots, a_n]$.

DEFINITION A.3. Let α be any real number, and let k be a non-negative integer. Then the binomial coefficient $\binom{\alpha}{k}$ is given by the formula

$$\binom{\alpha}{k} = \frac{\alpha(\alpha-1)\cdots(\alpha-k+1)}{k!}.$$

Suppose that n and k are both integers. From the formula we see that if $0 \leq k \leq n$ then $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, whereas if $0 \leq n < k$, then $\binom{n}{k} = 0$. Here we employ the convention that $0! = 1$.

THEOREM A.4. *Let $\mathcal{S}$ be a set containing exactly n elements. For any non-negative integer k, the number of subsets of $\mathcal{S}$ containing precisely k elements is $\binom{n}{k}$.*

THEOREM A.5. *Let f denote a polynomial with integral coefficients. If $a \equiv b (\mathrm{mod}\ m)$ then $f(a) \equiv f(b) (\mathrm{mod}\ m)$*

THEOREM A.6. *Let $m_1, m_2, \ldots, m_r$ be non-zero integers. $x \equiv y (\mathrm{mod}\ m_i)$ for $i = 1, 2, \ldots, r$ if and only if $x \equiv y (\mathrm{mod}\ [m_1, \ldots, m_r])$.*

THEOREM A.7. *If $b \equiv c (\mathrm{mod}\ m)$ then $\gcd(b, m) = \gcd(c, m)$*

THEOREM A.8. **(Fermat's Little Theorem)** *Let p denote a prime. If p does not divide a then $a^{p-1} \equiv 1 (\mathrm{mod}\ p)$. For every integer a, $a^p \equiv a (\mathrm{mod}\ p)$.*

THEOREM A.9. **(Euler-Fermat Theorem)** *If $\gcd(a, m) = 1$, then*

$$a^{\phi(m)} \equiv 1 (\mathrm{mod}\ m),$$

where $\phi(m)$ denotes the number of positive integers less than or equal to m that are relatively prime to m.

$\mathbb{Z}$ denotes the set of integers. $\mathbb{Z}^+$ denotes the set of non-negative integers. $\mathbb{N}$ denotes the set of natural numbers. $\mathbb{Q}$ denotes the set of rationals. $\mathbb{R}$ denotes the set of real numbers. $\mathbb{C}$ denotes the set of complex numbers.

A.2. Asymptotic notation

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ read "the ceiling of x"). Let $f, g : \mathbb{N} \to \mathbb{Z}^+$.

(1) $\Theta(g(n)) = \{f(n)$: there exists positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$.
(2) $O(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.
(3) $\Omega(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.

A.3. Basics of Algebra & notation

- Order of a group G is the cardinality of G, and it is denoted by $o(G)$.
- Let G be a finite group and let H be a subgroup of G. The index of H in G, denoted by $[G : H]$, is the number of distinct left (or right) cosets of H in G. In particular, the index of H in G is,

$$[G : H] = \frac{o(G)}{o(H)}.$$

- Let ϕ be a homomorphism of G onto $\overline{G}$ with kernel K. Then, K is a normal subgroup of G, and G/K and $\overline{G}$ are isomorphic and it is denoted by $\left(\frac{G}{K}\right) \approx \overline{G}$.
- If G is a finite group and N is a normal subgroup of G, then $[G : N]$ denotes the order of the quotient group G/N.
- For $n \in \mathbb{Z}$ and $n \geq 1$, $(\mathbb{Z}_n, +_n)$ denotes the finite additive abelian group of integers modulo n.
- For $n \in \mathbb{Z}$ and $n \geq 1$, $(\mathbb{Z}_n^*, .n)$ denotes the finite multiplicative abelian group of integers modulo n. Elements of this group are elements of $\mathbb{Z}_n$ that are relatively prime to n.
- If r is a prime, then $(\mathbb{Z}_r, +_r, .r)$ is a finite field under addition and multiplication of integers modulo r.
- $\mathbb{Q}$ is a field under addition and multiplication of rationals and $\mathbb{R}$ is a field under addition and multiplication of reals.
- For $n \geq 1$, $\mathbb{Q}^n$ denotes the set of all column vectors with n components, each containing a rational. We can add two vectors in $\mathbb{Q}^n$ and multiply a vector in $\mathbb{Q}^n$ by a scalar from $\mathbb{Q}$. In other words, we can take linear combinations of vectors in $\mathbb{Q}^n$ with coefficients from $\mathbb{Q}$.
- A set of columns of a vector space V over a field $\mathbb{F}$ is linearly dependent if there exists a linear combination of vectors with non-zero coefficients in $\mathbb{F}$ which is equal to the zero vector. Otherwise the set of columns is linearly independent.
- A set of vectors B, in a vector space V over a field $\mathbb{F}$, spans V if every vector in V can be expressed as a linear combination of vectors in B with coefficients in $\mathbb{F}$.

