Unambiguous Functions in Logarithmic Space

Unambiguous Functions in Logarithmic Space

By
Grzegorz Herman, M.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Doctor of Philosophy

McMaster University

DOCTOR OF PHILOSOPHY (2009) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: Unambiguous Functions in Logarithmic Space

AUTHOR: Grzegorz Herman, M.Sc. (Jagiellonian University, Kraków, Poland)

SUPERVISOR: Michael Soltys, Ph.D.

NUMBER OF PAGES: v, 46

Abstract

The notion of nondeterminism is one of the most fundamental concepts in many areas of computer science. Unambiguity, requiring that there be at most one correct sequence of nondeterministic choices, has proved to be one of the most meaningful restrictions of nondeterminism. In the context of space-bounded Turing Machines, several variants of unambiguity have been proposed and studied, and some interesting results have been established, narrowing slightly the gap between deterministic and nondeterministic logarithmic-space computation.

We study the different variants of unambiguity in the context of computing multi-valued functions (as opposed to the usual yes/no decision problems). We propose a modification to the standard computation models of Turing Machines and configuration graphs, which allows for unambiguity-preserving composition. We introduce a unified notation, capturing the different flavors of ambiguity. Furthermore, we define a notion of reductions (based on function composition), which allows nondeterminism but controls its level of ambiguity. In the light of this framework we establish some reductions between different variants of path counting problems. We also investigate more carefully the technique of inductive counting, and obtain improvement of some existing results.

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Michael Soltys. It was much thanks to him—being not only an inspiring and patient advisor, but also a great person—that my years at McMaster were truly pleasant and enriching.

The other members of my supervisory committee: Dr. Ryszard Janicki and Dr. Emil Sekerinski, as well as Prof. Stephen Cook, who has agreed to review my thesis, have all provided many valuable remarks.

I am deeply grateful to my parents Izabela and Krzysztof, and to my grandmother Zofia—I think it is simply not possible to overestimate the support and encouragement I have received from them.

Many of my fiends also deserve my sincere thanks. In Poland: Lech, Ania and Michał, Marta, Patryk, Kasia, Ania, and of course Dominika, who has waited for me so patiently... In Canada, I have been blessed to meet father Peter, Allan, Theresa and Elaine, Lauren and Duc, Meghan, and many others. Thank you all for being there for me!

Contents

Chapter 1

Introduction

1.1 A Bit of History

The notion of nondeterminism is one of the fundamental notions in theoretical computer science. Given any model of computation which allows it, the question whether nondeterminism yields a proper increase in the power of the model, is one of the first to be asked. The relationship between deterministic and nondeterministic flavors of resource-bounded computation classes has been under intense scrutiny for the last four decades, giving rise to numerous formulations, techniques, and even new branches of computer science (e.g., proof complexity has emerged from trying to tackle the **P** vs. **NP** question). It seems however, that despite all the progress, we still do not have an adequate understanding of these issues. And so, even in the most natural and well studied cases—of polynomial time and logarithmic space—the questions of nondeterminism (i.e., **P** vs. **NP** and **L** vs. **NL**) stand open.

The study of nondeterminism in the space-bounded context has been more fruitful in unconditional results, than in the time-bounded context. In particular, very low space bounds ($o(\log\log(n))$) have been shown in [34, 12] to capture only regular languages, with nondeterminism yielding no additional power. For bounds at least polylogarithmic ($(\log(n)^{O(1)}$), an algorithm due to Savitch (see [33]) allows us to do away with nondeterminism. Between these two bounds the question remains open, though it has been shown by Kannan (see [16]) that the equality of $\mathbf{DSPACE}(\log\log(n))$ and $\mathbf{NSPACE}(\log\log(n))$ cannot be proven by means of simulation, and by Szepietowski (see [37]) that this equality would imply $\mathbf{L} = \mathbf{NL}$. Because sub-logarithmic space machines lack the ability to count, the world between the space bounds of $O(\log\log(n))$ and $o(\log(n))$ is very sensitive to minor modifications in the definitions and in general requires special proof techniques ([38] contains a thorough presentation of these issues; for a survey and some interesting results see

also [22]).

Among all space bounds, it is the logarithmic one that has received the most attention. It is large enough to allow counting (and thus many "natural" algorithmic approaches), while at the same time small enough (even in the nondeterministic variant) to be contained in polynomial time (in fact, even in $\mathbf{NC}^2$), which makes it well-fit for reductions between polynomial-time problems. Many things are known about nondeterministic logarithmic space. It is closed under complementation (see [13, 36]) and thus both the oracle and the alternation hierarchies collapse to **NL**. Its unbounded alternation class, called **AL**, is equal to **P** (see [9]). Finally, the symmetric variant of **NL** (defined in [21]) has been recently shown (by Reingold; see [28], building on results from [29]) to be equal to **L**. However, the "big question"—whether **L** equals **NL**—remains open.

When a notion resists a complete understanding, it is natural to restrict it in some way, hoping that the restricted case will be easier to analyze. Accordingly, numerous restrictions of nondeterminism have been introduced and studied. A notorious restriction is that of *unambiguity*, in which the machine does not need to know the path to an answer (and thus can make nondeterministic choices), but the path itself is required to be unique. Unambiguity has been first introduced for context-free languages (requiring that every word has at most one derivation). For polynomial time bounds, it has been defined by Valiant (see [39]). Although much has been said about the class **UP** (especially concerning its relation to one-way functions, one of the core concepts of cryptography), its exact relations to both **P** and **NP** remain unknown.

The unambiguous version of **NL**, called **UL**, has been first explicitly considered in [4] and [8]. In the latter, variants of **UL** that allow polynomially many accepting computation paths, as well as variants that consider not only accepting, but all reachable or all paths, have been proposed. Some inclusions between these classes have been presented, and the classes **ReachUL** and **StrongUL** have been shown to be closed under complementation. The Immerman-Szelepcsényi technique of inductive counting has been extended in [7], allowing the removal of ambiguity at the cost of a relatively small increase in required computation space. **StrongUL** has been shown by Allender and Lange to be contained in deterministic space $O(\frac{\log(n)^2}{\log\log(n)})$ (see [2]). In [20], Lange has exposed a problem complete for **ReachUL**. Finally, inductive counting has been used again by Reinhardt and Allender in [30] to show that **UL** and **NL** coincide in the nonuniform setting (i.e., **NL/poly=UL/poly**), and thus also in the uniform setting under some hardness assumptions (see [3] for details).

The study of space-bounded computation has also included numerous extensions of the model. Of these, the concept of an auxiliary pushdown automaton (a space-bounded, possibly nondeterministic Turing Machine, with an additional push-

down (last-in, first-out) storage, not subject to the space bound) has received a lot of attention. Cook has shown in [10] that logarithmic-space AuxPDAs (both deterministic and nondeterministic) capture deterministic polynomial-time exactly. When a polynomial-time restriction is added, deterministic and nondeterministic logarithmic-space AuxPDAs have been shown in [32] and [35] to capture languages that are log-space reducible to deterministic and general context-free languages, respectively. The concept of unambiguity has been investigated also in this setting, but here the equivalence of unambiguous AuxPDAs and unambiguous context-free languages has not been exposed so far (Niedermeier and Rossmanith have shown in [26] that the latter can be recognized within the class **StrongUAuxPDA**, but the converse inclusion remains open). The aforementioned result of Reinhardt and Allender proves that unambiguous AuxPDAs can recognize all context-free languages, but it requires the use of advice, and thus applies only in the nonuniform setting.

Altogether, collapses of deterministic and nondeterministic complexity classes, and equivalences of various extensions or restrictions of nondeterminism, have been shown in the context of bounded space—even though not all space bounds have been covered unconditionally. On the other hand, known separation results (such as those for sub-logarithmic alternation hierarchies; see [22]) work under "ill" conditions, such as inconstructibility of respective space bounds or the machine's inability to count. This provides a strong evidence toward the claim that nondeterminism does not increase the power of a space-bounded Turing Machine. However, it seems that the current techniques are insufficient to provide a proof of this claim, and applying ideas from other branches of mathematics (as an example one can take the Reingold's proof that $\mathbf{L} = \mathbf{SL}$, based on expander graphs and analysis of eigenvalues of graph adjacency matrices) might be necessary to resolve the question. The lesser claim of $\mathbf{USPACE}(f(n)) = \mathbf{NSPACE}(f(n))$ (and, in particular, $\mathbf{UL} = \mathbf{NL}$) should be in much closer range.

1.2 Contributions

In our study, we initially set out to solve the **UL** vs. **NL** problem. As providing any unconditional results in this field seems quite difficult, we decided to analyze the relative complexity (in terms of the required ambiguity) of problems. This has led to a search of a notion of reduction which would be at the same time stronger than deterministic log-space, and weak enough to provide meaningful comparison of problems and classes. Finally, opting for a variant of functional many-one reductions, we have introduced a generic framework for analysis of space-bounded, limited-ambiguity functions. The contributions of this thesis can thus be described as follows:

1. **Model:** We provide a modification of the standard model of oracle Turing Machines, which allows for nondeterministic computation of deterministically valued (i.e., well defined) functions. In the space-bounded setting, such computation has been defined (as in [14]) only based on *deterministic* machines having access to an oracle for a language from a possibly nondeterministic class. In a different manner, functions (including the path-counting functions which we consider in Chapter 3) have been defined based on properties of the *computation tree* of a nondeterministic machine (giving rise to classes such as **#L**, **GapL**, etc.; see [4]), and not *computed* by the machine itself. Our model agrees with the first of the above when full power of nondeterminism is allowed, but has the advantage of being easily adaptable to classes of limited ambiguity. The different variants of unambiguity are achieved through restricting the shape of the configuration graph of our machines. The computation is shown to compose nicely, preserving both space and ambiguity constraints.

2. **Reductions:** Based on our computation model, we introduce a notion of nondeterministic, unambiguous reductions. The notion of space-bounded reducibility has been extensively studied. Nondeterminism has been first added in this context to Turing reductions (see [18, 31]), this model has been also shown not to be robust with respect to minor definition changes. Also, as the bias of our work is toward showing collapses rather than separations, we tried to avoid creating oracle hierarchies (we use oracles only as a tool of function composition, not as sources of additional computation power). Nondeterministic many-one reductions have been introduced in [19]—there, however, a reduction effectively computes a relation (a set-valued function). Our notion is much better fit for the purpose of comparing the ambiguity-complexity of functions, it is also as natural to work with as many-one reductions.

3. **Counting:** Within our framework, we analyze variants of the path-counting problem. We exhibit some dependencies between different count ranges. In particular, by Propositions 3.2.7 and 3.2.10 we obtain the equivalence of counting up to any constant number of (arbitrary or simple) paths. Furthermore, we take a closer look on the inductive counting technique of [13, 36], which allows us to combine the results of [7] and [30] into Algorithm 3.3.12: an unambiguous algorithm for reachability on graphs with restricted ambiguity of shortest paths. In the process, some of the interplay between the ambiguity of a graph and the ambiguity of its traversal, is shown.

1.3 Structure and Notation

This thesis has the following structure: In Chapter 2, we discuss our formal models of computation. First, we define Quering Turing Machines and Quering Computation Graphs, together with their restricted-ambiguity variants. Based on these definitions, we introduce a consistent naming scheme for various ambiguity-related function and language classes. Next, we introduce our concept of unambiguous reductions, and show how they can be used to place functions in some of the classes under consideration. In Chapter 3, we look at specific problems related to path counting. We exhibit some relationships between counting up to different bounds. Finally, we examine the approaches to reachability based on inductive counting and graph traversal, which enables us to improve the results from [13, 36] and [7]. We conclude with a short discussion of possible future work directions. In Appendix A, we include an outline of a possible approach to the **UL** vs. **NL** question.

Throughout the thesis, we employ a consistent notation, denoting:

- natural numbers and indices by small letters i through n,
- complexity bounds (functions on natural numbers) by small letters f, g, and h,
- polynomials by small letters p and r,
- alphabets and arbitrary sets by capital Greek letters Σ, Γ, Δ,
- single characters (alphabet symbols) by small letters a through d,
- words (strings of characters) by capital letters W through Z,
- languages (sets of words) by small Greek letters α through δ,
- functions on words by small Greek letters ϕ, ψ, ξ and θ,
- machines and oracles by capital letters M, N, and O,
- machine states by small letter q,
- machine configurations by capital letter C,
- graphs by capital letters G, H, and I,
- sets of graph vertices and edges by capital letters V and E, respectively,
- graph vertices and edges by small letters s through z, and e, respectively,

- (arbitrary) complexity classes and graph classes in script letters (e.g., $\mathcal{C}$),
- specific complexity classes and problems in boldface (e.g, $\mathbf{QFunc}(\log(n))$, **Reach**),
- families of complexity classes in boldblank (e.g., $\mathbb{REM}$).

When an index (e.g., i) appears in a place where a string is expected, we assume the natural binary encoding is used. We will use ϵ to denote an empty string, juxtaposition (e.g., XY) for string concatenation, Kleene star (*) for repetition, and square brackets to access individual characters of a string (e.g., $X[i]$). We will use angle brackets to denote sequences, write $\langle \ldots \rangle_i$ for a sequence over possible i's, and S_i to denote the i-th component of a sequence S. Finally, in the text, we will use *italics* for emphasis and **boldface** for newly-defined terms.

Chapter 2

Models of Computation

We begin with providing a model of computation which will be employed throughout this work. Since the goal is the analysis of functions that can be computed unambiguously in logarithmic space, the following natural requirements emerge:

- the model must allow computing functions with an arbitrary range,
- sub-linear (and, in particular, logarithmic) space bounds must be enforceable,
- nondeterministic computation must be possible, but its level of ambiguity held under control,
- computations have to be composable, and the composition should obey the space and ambiguity restrictions as much as possible,
- the complexity classes based on the new model should coincide with the classical ones.

2.1 Quering Turing Machines

As we have outlined before, we need a model of computation that allows function composition, and makes the analysis of such complex functions straightforward. The usual model of Turing Machines does not behave well when composing (under sub-linear space bounds): the input and output tapes, not subject to the space bounds, become an internal tape of the composed machine, which should obey the space restrictions. To deal with that issue we employ its well-known modification: instead of producing a (possibly long) output in its entirety, the machine computes just one requested character. Moreover, we use the same approach to access the machine's input—it writes the index of the input character it is interested in on one of its tapes,

and queries an oracle[1] (by entering a special state). Therefore such a machine can be seen as one rewriting **requests**[2] about its **output** to (sequences of) **queries** about its **input**, and generating an **answer** based on the results of these queries. This model, which we will call a **Quering Turing Machine**, can be formally defined as follows[3]:

Definition 2.1.1 *A k-tape* **Quering Turing Machine** *is a tuple consisting of:*

- *a finite* **alphabet**[4] Σ *(*$|\Sigma| \geq 2$*),*
- *a finite set of* **states** Γ*,*
- *an* **initial state** $q_{init} \in \Gamma$*,*
- *an* **answer state** $q_a \in \Gamma$ *for every* $a \in \Sigma$*,*
- *a* **query state** $\hat{q}_{query} \in \Gamma$*,*
- *a* **response state** $\hat{q}_a \in \Gamma$ *for every* $a \in \Sigma$*,*
- *a* **transition relation** $\Delta \subseteq \Gamma \times \Sigma^k \times \Gamma \times \Sigma^k \times \{\leftarrow, -, \rightarrow\}^k$.

We will use the usual notion of configuration for Turing Machines:

Definition 2.1.2 *A* **configuration** *of a Quering Turing Machine consists of the current state* $q \in \Gamma$ *and, for every* $i \in \{1, \ldots, k\}$*, a pair of strings* $\langle X_i, Y_i \rangle \in \Sigma^* \times \Sigma^*$*, representing the contents of the i-th tape (to the left and to the right of the position of the head, respectively; the head can be seen as reading the first symbol of* Y_i*, or a blank if* $Y_i = \epsilon$*). The* **initial configuration** *on request i is* $\langle q_{init}, \langle \epsilon, i \rangle, \langle \epsilon, \epsilon \rangle, \ldots, \langle \epsilon, \epsilon \rangle \rangle$*: the request is written (in binary) on tape 1, all other tapes are empty.*

As a Quering Turing Machine accesses an oracle, there are two kinds of state changes: those intrinsic to the machine itself (as described by the transition relation Δ), and those "performed" by the oracle, which is not part of the machine. Consequently, we can define two relations formalizing these state changes:

[1] In some cases we will consider machines with more than one input oracle (thus computing functions of bigger arity). However such a tandem can always be seen as a single oracle whose queries specify the requested input next to the bit index, and thus all formalisms will consider single-input machines only.

[2] We will use the words: input, output, request, query, and answer, to mean precisely the roles described here.

[3] For a nice exposition of regular and oracle Turing Machines, see for example [27, 11].

[4] Usually, a Turing Machine has two alphabets: the input/output alphabet, and a larger tape alphabet. However, as in our model the input and output are never written anywhere, and any alphabet of size at least two can be easily encoded (with only a constant factor space cost) using any other, we have decided to unify them for simplicity.

Definition 2.1.3 *A configuration C* **intrinsically yields** *a configuration C' iff for some $q, q' \in \Gamma$, $X_1, \ldots, X_k, Y_1, \ldots, Y_k \in \Sigma^*$, $a_1, \ldots, a_k \in \Sigma \cup \{\epsilon\}$, $b_1 \ldots, b_k, c_1, \ldots, c_k \in \Sigma$, $m_1 \ldots, m_k \in \{\leftarrow, -, \rightarrow\}$, we have:*

$$C = \langle q, \langle X_1 a_1, b_1 Y_1 \rangle, \ldots, \langle X_k a_k, b_k Y_k \rangle \rangle,$$
$$C' = \langle q', \langle X_1 W_1, Z_1 Y_1 \rangle, \ldots, \langle X_k W_k, Z_k Y_k \rangle \rangle,$$
$$\Delta \ni \langle q, \langle b_1, \ldots, b_k \rangle, q', \langle c_1, \ldots, c_k \rangle, \langle m_1, \ldots, m_k \rangle \rangle,$$
$$\forall_{i \in \{1,\ldots,k\}}, \langle W_i, Z_i \rangle = \begin{cases} \langle \epsilon, a_i c_i \rangle, & \text{if} \quad m_i = \leftarrow \text{ and } a_i \neq \epsilon, \\ \langle a_i, c_i \rangle, & \text{if} \quad m_i = -, \\ \langle a_i c_i, \epsilon \rangle, & \text{if} \quad m_i = \rightarrow . \end{cases}$$

A configuration C **extrinsically yields** *a configuration C' under input oracle O iff for some $X_1, \ldots, X_k, Y_1, \ldots, Y_k \in \Sigma^*$, $a \in \Sigma$, we have:*

$$C = \langle \hat{q}_{query}, \langle X_1, Y_1 \rangle, \ldots, \langle X_k, Y_k \rangle \rangle,$$
$$C' = \langle \hat{q}_a, \langle X_1, Y_1 \rangle, \ldots, \langle X_k, Y_k \rangle \rangle,$$

and a is a possible response of O given the query $X_j Y_j$ (with j being the index of the oracle query tape).

Given the above, we can define the outcome of a computation as follows:

Definition 2.1.4 *A Quering Turing Machine* **can answer** *a on request i (given input oracle O) iff any configuration with state q_a is reachable from the initial configuration on i via the transitive reflexive closure of the union of intrinsic and extrinsic yield relations.*

The specifics of the above model merit a short discussion. Using the power of nondeterministic guesses (and, later on, talking about unambiguity properties) requires the ability to terminate branches on which the computation "went wrong" (i.e., invalid guesses have been made). The usual model incorporates such situations into the "reject" answer from the machine. However, we find it much easier to analyze complex scenarios (such as interplay between multiple computations) if such **failures** are made explicitly distinct from any possible answer the machine might give[5]. We will represent the failure by any configuration that does not yield a new one. For further simplification, we do not require the answer states q_a to be final—terminating or

[5] An example of such situation might be the Immerman-Szelepcsényi technique of inductive counting (see [13, 36]). There, the machine might reject a computation branch either because it has successfully verified that there is a path between the designated vertices, or because it has made some incorrect guesses on the way (which in our formalism would constitute a failure).

continuing the computation is seen as yet another nondeterministic choice. Moreover, as the input oracle will often be substituted with another Quering Turing Machine, we allow oracles in principle to give inconsistent ("nondeterministic") answers as well as to fail, and make all failures unrecoverable, i.e., propagating from any component to the whole computation.

With the above in place, the only externally visible difference between a Quering Turing Machine and an oracle is that the former needs to be provided with an input (oracle) before one can talk about its answers. Therefore we will freely use the term "oracle" to refer also to **closed** Quering Turing Machines—those with a specific input "plugged in."

A deterministic Quering Turing Machine "computes" a well-defined function. When allowed making guesses however, it might end up yielding different answers (or failing) on different computation branches. We could therefore define the machine to compute a relation (not necessarily a function). However, the notion of reductions based on this understanding has been shown in [19] to be very strong (applying them to **NL** yields the whole **NP**). Thus instead we interpret this "ambiguity" as "computing" multiple functions:

Definition 2.1.5 *An oracle* **consistently computes (returns)** *a string $X \in \Sigma^*$ iff, for any request i, it can only answer $X[i]$ (or fail, which is always allowed).*

Definition 2.1.6 *A Quering Turing Machine M is* **sound** *for the function $\phi : \alpha \to \beta$ iff, when supplied with an input oracle consistently returning $X \in \alpha$, it consistently computes $\phi(X)$.*

Note that according to the above definition, a single machine might be sound for many functions—in particular, a machine that always fails is sound for every possible function on Σ^*! Therefore we need a notion that would require a machine to succeed (i.e., not fail) on at least some nondeterministic branches. As it turns out however, we do not need it to succeed on all possible inputs:

Definition 2.1.7 *For any $a \in \Sigma$, we say that a Quering Turing Machine M is a-***total**[6] *for a function $\phi : \alpha \to \beta$ iff, whenever supplied an input oracle consistently returning $X \in \alpha$ and given the request i such that the i-th character of $\phi(X)$ is a, it can answer a. M is* **total** *for ϕ iff it is a-total for ϕ for every $a \in \Sigma$.*

[6]Using a term "complete" instead would make obvious the analogy to soundness and completeness of logical frameworks—however it could be easily confused with completeness with respect to a complexity class (even though one applies to languages, while the other to machines).