- A basis for a vector space V over a field $\mathbb{F}$ is a set of vectors B such that vectors in B are linearly independent, and B spans V.
- Let $A \in \mathbb{R}^{n\times n}$. We define the *permanent* of A as:

$$\text{perm}(A) \quad = \quad \sum_{\sigma\in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}. \tag{A.1}$$

- Let $A \in \mathbb{R}^{n\times n}$. We define the *determinant* of A as:

$$\det(A) \quad = \quad \sum_{\sigma\in S_n} sgn(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}. \tag{A.2}$$

- Let $A \in \mathbb{R}^{n\times n}$. The *characteristic polynomial* of A is defined as $\det(\lambda I_n - A)$, where λ is a variable and I_n is the $n \times n$ identity matrix.
- The Cayley-Hamilton Theorem states that *every matrix A satisfies its own characteristic equation*. In other words, if $\chi(x)$ denotes the characteristic polynomial of A, then $\chi(A) = 0$.
- Let $A \in \mathbb{R}^{n\times n}$. Roots of the characteristic polynomial of A are called as *eigenvalues* of A. Clearly, the roots of A can be real or complex numbers.
- Let $A \in \mathbb{R}^{n\times n}$. $rank(A)$ is defined as the maximum number of linearly independent columns of A over $\mathbb{R}$.
- Let $A \in \mathbb{R}^{n\times n}$. The *minimal polynomial* of A is the unique monic polynomial of the smallest degree satisfied by A.
- Let $A \in \mathbb{R}^{n\times n}$. The minimal polynomial of A is a factor of the characteristic polynomial of A.
- Let $A \in \mathbb{R}^{n\times n}$. Inertia of A, denoted by $i(A)$, is defined as the triple $i(A) = (i_+(A), i_-(A), i_0(A))$, where $i_+(A), i_-(A)$ and $i_0(A)$ are the number of eigenvalues of A, counting multiplicities, with positive, negative and zero real part.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[ABC+09] Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and Grid Graph Reachability Problems. *Theory of Computing Systems*, 45: 675-723, 2009.

[ABO99] Eric Allender, Robert Beals and Mitsunori Ogihara. The complexity of matix rank and feasible systems of linear equations. *Computational Complexity*, 8:99–126, Birkhauser,1999.

[AHT07] Manindra Agrawal, Thanh Minh Hoang and Thomas Thierauf. The Polynomially Bounded Perfect Matching Problem Is in NC^2. In *STACS '07: Proceedings of the* 24^{th} *Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 4393, pp. 489-499, Springer, 2007.

[AJ93] Carme Àlvarez and Birgit Jenner. A very hard log-space counting class. *Theoretical Computer Science*, 107(1):3-30, Elsevier, 1993.

[AKV05] Vikraman Arvind, Piyush P Kurur, T.C. Vijayaraghavan. Bounded Color Multiplicity Graph Isomorphism is in the $\sharp$L Hierarchy.. In *CCC '05: Proceedings of the* 20^{th} *Annual IEEE Conference on Computational Complexity*, pp. 13-27, IEEE Computer Society, 2005.

[All04] Eric Allender. Arithmetic Circuits and Counting Complexity Classes. *Complexity of Computations and Proofs*, ed. Jan Krajiček, Quaderni di Matematica Vol. 13, Seconda Universita di Napoli, pp. 33-72, 2004.

[AO96] Eric Allender and Mitsunori Ogihara. Relationships among PL, #L and the Determinant. *RAIRO-Theoretical Informatics and Applications*, 30:1–21, 1996.

[AV10] V. Arvind and T. C. Vijayaraghavan. Classifying Problems on Linear Congruences and Abelian Permutation Groups using Logspace Counting Classes, *Computational Complexity*, 19(1):57-98, Birkhauser, 2010.

[AV11] V. Arvind and T. C. Vijayaraghavan. The Orbit Problem is in the GapL hierarchy, *Journal of Combinatorial Optimization*, 21:124-137, 2011.

[BCH86] Paul W. Beame, Stephen A. Cook and James Hoover. Log depth circuits for division and related problems, *SIAM Journal on Computing*, 15(4):994-1003, 1986.