How do traditional (nondeterministic) deciders fit into this picture? Each can be seen as computing a characteristic function—making sure that Σ contains the two characters "0" and "1", the output can be seen as a member of Σ^*. The problem is that the Quering Turing Machine formed this way does not necessarily have to be sound for any function (as its answers are not guaranteed to be consistent among possible computation branches). However, if the original decider was an "existential" nondeterministic machine (i.e., accepting a word when there exists an accepting branch), we can turn all "0" answers into failures, achieving a sound, 1-total Quering Turing Machine for the original language. Analogously, a "universal" decider can be turned into a sound, 0-total Quering Turing Machine by failing all accepting paths. Furthermore, the connection goes in the other direction as well: we can wrap a sound, 1-total (0-total) Quering Turing Machine into an existential (universal) decider by treating every failure as a reject (accept, respectively).

The following simple result allows us to combine "partially total" Quering Turing Machines into a total one:

Proposition 2.1.8 *For a function ϕ, if for every $a \in \Sigma$ there exists a Quering Turing Machine sound and a-total for ϕ, then we can build a Quering Turing Machine sound and total for ϕ.*

Proof Let the a-total Quering Turing Machine for ϕ be called M_a. The desired machine M can be built as follows: nondeterministically guess the right answer a, and invoke the program for M_a. The soundness and totality of M are clear—it can never reach a wrong answer (as one of the M_a's would have to do it), and if it correctly guesses the answer a, M_a (being a-total) will "confirm" it. ■

As both input and output are implicit in a Quering Turing Machine, i.e., never available in their entirety, we need special arrangements to give a meaningful definition of the space consumed by it. First, we enrich the alphabet Σ with yet one more special symbol, the blank. Then we make sure that whenever an oracle is queried about an index beyond its output (and only then), it answers with the blank symbol[7]. Finally, we can define our space bounds:

Definition 2.1.9 [8] *The* **size (length)** *of an oracle is the smallest value of a query (i.e., the smallest character index) to which it might respond with a blank.*

[7] This can be achieved formally by adjusting Definition 2.1.5 and propagating this change wherever necessary. It is a very intuitive change, yet including it would affect the clarity of all further results—thus we opted for keeping it "under the hood."

[8] Definition 2.1.9 might seem roundabout, but remember that we are trying to take into consideration the fact that the input oracle might itself happen to be a QTM.

Definition 2.1.10 *A Quering Turing Machine M* **operates in space** $f(n)$ *iff, whenever supplied with an input oracle of size n, it reads or writes no more than $f(n)$ cells on all its tapes, including the oracle tape.*

The above definition does *not* take the size of the query into consideration. On one hand it eliminates the ill case of the request being very short compared to the input, while on the other it forces the machine to make sure it will not consume too much space just reading its request. However, for the algorithms that are "aware" of their own space bounds (and whose space bounds are space-constructible according to the usual definition, as in [34]), it is not an issue—they can discover the size of their input and return a blank if the request is beyond the longest possible output. Finally, just issuing oracle queries requires space logarithmic in the oracle size, and hence we will not consider sub-logarithmic space bounds.

Building on the above, we can naturally define some complexity classes:

Definition 2.1.11 *For a space bound $f(n) \geq \log(n)$, the class* $\mathbf{QFunc}(f(n))$ *consists of all functions that have sound and total Quering Turing Machines operating in space $O(f(n))$.*

Definition 2.1.12 *For a space bound $f(n) \geq \log(n)$, the class* $\mathbf{QSpace}(f(n))$ *(and* $\mathbf{co\text{-}QSpace}(f(n))$*) consists of those languages, whose characteristic functions have sound, 1-total (respectively, 0-total) Quering Turing Machines operating in space $O(f(n))$.*

The classes defined this way correspond naturally to the classical ones:

$$\mathbf{QSpace}(f(n)) = \mathbf{NSpace}(f(n)),$$
$$\mathbf{co\text{-}QSpace}(f(n)) = \mathbf{co\text{-}NSpace}(f(n)).$$

Furthermore, we can show the following:

Proposition 2.1.13 $\mathbf{QFunc}(f(n)) = \mathbf{FNSpace}(f(n))$.

Proof We will only show that $\mathbf{QFunc}(\log(n)) = \mathbf{FNL}$—the result generalizes easily to larger space bounds. It has been shown in [14] that several definitions of **FNL** are in fact equivalent. Here we use the following:

Definition 2.1.14 *A function ϕ is in* **FNL** *iff:*

- *there is a polynomial p such that for every X, $|\phi(X)| \leq p(|X|)$,*
- *the language $L_\phi = \{\langle X, i, a\rangle : \phi(X)[i] = a\}$ (known as the graph of ϕ) is in* **NL**.

It is now easy to see that this definition is equivalent to $\mathbf{QFunc}(\log(n))$. Logarithmic space bounds for the Quering Turing Machine induce a logarithmic bound on the size of the request, and thus a polynomial bound on the length of the output. The **NL** machine for L_ϕ can be turned into a Quering Turing Machine for ϕ by guessing the appropriate a as in the proof of Proposition 2.1.8. Turning a Quering Turing Machine for ϕ into an **NL** machine for L_ϕ can be obtained by comparing the answer (of the original QTM) with the a given on input (and rejecting on all failed branches). ■

The above proof exhibits a close relationship between the function classes and the intersections of "existential" and "universal" language classes. This relationship can be captured as follows:

Corollary 2.1.15 *For every space bound f:*

$$\phi \in \mathbf{QSpace}(f(n)) \cap \mathbf{co\text{-}QSpace}(f(n)) \Rightarrow \phi \in \mathbf{QFunc}(f(n)),$$
$$\phi \in \mathbf{QFunc}(f(n)) \Rightarrow L_\phi \in \mathbf{QSpace}(f(n)) \cap \mathbf{co\text{-}QSpace}(f(n)).$$

As mentioned earlier, we want to be able to compose Quering Turing Machines (i.e., use the output of one as the input to another). We do it in the most natural way, using separate tapes for the two machines, and invoking the program of the inner one whenever the outer one wants to make an input query. The following can be easily seen:

Proposition 2.1.16 *If M and N are Quering Turing Machines sound for ϕ and ψ, respectively, then their composition is sound for $\phi \circ \psi$. Moreover, if they are total, so is the composition.*

Proof The soundness of the composition can be seen by simply unrolling Definition 2.1.6. To show that it is also total, it is enough to note that the totality of N implies that a correct answer can be reached for any query, and thus M can always continue with its computation. ■

Furthermore, we can bound the space used by a composition of space-bounded Quering Turing Machines:

Proposition 2.1.17 *For any pair of Quering Turing Machines M and N, which run in space $f(n)$ and $g(n)$, respectively, their composition operates within space $O(f(2^{O(g(n))}))$.*

Proof On input of length n, N uses space $g(n) = \log(2^{g(n)}) \leq f(2^{g(n)})$. It can answer (with a non-blank) to requests of length at most $g(n)$, and thus its output has length $2^{O(g(n))}$. Given an input that long, M can use no more than $f(2^{O(g(n))})$ space. The overall space needed is bounded by the sum of these two, which is in $O(f(2^{O(g(n))}))$, as required. ■

2.2 Quering Computation Graphs

We now proceed to present the "static" way of looking at query-based computation, extending the concept of configuration graphs to express issuing queries and producing answers.

Definition 2.2.1 *A* **Quering Computation Graph** $\langle V, E, S, c\rangle$ *is a directed graph with a distinguished subset*[9] *of* **source vertices** $S \subseteq V$*, together with a coloring function* $c : V \cup E \to \Sigma \cup \{\perp\}$ *($\perp$ denoting "no color"), such that there is at most one vertex of every color (i.e.,* $c(u) = c(v) \neq \perp \Rightarrow u = v$*), but there may be many edges of a single color.*

The "colors" are purely a conceptual convenience—they correspond to the symbols of the alphabet. Intuitively, we take the usual configuration graph representation of nondeterministic computation (i.e., edges following the intrinsic yield relation) and add colored vertices and edges to represent answer configurations and transitions dependent on oracle queries (i.e., extrinsic yield), respectively. Note that we specify a set (i.e., not a single vertex) as the "source"—the computation may start in different configurations, depending on the initial request. We will formalize this dependency later on. Moreover, the definition of a Quering Computation Graph requires every possible answer to be returned in at most one configuration—but this can be easily enforced if the machine clears the content of all its tapes before moving into an answer state.

Oracles (and, equivalently, closed Quering Turing Machines) do not issue any input queries. Thus their operation can be modeled with the following restriction of Quering Computation Graphs:

Definition 2.2.2 *A* **Closed Computation Graph** *is a Quering Computation Graph in which all edges are uncolored.*

To compose computations (i.e., making a machine use the answers of another one as its input) we need a corresponding operation on Quering Computation Graphs. Let us define it as follows:

Definition 2.2.3 *Given Quering Computation Graphs* $G = \langle V_G, E_G, S_G, c_G\rangle$ *and* $H = \langle V_H, E_H, S_H, c_H\rangle$*, and a function* $f : V_G \to S_H$*, the* f**-composition** *of* G

[9] All results in this section hold regardless of S, so we could assume that $S = V$ and omit it from the definition. However, in later sections we will use S as a means of simplifying many arguments.

and H (denoted $G \circ_f H$) is a Quering Computation Graph $\langle V, E, S, c\rangle$ with:

$$
\begin{aligned}
V =& V_G \times V_H,\\
S =& \{\langle u, f(u)\rangle : u \in S_G\},\\
E =& E_1 \cup E_2 \cup E_3, \text{ where}\\
& E_1 = \{\langle\langle u, f(u)\rangle, \langle v, f(v)\rangle\rangle : \langle u, v\rangle \in E_G, c_G(\langle u, v\rangle) = \bot\},\\
& E_2 = \{\langle\langle u, x\rangle, \langle u, y\rangle\rangle : \langle x, y\rangle \in E_H\},\\
& E_3 = \{\langle\langle u, x\rangle, \langle v, f(v)\rangle\rangle : \langle u, v\rangle \in E_G, c_G(\langle u, v\rangle) = c_H(x) \neq \bot\},\\
c(\langle u, x\rangle) =& \begin{cases} c_G(u) & \text{if } x = f(u),\\ \bot & \text{otherwise,}\end{cases}\\
c(\langle\langle u, x\rangle, \langle v, y\rangle\rangle) =& \begin{cases} c_H(\langle x, y\rangle) & \text{if } u = v,\\ \bot & \text{otherwise.}\end{cases}
\end{aligned}
$$

We say that an edge in the f-composition is of **type 1, 2 or 3**, *depending on which of the sets E_1, E_2 and E_3 it belongs to.*

The correspondence between f-composition and "plugging in" one Quering Turing Machine as an input oracle of another is as follows. The function f represents a way of extracting the oracle query, and so the initial configuration of the inner machine, from the configuration of the outer machine. Usually it is as simple as taking a segment of the configuration corresponding to the contents of the oracle tape, in which case we will omit f entirely and simply write $G \circ H$. The transitions in the composed computation can be divided into three groups (edges of type 1, 2, and 3, respectively): the uncolored (i.e., not depending on the oracle answers) transitions of the outer machine, the inner computation, and transferring the answer of the inner to the outer machine (in which case the color of the answer has to match that of the "conditional" edge).

The following is an expected consequence of our definitions:

Observation 2.2.4 *The composition of two Quering Computation Graphs is a Quering Computation Graph. The composition of a Quering Computation Graph and a Closed Computation Graph is a Closed Computation Graph.*

It is natural to require the f-composition to be associative. Before we can claim that, we need to address a technical detail of our notation. We would like to be able to write

$$G \circ_f H \circ_g I,$$

and understand it as being parenthesized in any order. However, if we group the left terms first, we would need $f : V_G \to S_H$ and $g : V_G \times V_H \to S_I$, while in the other

case we should have $f : V_G \to S_H \times S_I$ and $g : V_H \to S_I$. To reconcile these, let us go back to what these functions are supposed to represent in the composition. Each of them extracts the initial state of the "inner" machine from the current state of the "outer" one. It is clear that this dependency should not change with the context in which this composition of machines is used. Thus we will assume in the above expression $f : V_G \to S_H$ and $g : V_H \to S_I$, and then apply the function to only the last component, and produce only the first component of the respective tuple. Having clarified that we can now prove the following:

Proposition 2.2.5 *f-composition is associative.*

Proof Consider the expressions

$$\begin{aligned} L &:= (G \circ_f H) \circ_g I \text{ and} \\ R &:= G \circ_f (H \circ_g I). \end{aligned}$$

In the light of the above discussion, we formally mean

$$\begin{aligned} L &:= (G \circ_f H) \circ_{g'} I \text{ and} \\ R &:= G \circ_{f'} (H \circ_g I), \end{aligned}$$

where

$$\begin{aligned} g'(\langle s, u \rangle) &= g(u), \\ f'(s) &= \langle f(s), g(f(s)) \rangle. \end{aligned}$$

The sets of vertices and source vertices of L and R are trivially equal, and so are their colorings. It remains to show that the same holds true for the sets of edges. Let us write $x \to^a y \in G$ to denote that $\langle x, y \rangle \in E_G$ and $c_G(\langle x, y \rangle) = a$, and consider a hypothetical edge $\langle s, u, x \rangle \to \langle t, v, y \rangle$ in either L or R. Unwinding the definition of composition we get:

$$\begin{aligned} \langle s, u \rangle \to^\perp \langle t, v \rangle \in G \circ_f H \equiv \\ (s \to^\perp t \in G \wedge u = f(s) \wedge v = f(t)) \\ \vee (s = t \wedge u \to^\perp v \in H) \\ \vee (s \to^a t \in G \wedge v = f(t) \wedge c_H(u) = a \neq \perp), \\ \langle s, u \rangle \to^a \langle t, v \rangle \in G \circ_f H \equiv (\text{with } a \neq \perp) \\ (s = t \wedge u \to^a v \in H), \end{aligned}$$

$$\begin{aligned}
\langle u,x\rangle \rightarrow^{\perp} \langle v,y\rangle \in H \circ_g I \equiv \\
&(u \rightarrow^{\perp} v \in H \wedge x = g(u) \wedge y = g(v)) \\
&\vee(u = v \wedge x \rightarrow^{\perp} y \in I) \\
&\vee(u \rightarrow^{a} v \in H \wedge y = g(v) \wedge c_I(x) = a \neq \perp), \\
\langle u,x\rangle \rightarrow^{a} \langle v,y\rangle \in H \circ_g I \equiv (\text{with } a \neq \perp) \\
&(u = v \wedge x \rightarrow^{a} y \in I),
\end{aligned}$$

$$\begin{aligned}
\langle s,u,x\rangle \rightarrow^{\perp} \langle t,v,y\rangle \in L \equiv \\
&(s \rightarrow^{\perp} t \in G \wedge u = f(s) \wedge v = f(t) \wedge x = g(u) \wedge y = g(v)) \\
&\vee(s = t \wedge u \rightarrow^{\perp} v \in H \wedge x = g(u) \wedge y = g(v)) \\
&\vee(s \rightarrow^{a} t \in G \wedge v = f(t) \wedge c_H(u) = a \neq \perp \wedge x = g(u) \wedge y = g(v)) \\
&\vee(s = t \wedge u = v \wedge x \rightarrow^{\perp} y \in I) \\
&\vee(s = t \wedge u \rightarrow^{a} v \in H \wedge y = g(v) \wedge c_I(x) = a \neq \perp), \\
\langle s,u,x\rangle \rightarrow^{a} \langle t,v,y\rangle \in L \equiv (\text{with } a \neq \perp) \\
&(s = t \wedge u = v \wedge x \rightarrow^{a} y \in I),
\end{aligned}$$

$$\begin{aligned}
\langle s,u,x\rangle \rightarrow^{\perp} \langle t,v,y\rangle \in R \equiv \\
&(s \rightarrow^{\perp} t \in G \wedge u = f(s) \wedge v = f(t) \wedge x = g(u) \wedge y = g(v)) \\
&\vee(s = t \wedge u \rightarrow^{\perp} v \in H \wedge x = g(u) \wedge y = g(v)) \\
&\vee(s = t \wedge u = v \wedge x \rightarrow^{\perp} y \in I) \\
&\vee(s = t \wedge u \rightarrow^{a} v \in H \wedge y = g(v) \wedge c_I(x) = a \neq \perp) \\
&\vee(s \rightarrow^{a} t \in G \wedge v = f(t) \wedge y = g(v) \wedge c_H(u) = a \neq \perp \wedge x = g(u)), \\
\langle s,u,x\rangle \rightarrow^{a} \langle t,v,y\rangle \in R \equiv (\text{with } a \neq \perp) \\
&(s = t \wedge u = v \wedge x \rightarrow^{a} y \in I),
\end{aligned}$$

from which it can be seen (the lines for R being a permutation of those for L), that the edges (and their colors) match and thus $L = R$, as required. ■

When relating Quering Computation Graphs to Quering Turing Machines, we need to be able to talk about the size of a graph for the particular machine. However, as the space bounds on the computation do not depend on the length of the request (only on the input), we already have infinitely many distinct initial configurations. To deal with that issue, let us note that a space-bounded Quering Turing Machine, once a specific input oracle is supplied, will not even read the bits of the request beyond

a specific point. Thus, all computations for requests differing only at these "far bits" are going to be exactly identical, allowing us to divide all configurations (and thus the vertices of the graph) into finitely many equivalence classes and count them instead. This makes the size of the Quering Computation Graph dependent on the size of the input only, which is what we are used to in the "standard" model. The bound on the graph size is of course an exponential function of the bound on the space consumed by the machine:

Proposition 2.2.6 *A Closed Computation Graph, corresponding to a Quering Turing Machine working in space $f(n)$ and supplied with an input of size n, has size $2^{O(f(n))}$.*

2.3 Ambiguity

To capture the degree of ambiguity of a computation, we look at the shape of its Quering Computation Graph:

Definition 2.3.1 *For a family $\mathcal{C}$ of Closed Computation Graphs, we say that a Quering Turing Machine M is a* $\mathcal{C}$**-machine** *iff, when supplied with any consistent input, its Closed Computation Graph belongs to $\mathcal{C}$.*

The above notion can be naturally extended to complexity classes:

Definition 2.3.2 [10] *The class* $\mathcal{C}$**-QFunc**$(f(n))$ *consists of all functions that have sound, total $\mathcal{C}$-machines operating in space $O(f(n))$. Analogously we can define the classes* $\mathcal{C}$**-QSpace**$(f(n))$ *and* co-$\mathcal{C}$**-QSpace**$(f(n))$.

To be able to talk about classical deterministic and non-deterministic algorithms, we introduce two classes of Closed Computation Graphs: **D**—those of out-degree 1, and **N**—the class of all Closed Computation Graphs.

Computational (un)ambiguity is expressed by limits on the number of distinct ways to reach (from a source vertex) a node in the Closed Computation Graph. The variant of this restriction will be denoted by specifying the following (orthogonal) aspects:

1. The number of paths allowed (as a function of the size of the graph, with k and p standing for arbitrary constants and polynomials, respectively),

[10] Note that the classes from Definition 2.3.2 are *semantic*, i.e., defined by machines that have to meet undecidable criteria (here, their Closed Computation Graph on *every* possible input being in class $\mathcal{C}$). Therefore we cannot immediately provide complete problems for them.

2. The types of paths that are counted:
   - **A** = all paths,
   - **S** = simple paths (i.e., without loops),
   - **M** = minimal-length paths.
3. The types of target nodes of the paths of interest:
   - **A** = all nodes,
   - **F** = colored ("final") nodes only.

For example, p**AF**-graphs are those Closed Computation Graphs with at most $p(n)$ paths between a source and any final vertex, and 1**MA**-graphs—those with a unique minimal-length path to any (reachable) vertex.

Note that within this framework, we could consider notions of "unambiguity" other than based on path counts, for example planar graphs (this would allow our framework to capture the recent result of [5], showing that reachability on directed planar graphs can be solved in **UL**). However, the corresponding restrictions of the machines (e.g., "planar machines") are unnatural, and furthermore an equivalent of Lemma 2.4.2 (and thus also Proposition 2.4.4) does not hold for every possible class of Closed Computation Graphs.

In the above notation, a number of classical complexity classes can be captured in a unified manner. In particular

$$\begin{aligned}
\mathbf{L} &= \mathbf{D\text{-}QSpace}(\log(n)),\\
\mathbf{FL} &= \mathbf{D\text{-}QFunc}(\log(n)),\\
\mathbf{UL} &= 1\mathbf{AF\text{-}QSpace}(\log(n)),\\
\mathbf{RUL} &= 1\mathbf{AA\text{-}QSpace}(\log(n)),\ \text{and}\\
\mathbf{FewL} &= \bigcup_{p(n)\in n^{O(1)}} p\mathbf{AF\text{-}QSpace}(\log(n)).
\end{aligned}$$

The class **FewUL** (defined in [6] under the name of **LogFewNL**, requires a unique computation path to every accepting configuration, but allows multiple such configurations to exist) does not seem to be directly captured by our framework. However, the corresponding function class can be shown (by a technique similar to Theorem 6 of [8]) to be the same as for **UL**:

Proposition 2.3.3 $\mathbf{FL}^{\mathbf{FewUL}} = \mathbf{FL}^{\mathbf{UL}}$.