[BDG95] Jose Luis Balcazar, Josep Diaz and Joaquim Gabarro. *Structural Complexity I, Second Edition*. Springer, 1995.

[BDH+92] Gerhard Buntrock, Carsten Damm, Ulrich Hertrampf and Christoph Meinel. Structure and Importance of Logspace-MOD Classes. *Mathematical Systems Theory*, 25(3):223-237, Springer-Verlag, 1992.

[BG92] Richard Beigel and John Gill. Counting classes: Thresholds, parity, mods and fewness. *Theoretical Computer Science*, 103(1):3-23, 1992.

[BJL+91] Gerhard Buntrock, Birgit Jenner, Klaus-Jörn Lange and Peter Rossmanith. Unambiguity and fewness for logarithmic space. In *FCT '91: Proceedings of the* 8^{th} *International Conference on Fundamentals of Computation Theory*, LNCS 529, pp. 168-179, Springer, 1991.

[Bor77] A. Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6: 733-744, 1977.

[BR91] Richard A. Brualdi and Herbert J. Ryser. *Combinatorial Matrix Theory*. Encyclopedia of Mathematics and its Applications, Cambridge University Press, 1991.

[BRS95] Richard Beigel, Nick Reingold and Daniel Spielman. PP is Closed under Intersection, *Journal of Computer and System Sciences*, 50(2): 191-202, 1995.

[BTV09] Chris Bourke, Raghunath Tewari and N. V. Vinodchandran. Directed Planar Reachability Is in Unambiguous Log-Space, *ACM Transactions on Computation Theory*, Vol. 1, No. 1, Article 4, 2009.

[CDL01] Andrew Chiu, George Davida and Bruce Litow. Division in logspace-uniform NC^1. *RAIRO-Theoretical Informatics and Applications*, 35(3):259-276, 2001.

[CLR+22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms, Fourth edition*. MIT Press, 2022.

[Coo85] Stephen A. Cook. A Taxonomy of Problems with Fast Parallel Algorithms. *Information and Control*, 64(1-3):2-21, 1985.

[DK14] Ding-Zhu Du and Ker-I Ko. *Theory of Computational Complexity, Second Edition*. John Wiley & Sons, 2014.

[DM96] John D. Dixon and Brian Mortimer. *Permutation Groups*. Graduate Texts in Mathematics 163, 1996.

[For97] Lance Fortnow. Counting Complexity, In *Complexity Theory Retrospective II*, editors Lane A. Hemaspaandra and Alan Selman, pp. 81-107, Springer, 1997.

[Gil77] John Gill. Computational Complexity of Probabilistic Turing Machines. *SIAM Journal on Computing*, 6(4):675-695, 1977.

[HAB02] William Hesse, Eric Allender and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695-716, 2002.

[Her75] I. N. Herstein. *Topics in Algebra* Wiley Eastern Limited, 1975, Twelfth Wiley Eastern Reprint, 1992.

[HJ13] Roger A. Horn and Charles R. Johnson. *Matrix Analysis, Second Edition, South Asia Edition*, Cambridge University Press, 2013.

[HO01] Lane A. Hemaspaandra and Mitsunori Ogihara. *The Complexity Theory Companion*, Springer, 2001.

[HRV00] Ulrich Hertrampf, Steffen Reith and Heribert Vollmer. A note on closure properties of logspace MOD classes. *Information Processing Letters*, 75(3):91-93, 2000.

[HT03] Thanh Minh Hoang and Thomas Thierauf. The complexity of the characteristic and the minimal polynomial. *Theoretical Computer Science*, 295(1-3):205-222, 2003.

[HT05] Thanh Minh Hoang and Thomas Thierauf. The Complexity of the Inertia and Some Closure Properties of GapL. In *CCC '05: Proceedings of the 20th Annual IEEE Conference on Computational Complexity*, pp. 28-37, 2005.

[HT10] Thanh Minh Hoang and Thomas Thierauf. The complexity of the inertia. *Computational Compelxity*, 19(4):559-580, 2010.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*, Narosa Publishing House, 1979, Seventeenth Reprint, 1997.

[Imm99] Neil Immerman. *Descriptive Complexity*, Graduate Texts in Computer Science, Springer 1999.

[JKM+03] Birgit Jenner, Johannes Kobler, Pierre McKenzie, and Jacobo Toran. Completeness results for graph isomorphism. *Journal of Computer and System Sciences*, 66: 549-566, 2003.