Proof Since inclusion is obvious (**UL** $\subseteq$ **FewUL**), it is enough to show how to simulate a single call to a **FewUL** oracle. Let M be the machine we want to simulate. Now if we define

$$L_M := \{\langle X, C\rangle | C \text{ is an accepting configuration of } M, \text{ reachable on input } X\},$$

we will see that $L_M \in$ **UL**, as M can have at most one computation path to any accepting configuration. Now the question of whether M accepts X can be rephrased as of whether there exists a C such that $\langle X, C\rangle \in L_M$. As the maximal size of configurations is logarithmic in the length of X, the above can be answered by enumerating over all possible Cs and quering a **UL** oracle for L_M. ■

For convenience, we will use $\mathbb{ALL}$ to denote the set of Closed Computation Graph classes obtained from any combination of the above restrictions (including **D**-graphs and the class of all graphs), $\mathbb{UNI}$—those with a "unique" path of a given type, and $\mathbb{REM}$—those closed under edge removal (these do not contain the classes based on minimum-length paths, as removing an edge might invalidate a minimum length path and make several longer paths take its place):

$$\begin{aligned}
\mathbb{ALL} &:= \{\mathbf{D}, \mathbf{N}\} \cup \{p\mathbf{AF}, p\mathbf{AA}, p\mathbf{SF}, p\mathbf{SA}, p\mathbf{MF}, p\mathbf{MA} | p \in n^{O(1)}\},\\
\mathbb{UNI} &:= \{\mathbf{D}, 1\mathbf{AF}, 1\mathbf{AA}, 1\mathbf{SF}, 1\mathbf{SA}, 1\mathbf{MF}, 1\mathbf{MA}\},\\
\mathbb{REM} &:= \{\mathbf{D}, \mathbf{N}\} \cup \{p\mathbf{AF}, p\mathbf{AA}, p\mathbf{SF}, p\mathbf{SA} | p \in n^{O(1)}\}.
\end{aligned}$$

We can also extend Proposition 2.1.8 to $\mathcal{C}$-machines:

Proposition 2.3.4 *For a Closed Computation Graph class $\mathcal{C} \in \mathbb{ALL}$ and a function ϕ, if for every $a \in \Sigma$ there exists a $\mathcal{C}$-machine sound and a-total for ϕ, we can build a single $\mathcal{C}$-machine, sound and total for ϕ.*

Proof Using the construction from the proof of Proposition 2.1.8 almost works. What can go wrong, is that we might introduce more vertices with the the same color (or, if we merge them together, increase the number of paths of interest). To prevent this, in each M_a we make all answers but a fail (in the Closed Computation Graph this can be seen as removing all colors except for a). This way each M_a is solely responsible for returning the answer of a, the answer vertices are unique, and the path count does not grow. ■

2.4 Reductions

To talk about the relative complexity of different problems (functions), we would like to introduce a notion analogous to log-space reductions, a notion that would allow nondeterminism but at the same time limit its level of ambiguity. The obvious approach would be to adapt the usual many-one reductions of Karp (ϕ being reducible to ψ iff there exists a θ computable in the appropriate class, such that $\phi = \psi \circ \theta$; for detailed definitions for decision and functional problems, see [17] and [4]). However, this requires the ranges of the functions being compared to be identical—a limitation which we find too strict. Moreover, even the very nature of Quering Turing Machines (being queried multiple times about different characters of their output) suggests employing some variant of Turing reductions.

In the space-bounded setting however, the notion of Turing reducibility becomes very sensitive to the exact definition. By analogy to the model with a read-only input tape and append-only output tape, it is natural to make oracle tape append-only and not subject to the space bounds. This turns out to be much too strong—it has been shown in [23] that even a *deterministic, constant-space* machine, when given such access to an oracle for a particular language in $\mathbf{CFL} \cap \mathbf{L}$, can decide *all recursive* languages (the basic idea is that if the contents of the oracle tape are preserved over the queries, one can use it to simulate two stacks by appending symbols corresponding to pushing and popping, and two stacks are sufficient to have a general model of computation).

Specifying that the oracle tape is erased after every query (as proposed by Ladner and Lynch in [18]) makes the machine much weaker, but still too strong—it is possible to decide 3-CNF satisfiability (and thus every language in $\mathbf{NP}$) in $\mathbf{NL}^{\mathbf{L}}$ by first copying the input formula to the oracle tape, followed by a (guessed) satisfying assignment, and then quering an oracle for the formula value problem, which is in $\mathbf{L}$.

In [31], Ruzzo, Simon, and Tompa have suggested a further restriction of that model, requiring the machine to operate deterministically from the moment of the first write to the oracle tape, up until the oracle is queried—this way the complete contents of the query depend deterministically on the configuration in which the machine started to write the query. This approach however, as all approaches (even completely deterministic) with unbounded oracle tapes, has been shown in [24] *not* to be robust with respect to the number of oracles (even if the oracles are for the same language).

It seems then, that for a sufficiently weak and robust definition of reducibility, we have to make the oracle tape(s) subject to the space bounds. But in that case our notion would not contain the natural many-one reductions: a language decidable in space $O(n)$ but not in space $O(\log(n))$ (guaranteed to exist by the Space Hierarchy

Theorems, see [34]) would not be reducible to itself, as the machine could not pass its input (even unchanged) to the oracle. We can fix this issue by letting the oracle access the machine's input (which, in case of Quering Turing Machines is implicit anyway). However, if we restrict the queries to different requests on the same input, we will end up with a notion "dual" to many-one reductions (with the functions being composed in the opposite order)—again too weak for our purposes.

From the above discussion we see a need for a way of transforming the input that would fit within some required restrictions. A similar concept has been introduced in [31], where it has also been shown to be equivalent to the "deterministic query writing" in case of a single oracle tape. We extend it to consider ambiguity: taking a function class $\mathcal{C}$-**QFunc**$(f(n))$ and requiring the desired "type" of modifications (e.g., edge removal) to be computable in that class, while making the "parameters" of the change (e.g., which edge to remove) part of the oracle query. Another way of looking at it is as a function outputting a sequence of objects corresponding to every possible modification of the desired type, with the request containing the index of the object to extract (next to the usual index of the character we are interested in). The resulting model ends up being close to many-one reducibility (as it is based on function composition), but with each reduction consisting of two parts—the family of input transformations, and the actual algorithm, allowed to query the oracle on any member of this family:

Definition 2.4.1 *A function $\phi : \alpha \to \beta$ is $\mathcal{C}/\mathcal{D}$-**reducible** to a function $\psi : \gamma \to \delta$ (written $\phi \preceq^{\mathcal{C}}_{\mathcal{D}} \psi$) iff there exist a (possibly infinite) family of functions $\theta_i : \alpha \to \gamma$, a polynomial p, and a function $\xi : \delta^* \to \beta$ such that:*

- *taking $\theta(X) := \langle \theta_i(X) \rangle_{i \leq p(|X|)}$ we have $\theta \in \mathcal{C}$-**QFunc**$(\log(n))$ (i.e., the functions θ_i can be "uniformly" computed in $\mathcal{C}$-**QFunc**$(\log(n))$),*
- *$\xi \in \mathcal{D}$-**QFunc**$(\log(n))$,*
- *for every $X \in \alpha$, $\xi(\langle \psi \circ \theta_i(X) \rangle_i) = \phi(X)$.*

If the complexity class of θ_i or ξ is not known, we will use the function itself as the subscript/superscript of $\preceq$. Moreover, we will omit the subscript/superscript entirely if the corresponding function is the identity.

The following technical result is the key to making use of unambiguous, nondeterministic reductions:

Lemma 2.4.2 *For any transformations $\psi : \alpha \to \beta$, $\phi : \beta \to \delta$, and Closed Computation Graph classes $\mathcal{C} \in \mathbb{UNI}$ and $\mathcal{D} \in \mathbb{ALL}, \mathcal{C} \subseteq \mathcal{D}$, if there exist:*

- *a $\mathcal{D}$-machine M sound (and total) for ϕ, working in space $f(n)$, and*
- *a $\mathcal{C}$-machine N sound (and total) for ψ, working in space $g(n)$,*

then we can build a $\mathcal{D}$-machine sound (and total) for the composition $\phi \circ \psi$, requiring space $O(f(2^{O(g(n))}))$.

Proof Using the natural composition of M and N meets the soundness, totality, and space requirements according to Propositions 2.1.16 and 2.1.17. It remains to show how to obtain the desired (un)ambiguity properties. First, let us make the following simple observation about the composition:

Observation 2.4.3 *Every path in the f-composition of Quering Computation Graphs G and H has the following structure:*

- *(optionally) a path in one of the copies of H (edges of type 2), followed by one edge of type 3,*
- *a (possibly empty) path in G, with uncolored edges followed directly (as type 1), and colored edges represented by paths in copies of H (each ending at H's colored vertex, with a type 3 edge following it),*
- *(optionally) a path in one of the copies of H.*

The Closed Computation Graph corresponding to the new machine on any input X is of course the composition of the Quering Computation Graph of M and the Closed Computation Graph of N on X. Therefore its paths follow our observation. Requiring $\mathcal{C}$ to be a subset of $\mathcal{D}$ makes its ambiguity constraints apply to at least the types of paths we are concerned with. Making it one of the $\mathbb{UNI}$ classes prevents N from increasing the number of paths of interest in the overall computation within a single query processing. As we are about to show, with some precautions we can avoid any other paths of interest from appearing and thus complete the proof.

The cases in which we consider all paths (to either all reachable or all final vertices) are immediate consequences of Observation 2.4.3. If we count simple paths only, it is enough to notice that a cycle in the composition graph must mirror one in either of the components. The matters get slightly more complicated with minimum-length paths, as we must make some guarantees regardless of the time needed to process any N queries. To achieve that, we introduce an additional counter tape, and we make every step of M take an amount of time larger than all possible N queries combined (in the query graph it might be seen as making type 1 and type 3 edges "longer"—i.e., replacing them with sequences of edges).

As M uses space $f(2^{O(g(n))})$, it cannot take more than $2^{O(f(2^{O(g(n))}))}$ steps. If each of them was an oracle query, they would add up to at most $2^{O(f(2^{O(g(n))})+g(n))}$ steps. Therefore a counter of length $O(f(2^{O(g(n))}) + g(n)) = O(f(2^{O(g(n))}))$ is enough for the purpose. Now a minimum-length path in the new machine must be a minimum length path of M augmented with some queries. Moreover, each of them has to be minimum-length within the query, or otherwise a shorter overall path would exist to the same configuration. ■

The next proposition justifies the definition of our notion of reduction, as it shows that the right properties of computation graphs are maintained after the reduction is applied:

Proposition 2.4.4 *For Closed Computation Graph classes $\mathcal{C} \in \mathbb{UNI}$ and $\mathcal{D} \in \mathbb{ALL}$, $\mathcal{C} \subseteq \mathcal{D}$, and transformations $\phi : \alpha \to \beta$ and $\psi : \gamma \to \delta$, if $\phi \preceq_{\mathcal{D}}^{\mathcal{C}} \psi$ and $\psi \in \mathcal{C}$-**QFunc**$(f(n))$, then $\phi \in \mathcal{D}$-**QFunc**$(f(n))$.*

Proof Take the functions θ and ξ to be as in Definition 2.4.1. Define $\Psi : \gamma^* \to \delta^*$ as

$$\Psi(\langle X_i \rangle_i) := \langle \psi(X_i) \rangle_i.$$

Ψ can be computed in $\mathcal{C}$-**QFunc**$(f(n))$—just preserve the index i of the part of the output you are asked for (as the value of i is bounded by a polynomial in $|X|$, logarithmic space is enough to make a copy of it), and use it whenever making a query to the input oracle. Moreover, $\phi = \xi \circ \Psi \circ \theta$. Now, applying Lemma 2.4.2 to the machines for ξ, Ψ and θ, we get an $O(f(n))$-space bounded $\mathcal{D}$-machine for ϕ, as required. ■

Chapter 3

Results

In this chapter we define the function problems of our interest. We also present some simple yet interesting dependencies between these problems, and between the unambiguous complexity classes. Finally, we show how some well-known theorems can be viewed (and in some cases improved) in our framework.

3.1 Problems with Promises

The computation of a Quering Turing Machine on a specific input and request can be viewed as following a path in the corresponding Closed Computation Graph. The question whether it returns a specific answer is the same as asking whether a given colored vertex is reachable from the given source. The problem of **reachability** (denoted **Reach**, also known as **st-connectivity** or **graph accessibility problem**) is well known (see [15]) to be *the* canonical problem for space-bounded computation.

Since we are working with restricted classes of graphs, it is natural to ask the question whether a given graph meets the specific criteria, i.e., whether it belongs to the given class. This introduces a family of **testing** problems, with **Test**$\mathcal{C}$ denoting the problem of checking whether a given Closed Computation Graph belongs to the class $\mathcal{C}$.

We are also going to consider the (functional) problem of **path counting**. Here **Count**$\mathcal{X}$ will denote counting all paths of type $\mathcal{X}$ (following the notation for ambiguity classes, e.g., **SF** denoting simple paths from start to colored vertices), taking the maximum over all start-end pairs. In particular, **CountSF** is known to be complete for the class **#L** (by definition, **#L** contains all functions counting the accepting computations of **NL** machines; the equivalence between machines and computation graphs is well known, see Proposition 3.2.1 below for details). In this work we are going to focus on bounded version(s) of counting—the problem **Count**$k\mathcal{X}$ will be the one of count-

ing *up to* k paths (i.e., the set of answers being $\{0, 1, \ldots, k-1, k^+\}$, with k^+ denoting "k or more") of type $\mathcal{X}$. In this notation, a result of Allender, Reinhardt and Zhou (Theorem 5.1 in [3]) implies that for a polynomial p, **Count**p**SF** $\in$ **QFunc**$(\log(n))$ (i.e., limited counting *can* be solved nondeterministically in logarithmic space, but with *no* bounds on ambiguity).

Most of the problems discussed might vary in difficulty when given different "promises" about the input graph. Therefore we employ the following consistent notation: $\mathcal{C}$-α denotes the problem α on graphs in class $\mathcal{C}$. For example, 1**MATest1AA** denotes the problem of testing whether the paths to all reachable vertices are unique (indicated by the suffix 1**AA**), restricted to graphs when the minimum-length paths to all reachable vertices are guaranteed to be unique (indicated by the prefix 1**MA**).

3.2 Basic Observations

First let us see how the problem of reachability fits in the new framework. The following result is an extension of the well known fact of **Reach** being complete for **NL**:

Proposition 3.2.1 *For any class $\mathcal{C}$, $\mathcal{C}$-***Reach** *is complete (with respect to deterministic reductions) for $\mathcal{C}$-***QSpace**$(\log(n))$*, i.e., for any language $\alpha \subseteq \Sigma^*$:*

$$\alpha \in \mathcal{C}\text{-}\mathbf{QSpace}(\log(n)) \iff \alpha \preceq_{\mathrm{D}}^{\mathrm{D}} \mathcal{C}\text{-}\mathbf{Reach}.$$

Proof If α belongs to $\mathcal{C}$-**QSpace**$(\log(n))$, it can be solved by an $O(\log(n))$-space $\mathcal{C}$-machine. By definition the graph of this machine belongs to $\mathcal{C}$, and whether the machine returns a specific answer is equivalent to a specific vertex being reachable from the source.

For an algorithm, a machine that simply guesses a path edge by edge is enough. It works in space $O(\log(n))$, as it is enough to remember 2 nodes at a time. Its computation graph is identical to its input, thus it is a $\mathcal{C}$-machine. It is sound (it can only guess a path if there is one) and yes-total (there is a nondeterministic branch corresponding to every path in the graph). ■

Corollary 3.2.2 *For any class $\mathcal{C}$, $\mathcal{C}$-***Reach** *is hard for $\mathcal{C}$-***QFunc**$(\log(n))$.

The machine from the proof of Proposition 3.2.1 is not necessarily total, and thus we cannot claim that $\mathcal{C}$-**Reach** is always $\mathcal{C}$-**QFunc**$(\log(n))$-complete. However, the following can be seen as a consequence of Proposition 2.1.8:

Proposition 3.2.3 *For any class $\mathcal{C}$, the following conditions are equivalent:*

- $\mathcal{C}$-**QSpace**$(\log(n))$ = **co-**$\mathcal{C}$-**QSpace**$(\log(n))$,
- $\mathcal{C}$-**Reach** $\in$ **co-**$\mathcal{C}$-**QSpace**$(\log(n))$,
- $\mathcal{C}$-**Reach** $\in \mathcal{C}$-**QFunc**$(\log(n))$,
- $\mathcal{C}$-**Reach** *is* $\mathcal{C}$-**QFunc**$(\log(n))$*-complete.*

We have mentioned that the counting problems relate naturally to both reachability and testing. Formally, these natural relationships are:

Observation 3.2.4 *For any class $\mathcal{C}$, all the following denote the same function:*

$$\mathcal{C}\text{-}\mathbf{Reach} = \mathcal{C}\text{-}\mathbf{Count1AF} = \mathcal{C}\text{-}\mathbf{Count1SF} = \mathcal{C}\text{-}\mathbf{Count1MF}.$$

Proposition 3.2.5 *For any class $\mathcal{C}$ and path restriction $\mathcal{X}$,*

$$\mathcal{C}\text{-}\mathbf{Test}k\mathcal{X} \preceq_{\mathbf{D}} \mathcal{C}\text{-}\mathbf{Count}(k+1)\mathcal{X}.$$

Proof The graph G is in $k\mathcal{X}$ iff there are at most k paths of type $\mathcal{X}$. Therefore the desired deterministic reduction maps all answers up to k to "yes", and that of "$k+1$ or more" to "no". ■

How do counting problems for different values of k relate to each other? Obviously, decreasing the counter limit can only makes the problem easier, as we can simply glue together the previously distinct answers:

Observation 3.2.6 *For any class $\mathcal{C}$, path restriction $\mathcal{X}$ and bound k:*

$$\mathcal{C}\text{-}\mathbf{Count}k\mathcal{X} \preceq_{\mathbf{D}} \mathcal{C}\text{-}\mathbf{Count}(k+1)\mathcal{X}.$$

In the other direction, the following can be shown. Recall that $\mathbb{REM}$ is the family of Closed Computation Graph classes closed under edge removal.

Proposition 3.2.7 *For any class $\mathcal{C} \in \mathbb{REM}$ and constant $k \geq 1$,*

$$\mathcal{C}\text{-}\mathbf{Count}(k+1)\mathbf{SF} \preceq_{\mathbf{D}}^{\mathcal{C}\text{-}\mathbf{Count1SF}} \mathcal{C}\text{-}\mathbf{Count}k\mathbf{SF}.$$

In words, we show that given a graph G from a class $\mathcal{C} \in \mathbb{REM}$, and an algorithm for $\mathcal{C}$-**Count1SF**, we can create a sequence of graphs $\langle G_i \rangle_i$ such that the answer to $\mathcal{C}$-**Count**$k+1$**SF**(G) can be obtained deterministically from the answers $\langle \mathcal{C}\text{-}\mathbf{Count}k\mathbf{SF}(G_i) \rangle_i$.

Proof Let us first look at the case of $k = 1$. Our algorithm works as follows:

on graph G:
1. if $\mathcal{C}$-**Count1SF**$(G) = 0$, answer 0
2. for every edge e in G, let G_e be the same as G but with e removed
3. for every edge e in G, let $c_e :=$ $\mathcal{C}$-**Count1SF**(G_e)
4. remove edges from G, leaving only those for which $c_e = 0$; call the result G'
5. answer $2 -$ $\mathcal{C}$-**Count1SF**(G')

First, let us discuss the graph modification. The steps 2 to 4 are just a conceptual convenience—the graphs G_e and G' are never produced explicitly. Instead, whenever asked whether an edge $e = \langle u, v \rangle$ is in G', we answer "yes" if both

- $e \in G$, and
- $\mathcal{C}$-**Count1SF**$(G - e) = 0$.

If there is no path between the source s and the target t in G, we will discover it in step 1. If there is exactly one such simple path, removing any of its edges would disconnect s from t. Thus the same path is going to be present in G' and the algorithm will return 1. If there are at least two simple paths, consider the vertex x at which they diverge for the first time. Removing any single outgoing edge of x will not disconnect s and t, and thus x will become a sink in G'. But as any path from s to t has to go through x, there will be none, and our algorithm will correctly return 2. The procedure is thus sound and total. Moreover, as the only modification of the graph is removing edges and we have chosen $\mathcal{C}$ to be one of the classes closed under this operations, all calls to $\mathcal{C}$-**Count1SF** will have their promise fulfilled.

We can now proceed to higher values of k. It is clear that we only need to distinguish the cases of "exactly k" and "$k + 1$ or more" paths (the other answers can be copied exactly from $\mathcal{C}$-**Count**k**SF**). Having at least 2 paths guarantees the existence of the first point of divergence, as discussed above. Moreover, the same way of deleting edges makes the vertices on the "common prefix" of the paths have out-degree 1 in G', which allows us to deterministically find the split-point x. Now, x has at least two "meaningful" successors (on paths to the target)—thus if there are exactly k paths of interest, at most $k - 1$ of them can pass through any of the successors. Therefore, if we modify the graph to leave exactly one of x's outgoing edges (repeatedly for each of them), we can use $\mathcal{C}$-**Count**k**SF** to determine the exact count of the paths of interest. ■

Corollary 3.2.8 *For any class* $\mathcal{C} \in \mathbb{REM}$ *and constant* $k \geq 1$,

$$\mathcal{C}\text{-}\mathbf{Count1SF} \preceq_{\mathrm{D}} \mathcal{C}\text{-}\mathbf{Count}k\mathbf{SF} \preceq_{\mathrm{D}}^{\mathcal{C}\text{-}\mathbf{Count1SF}} \mathcal{C}\text{-}\mathbf{Count1SF}.$$

Corollary 3.2.9 *For any classes* $\mathcal{C} \in \mathbb{REM}$, $\mathcal{D} \in \mathbb{UNI}$, *and constant* $k \geq 1$,

$$\mathcal{C}\textbf{-Reach} \in \mathcal{D}\textbf{-QFunc}(\log(n)) \iff \mathcal{C}\textbf{-Count}k\textbf{SF} \in \mathcal{D}\textbf{-QFunc}(\log(n)).$$

It is also possible to extend the above results to count all, instead of only simple, paths:

Proposition 3.2.10 *For any class* $\mathcal{C} \in \mathbb{REM}$ *and constant* $k \geq 1$,

$$\mathcal{C}\textbf{-Count}(k+1)\textbf{AF} \preceq_{\mathbf{D}}^{\mathcal{C}\textbf{-Count1AF}} \mathcal{C}\textbf{-Count}k\textbf{AF}.$$

Proof First, let us note that if there is any non-simple path from the source to the target, we can obtain infinitely many paths by choosing the number of times we traverse its cycle. Therefore, knowing how to count simple paths, the problem of counting all paths becomes a matter of cycle detection. Let us recall the proof of Proposition 3.2.7 and look at the (only) path π leaving s in G'.