[JKM+06] Birgit Jenner, Johannes Kobler, Pierre McKenzie, and Jacobo Toran. Corrigendum to completeness results for graph isomorphism. *Journal of Computer and System Sciences*, 72: 783, 2006.

[Jun85] Hermann Jung. On probabilistic time and space. In *ICALP '85: Proceedings of the International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science 194, pp. 310-317, Springer, 1985.

[KT96] Johannes Köbler and Seinosuke Toda. On the power of generalized MOD-classes. *Mathematical Systems Theory*, 29(1):33-46, Springer-Verlag, 1996.

[Koz97] Dexter C. Kozen. *Automata and Computability*, Undergraduate Texts in Computer Science, Springer, 1997.

[LP98] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation, Second Edition*, Prentice-Hall, 1998.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*, Cambridge University Press, 1995.

[MV97] Meena Mahajan and V. Vinay. Determinant: Combinatorics, Algorithms, and Complexity. *Chicago Journal of Theoretical Computer Science*, 1997.

[Ogi98] Mitsunori Ogihara. The PL hierarchy collapses. *SIAM Journal on Computing*, 27(5):1430-1437, 1998.

[PTV12] A. Pavan, Raghunath Tewari amd N. V. Vinodchandran On the power of unambiguity in log-space. *Computational Complexity*, 21(4):643-670, 2012.

[RA00] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM Journal on Computing*, 29(4):1118-1131, 2000.

[RST84] Walter L. Ruzzo, Janos Simon, and Martin Tompa. Space-Bounded Hierarchies and Probabilistic Computations. *Journal of Computer and System Sciences*, 28(2):216-230, 1984.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation, Third edition*, Cengage Learning, 2013, First Indian reprint, 2018.

[Str06] Gilbert Strang. *Linear Algebra and its Applications, Fourth edition*, Cengage Learning, 2006, Fifteenth Indian reprint, 2014.

[TM97] J. P. Tremblay and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*, Tata McGraw-Hill Publishing Company Limited, 1997.

[Tor04] Jacobo Toran. On the hardness of Graph Isomorphism., *SIAM Journal on Computing*, 33(5): 1093-1108, 2004.

[Tor08] Jacobo Toran. Reductions to Graph Isomorphism, In *FSTTCS '08: Proceedings of the* 27^{th} *International Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 4855, pp. 158-167, Springer, 2008.

[TV12] Raghunath Tewari and N. V. Vinodchandran. Green's theorem and isolation in planar graphs, *Information and Computation*, 215:1-7, Elsevier, 2012.

[Vij08] T. C. Vijayaraghavan. *Classifying certain algebraic problems using logspace counting classes*, Ph.D Thesis, The Institute of Mathematical Sciences, Homi Bhabha National Institute, 2008.

[Vij10] T. C. Vijayaraghavan. *A note on Closure Properties of* ModL, Electronic Colloquium on Computational Complexity, Report No. 99, 2010.

[Vij22] T. C. Vijayaraghavan. *A combinatorial property of* $\sharp$L *assuming* $\mathrm{NL} = \mathrm{UL}$ *and its implications for* ModL, International Virtual Conference on Advances in Data Sciences and Theory of Computing (ICADSTOC-2022), pp. 51-56, Bharath Institute of Higher Education and Research, 2022. Also published as Revision 1 of Electronic Colloquium on Computational Complexity Report No. 82, 2009.

[Vol99] Heribert Vollmer. *Introduction to Circuit Compelxity*, Springer, 1999.

[Wag07] Fabian Wagner. Hardness Results for Tournament Isomorphism and Automorphism, In *Proceedings of the* 32^{nd} *International Symposium on the Mathematical Foundations of Computer Science (MFCS)*, LNCS 4708, pp. 572-583, Springer, 2007.

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions*, John Wiley & Sons, 1987.

Author Index

Subject Index

T. C. Vijayaraghavan

The Complexity of Logarithmic Space Bounded Counting Classes (Second Edition)

A logarithmic space bounded counting class is defined based on the number of accepting and/or the number of rejecting computation paths of a NL-Turing machine. This monograph gives a nice in-depth exposition of logarithmic space bounded counting classes and many important results on them including their properties. A major portion of results shown by the author have not appeared in any textbook on computational complexity.

This textbook is intended to be an introductory material on logarithmic space bounded counting classes for advanced under-graduate students or graduate students and researchers who have prior knowledge of basics concepts in the Design and Analysis of Algorithms, Theory of Computation, Computational Complexity and Discrete Mathematics.

DeepScience

9 789371 853033