If G contains a non-simple path from s to t, the first vertex that is visited twice on that path must lie either on π or "after" (and thus be reachable from) the divergence point x. In the latter case, the number of paths from one of the successors of x to t will be infinite, in which case the call to $\mathcal{C}$-**Count**k**AF** will return "k^+" and the whole procedure will correctly answer "$(k+1)^+$". Thus we only need to detect a situation in which some vertex $y \in \pi$ lies on a cycle, or equivalently, y is reachable from some successor z of y. As we can deterministically enumerate over all vertices on π and all successors of each of them, it remains to show how we can answer the question of y being reachable from z.

Let us then introduce an additional modification of our input graph, namely the change of source and target vertices. It is obvious that it can be done deterministically in **QFunc**(n). Moreover, as we are guaranteed that the new source z is reachable from the old source s, and likewise, the old target t is reachable from the new target y, we can see that the "interesting" paths in the new graph form a subset of those in the old one. From this it follows that the new graph belongs to $\mathcal{C}$, and thus we can simply use $\mathcal{C}$-**Count1AF** to check whether y is reachable from z. ■

Corollary 3.2.11 *For any classes* $\mathcal{C} \in \mathbb{REM}$, $\mathcal{D} \in \mathbb{UNI}$, *and constant* $k \geq 1$,

$$\mathcal{C}\textbf{-Reach} \in \mathcal{D}\textbf{-QFunc}(\log(n)) \iff \mathcal{C}\textbf{-Count}k\textbf{AF} \in \mathcal{D}\textbf{-QFunc}(\log(n)).$$

Finally, when our guarantees apply to all vertices reachable from the source, we are free to use our algorithms with an arbitrary vertex as the target. This allows us to conclude:

Corollary 3.2.12 *For any class* $\mathcal{D} \in \mathbb{UNI}$, *bound* p, *and constant* k,

$$p\textbf{AA-Reach} \in \mathcal{D}\textbf{-QFunc}(\log(n)) \Rightarrow p\textbf{AA-Count}k\textbf{AA} \in \mathcal{D}\textbf{-QFunc}(\log(n)),$$
$$p\textbf{SA-Reach} \in \mathcal{D}\textbf{-QFunc}(\log(n)) \Rightarrow p\textbf{SA-Count}k\textbf{AA} \in \mathcal{D}\textbf{-QFunc}(\log(n)).$$

3.3 Inductive Counting

In this section we present algorithms based on the technique called "inductive counting." They all look at the vertices of the input graph reachable from the source s in concentric "layers," with layer k (denoted by L_k) consisting of those whose distance (the length of the shortest path) from s is *at most* k. Then they analyze the layers one by one, using some information computed or verified for one layer to help analyze the next.

Let us start with the breakthrough due to Immerman and Szelepcsényi (see [13, 36]). Let us denote the number of vertices in L_k by C_k. It turns out that it is possible to calculate this number within $\mathbf{QFunc}(\log(n))$. We do this by induction on k, with $C_0 = 1$ (s is the only vertex with distance 0 from itself), and the calculation of C_{k+1} from C_k carried out by the following procedure (**guess** denotes making a nondeterministic choice):

Algorithm 3.3.1 (Inductive Counting)

1. *set* $C_{k+1} := 1$
2. *for every* $v \in V - \{s\}$*:*
3. *set* $C'_k := 0$, $F :=$ *false*
4. *for every* $u \in V$*:*
5. **guess** *whether* $u \in L_k$, *if not—move to the next* u
6. **guess** *a path from* s *to* u *of length* $\leq k$ *(or* **fail***)*
7. *set* $C'_k := C'_k + 1$
8. *if* $\langle u, v \rangle \in E$, *set* $F :=$ *true*
9. *if* $C'_k < C_k$, **fail**
10. *if* $F =$ *true, set* $C_{k+1} := C_{k+1} + 1$

It is not difficult to convince oneself that on all of the nondeterministic branches that have not failed, the value of C_{k+1} has been computed correctly. First of all, every vertex v contributes to the count only if there was a vertex $u \in L_k$, and an edge $\langle u, v \rangle$. Therefore, the only way the computation may go wrong is for some u being incorrectly guessed *not* to be in L_{k+1}. For that to happen, there must exist a vertex $u \in L_k$ (and the edge $\langle u, v \rangle$), which we have wrongly guessed (in step 5) not to be there. But then the checksum count C'_k would be smaller than the true count C_k, and the computation branch would fail in step 9.

Clearly, the above can be used to create a total algorithm for **Reach** (it is enough to compare the counts with the target vertex present and removed). Moreover, its correctness does not depend on the shape of the input graph.

Recalling that $\mathcal{C}$-α denotes the problem α restricted to graphs in class $\mathcal{C}$, we can conclude:

Corollary 3.3.2 **N-Reach** $\in$ **N-QFunc**$(\log(n))$.

Analyzing the algorithm a little closer, we can show more:

Proposition 3.3.3

$$\mathbf{1AA\text{-}Reach} \in \mathbf{1AF\text{-}QFunc}(\log(n)),$$
$$\mathbf{1SA\text{-}Reach} \in \mathbf{1SF\text{-}QFunc}(\log(n)),$$
$$\mathbf{1MA\text{-}Reach} \in \mathbf{1MF\text{-}QFunc}(\log(n)).$$

Proof On how many nondeterministic branches can Algorithm 3.3.1 succeed? The guesses made in step 5 are of no consequence here, as there is only one way of guessing that will not lead to a failure later on. The only ambiguity is therefore introduced in step 6. But the guesses made there correspond to the paths in the input graph, and therefore any uniqueness promises about them yield analogous unambiguity properties of the accepting paths. ■

Algorithm 3.3.1 guesses paths between the same pairs of vertices over and over again, and thus the result does not immediately extend to higher path counts. If however, we know a limit on these counts, we can modify the algorithm (following Buntrock, Hemachandra and Siefkes, see [7]) to guess and verify *all* the paths to every reachable vertex. How can we do this? First, assume that we have guessed the number p of distinct paths (of length at most l) between vertices u and v. Then we can use the following procedure to verify the existence of at least these many paths:

Algorithm 3.3.4

guesspaths(G, u, v, p, l)*:*
1. *if* $p = 1$, **guess** *a path from* u *to* v *of length* $\leq l$ *and return*
2. **guess** w*, the first divergence point of paths from* u *to* v
3. **guess** *a path from* u *to* w *(or* **fail***), let* $l' < l$ *be its length*
4. *let* w' *and* w'' *be the two successors of* w
5. **guess** *the number* p' $(0 < p' < p)$ *of distinct paths from* w' *to* v
6. *let* $p'' := p - p'$, $l'' := l - l' - 1$
7. **guesspaths**(G, w', v, p', l'')
8. **guesspaths**(G, w'', v, p'', l'')

Let us first see what happens if the procedure has been supplied with too high a value of p. If $p = 1$, it means that there is no path from u to v and we will fail in step 1. For $p > 1$ it is easy to see that even if the divergence point w, and the path

to it, are guessed correctly, at least one of the numbers p' or p'' will exceed the actual number of paths. Thus by a simple inductive reasoning, the procedure will fail.

What happens when the value of p is correct? If all the guesses are made correctly, the algorithm returns successfully. If the divergence point w is wrong, at least one of its successors will fail to have enough paths to v and the procedure will fail. The same is bound to happen if the number p' is guessed wrongly, as then either p' or p'' will be larger than the actual number of paths. It is then clear that with the correct p on input, the algorithm will succeed on *exactly one* computation branch.

The situation of the p provided being too low is a bit less fortunate, as then the procedure might succeed on multiple computation branches (effectively guessing any p distinct paths from u to v). But if we process the graph layer by layer, we can keep track of the *collective* number of paths (T_k, with $T_0 = 1$) to all vertices in L_k, and use it to cut off these "unfortunate" branches:

Algorithm 3.3.5

1. *set* $T_{k+1} := 1$
2. *for every* $v \in V - \{s\}$*:*
3. *set* $T'_k := 0$, $r := 0$
4. *for every* $u \in V$*:*
5. **guess** *the number* $p \geq 0$ *of distinct paths from* s *to* u *of length* $\leq k$
6. **guesspaths**(G, s, u, p, k)
7. *set* $T'_k := T'_k + p$
8. *if* $\langle u, v \rangle \in E$, *set* $r := r + p$
9. *if* $T'_k < T_k$, **fail**
10. *set* $T_{k+1} := T_{k+1} + r$

Using arguments analogous to the discussion following Algorithm 3.3.1, one can easily show that the value of T_{k+1} will be correctly computed on exactly one nondeterministic branch, and that all other branches will fail.

What are the space requirements of this procedure? Being almost identical to Algorithm 3.3.1, it uses the same amount of space, plus any calls to **guesspaths()**. The latter, not counting the recursive calls, uses logarithmic space too. One of the calls is a tail recursion, and thus can easily be eliminated. Moreover, if we modify the procedure to always handle the larger of the values of p' and p'' using tail recursion, the depth of the stack can be bounded by $O(\log(p))$. With each stack record consisting of a vertex (w' or w''), a path count (p' or p''), and a path length ($l'' \in n^{O(1)}$), the total space used by **guesspaths()** can be bounded by $O((\log(n) + \log(p)) \log(p)) = O(\log(np) \log(p))$.

As it is obvious that Algorithm 3.3.5 can be used to answer the question of reachability, we obtain the following:

Proposition 3.3.6 p**AA-Reach** $\in 1$**AF-QFunc**$(\log(np)\log(p))$.

Corollary 3.3.7 *For constants k and l,*

$$k\textbf{AA-Reach},$$
$$k\textbf{AA-Count}l\textbf{SF}, k\textbf{AA-Count}l\textbf{AF},$$
$$k\textbf{AA-Count}l\textbf{SA}, k\textbf{AA-Count}l\textbf{AA} \in 1\textbf{AF-QFunc}(\log(n)).$$

Moreover, taking $l = k$ and noting that if there are at most k paths of type $\mathcal{X}$, then **Count**$k\mathcal{X}$ becomes **Count**$\mathcal{X}$, we can obtain the following:

Corollary 3.3.8

$$k\textbf{AA-CountAF}, k\textbf{AA-CountAA} \in 1\textbf{AF-QFunc}(\log(n)).$$

Algorithms 3.3.1 and 3.3.5 consider *all* paths to every vertex. It is however possible to adapt them to consider only the *minimal length* paths. Following Allender and Reinhardt (see [30]), we can extend the former to count not only the *number* of vertices in L_k (C_k), but also the *sum of lengths of the shortest paths* to those vertices (denoted Σ_k). Obviously, $\Sigma_0 = 0$. The modified version of the algorithm looks as follows:

Algorithm 3.3.9 (Double Counting)

1. *set* $C_{k+1} := C_k$, $\Sigma_{k+1} := \Sigma_k$
2. *for every* $v \in V - \{s\}$*:*
3. *set* $C'_k := 0$, $\Sigma'_k := 0$, $F :=$ *false*
4. **guess** $d \leq k+2$ *(minimal d for which $v \in L_d$, with $k+2$ denoting "more than $k+1$")*
5. *for every* $u \in V$*:*
6. **guess** *whether* $u \in L_k$*, if not—move to the next* u
7. **guess** $l \leq k$ *(minimal for which* $u \in L_l$*)*
8. **guess** *a path from* s *to* u *of length* l *(or* **fail***)*
9. *set* $C'_k := C'_k + 1$, $\Sigma'_k := \Sigma'_k + l$
10. *if* $\langle u, v\rangle \in E$*, then*
11. *if* $l + 1 < d$, **fail**
12. *if* $l + 1 = d$, *set* $F :=$ *true*
13. *if* $C'_k < C_k$ *or* $\Sigma'_k > \Sigma_k$, **fail**
14. *if* $F =$ *false and* $d \leq k+1$, **fail**
15. *if* $d = k+1$, *set* $C_{k+1} := C_{k+1} + 1$ *and* $\Sigma_{k+1} := \Sigma_{k+1} + d$

If all the guesses are made right, the algorithm computes C_{k+1} and Σ_{k+1} correctly. What can then go wrong? If the value of d (step 4) is too small, there will be no valid predecessor u for v, and the algorithm will fail in step 14. If it is too large, a predecessor closer to s will be found and the branch will fail in step 11. If for some vertex u the value of l (step 7) is too small, no path can be found and failure will occur in step 8. If it is too large, then the checksum value of Σ'_k will be larger than the actual value Σ_k, which will force us to fail in step 13.

Knowing Σ_k in addition to C_k does not seem to have any interesting consequences. Note however, that if there is a *unique* shortest path from s to every reachable vertex, the algorithm will compute the correct value on *exactly one* computation branch (as all the nondeterministic choices will have exactly one "valid" value). From here, we can immediately conclude the following:

Corollary 3.3.10 **1MA-Reach** $\in$ 1**AF-QFunc**($\log(n)$).

If the graph on the input to Algorithm 3.3.9 does not belong to **1MA**, there is a vertex reachable from s by two distinct shortest paths. Take v to be such a vertex closest to s. It means that its predecessors on those paths must be distinct, and the statement in line 12 of the algorithm would set the flag F to true more than once for this vertex v. Simply testing for that situation allows us to discover the issue and, moreover, if the loops in steps 2 and 5 evaluate the vertices in some fixed order, the issue will be discovered on exactly one computation branch.

Corollary 3.3.11 **Test1MA** $\in$ 1**AF-QFunc**($\log(n)$).

As it turns out, the same "double counting" technique can be applied to Algorithm 3.3.5. First, we can make sure that **guesspaths()** considers only paths of length *exactly* l (instead of up to l). Then, making T_k denote the collective number of shortest paths to all vertices in L_k, we can compute T_{k+1} and Σ_{k+1} from T_k and Σ_k as follows:

Algorithm 3.3.12

1. *set* $T_{k+1} := T_k$, $\Sigma_{k+1} := \Sigma_k$
2. *for every* $v \in V - \{s\}$*:*
3. *set* $T'_k := 0$, $\Sigma'_k := 0$, $r := 0$
4. **guess** $d \leq k+2$ *(minimal* d *for which* $v \in L_d$*,*
with $k+2$ *denoting "more than* $k+1$*")*
5. *for every* $u \in V$*:*
6. **guess** *whether* $u \in L_k$*, if not—move to the next* u

7. **guess** $l \leq k$ *(minimal for which* $u \in L_l$*)*
8. **guess** $p \geq 1$ *(the number of distinct paths of length* l *from* s *to* u*)*
9. **guesspaths**(G, s, u, p, l)
10. *set* $T'_k := T'_k + p$, $\Sigma'_k := \Sigma'_k + l$
11. *if* $\langle u, v \rangle \in E$, *then*
12. *if* $l + 1 < d$, **fail**
13. *if* $l + 1 = d$, *set* $r := r + p$
14. *if* $T'_k < T_k$ *or* $\Sigma'_k > \Sigma_k$, **fail**
15. *if* $r = 0$ *and* $d \leq k + 1$, **fail**
16. *if* $d = k + 1$, *set* $T_{k+1} := T_{k+1} + r$ *and* $\Sigma_{k+1} := \Sigma_{k+1} + dr$

Again, a discussion similar to that following Algorithm 3.3.9 allows us to conclude that our procedure finishes successfully on exactly one computation branch, and that it can be used in a similar manner for both reachability and testing problems. Furthermore, its space requirements are precisely those of **guesspaths()**. Thus we can conclude the following:

Proposition 3.3.13

$$p\textbf{MA-Reach}, \textbf{Test}p\textbf{MA} \in 1\textbf{AF-QFunc}(\log(np)\log(p))).$$

Corollary 3.3.14 *For a constant* k*,*

$$k\textbf{MA-Reach}, \textbf{Test}k\textbf{MA} \in 1\textbf{AF-QFunc}(\log(n)).$$

3.4 Graph Traversal

All algorithms from the previous section make invalid computation paths fail based on "collective" quantities, and thus it is not known whether the bound of $1\textbf{AF}$ can be tightened to $1\textbf{AA}$. However, slightly strengthening the assumption allows us to use a different algorithm, due to Lange (see [20]). If L_k is promised to be a tree, the following procedure can be used to traverse it (the two possible successors of a vertex can be seen as its left and right children in the tree):

Algorithm 3.4.1

traverse(G, k)*:*
1. let z *be the rightmost leaf of* L_k
2. set $u := s, l := 0$
3. while $u \neq z$*:*

4. *if $l < k$ and u is not a leaf, then*
5. *$u :=$ left child of u, $l := l + 1$*
6. *else*
7. **guess** *$l' < l$ and a path of length l' from s, let v be its end*
8. *set $w :=$ left child of v*
9. *for every $i \in \{l' + 2, l' + 3, \ldots, l\}$:*
10. *$w :=$ right child of w*
11. *if $w \neq u$,* **fail**
12. *set $u :=$ right child of v, $l := l' + 1$*

It is a simple observation that (under the mentioned assumption) this procedure performs a depth-first traversal: for the internal nodes, the successor is chosen deterministically, while for the leaves the successor's parent v (and in a tree there can be only one) is guessed in step 7 and verified in steps 8-11. Moreover, knowing v unambiguously determines the guesses made in step 7, and hence for every v there is at most one computation branch leading to failure in step 11.

Now, if the input graph is in **1AA**, we can simply use Algorithm 3.4.1 to visit all reachable vertices. We can also employ it to check whether this condition holds (i.e., solve **Test1AA**). To see this, consider the following procedure:

Algorithm 3.4.2

1. *for every $k \in \{0, 1, \ldots, n-1\}$:*
2. *for every $v \in V$:*
3. *set $c := 0$*
4. *for every u visited by* **traverse**(G, k)*:*
5. *if $\langle u, v \rangle \in E$, set $c := c + 1$*
6. *if $c > 1$,* **answer "no"**
7. **answer "yes"**

If $G \in$ **1AA**, every L_k is a tree and every reachable v has exactly one valid predecessor u. Therefore the procedure answers "yes" (at exactly one computation path). If $G \notin$ **1AA**, there is a smallest k for which L_{k+1} is not a tree, and thus a vertex v with at least two valid predecessors within distance k. Then the algorithm answers "no" (also on exactly one branch). The procedure is deterministic except for the calls to **traverse**, and the latter is only called for those L_k's that are trees. Therefore we can conclude the following:

Corollary 3.4.3 **1AA-Reach**, **Test1AA** $\in$ **1AA-QFunc**$(\log(n))$.

It seems that it should be possible to apply the technique of "multiple path guessing" (Algorithm 3.3.4) to the procedures from this section, and thus extend the results from $1\mathbf{AA}$ to $k\mathbf{AA}$ for an arbitrary constant k. Any such application is a matter for future research.

3.5 Future Work

Our work provides a convenient formal framework for analyzing unambiguously-computable functions. In particular, it seems that the use of nondeterministic reductions should allow one to achieve results stronger than those presented in this chapter. Furthermore, there are at least a few possible ways in which our framework could be extended. Thus we would like to point out at least the following directions for future work:

1. Try to improve the results outlined in Section 3.1 from constant to polynomial bounds on path count. This will require a way of unambiguously characterizing a *set* of "pivot points," together with the means of verifying whether a given set meets this characterization.

2. Apply the technique of Algorithm 3.3.4 (unambiguously guessing multiple paths at the same time) to the graph traversal algorithm from Section 3.4. A success here would show the appropriate variants of reachability to be complete for classes $k\mathbf{AAQFunc}(\log(n))$ (for a constant k), and together with results from Section 3.1, possibly prove the collapse of these classes.

3. Incorporate strong unambiguity (limiting the number of paths between *any* two configurations, not necessarily reachable from the initial one), and possibly other restrictions, into the framework. The definitions and theorems from Chapter 2 could be easily adapted. However, it seems that it is difficult to construct a Quering Turing Machine algorithm that would make use of nondeterminism, and yet remain strongly unambiguous—such an algorithm would need to deal with unreachable situations, which in the case of Quering Turing Machines include possibly inconsistent answers from the input oracle.

4. Enrich the framework by a means of limiting the number of oracle queries allowed. On one hand it would allow a stronger version of Proposition 2.4.4—constantly or logarithmically many queries to a constant-ambiguity oracle yield a constant- or polynomial-ambiguity result, respectively. On the other hand, it is not clear whether at most logarithmically many queries to the input oracle

can be of any advantage (in particular, such a situation does not even allow the machine to read the whole input).

5. Extend the models to allow for unbounded-space pushdown storage. This would capture the existing variants of unambiguity for auxiliary pushdown automata, and could possibly yield results analogous to those for "standard" Turing Machines.

6. Consider different space bounds than just the logarithmic. In particular, the linear bound deserves some attention, as the question whether **DSPACE**(n) equals **NSPACE**(n) (the class of context-sensitive languages) is of great importance. On the other hand, we do not envision our framework being useful for lower space bounds, as it has been shown in [1] that reducibility below logarithmic space can be achieved by a two-way deterministic finite automata.

Appendix A

A Modular Approach

So far, we have not been able to provide a proof that **UL** = **NL**, but in this chapter we outline a promising approach. As this line of attack does not make use of the machinery developed in the core chapters of the thesis, and therefore we have decided to place it in the Appendix.

Let a **weight function** be a function assigning positive integers ("weights") to graph edges. It can also be seen as a graph transformation that replaces every edge with a sequence of consecutive edges (the length of that sequence being the weight assigned to the original edge).

As proposed in [30], Algorithm 3.3.9 (unambiguously solving reachability on 1**MA**-graphs) can be used together with a specialized version of the Isolation Lemma of Mulmuley et al. (see [25]) to obtain a collapse of **UL** and **NL** in the nonuniform setting:

Proposition A.1 *For all integers n large enough, there exists a sequence of n^2 weight functions $\langle w_i \rangle_i$ with ranges in $[1, 4n^3]$, such that for every graph G on n vertices, there exists an i such that $w_i(G) \in 1\mathbf{MA}$.*

Proof The argument for the existence of $\langle w_i \rangle_i$ is probabilistic. Let us choose our sequence of functions by assigning the weight to every edge independently and uniformly at random. Let us bound the probability that all functions obtained in such

way will be "bad" for at least one graph of size n:

$$\begin{aligned}
\mathbf{Pr}[\exists_{G,\#G=n}\forall_i w_i(G) \notin \mathbf{1MA}] \leq \\
\#\{G|\#G=n\}\ \mathbf{Pr}[\forall_i w_i(G) \notin \mathbf{1MA}] \leq \\
2^{n^2}(\mathbf{Pr}[\exists_{v\in V_G,\pi_1\neq\pi_2:s\rightsquigarrow v}\pi_1 \text{ and } \pi_2 \text{ are shortest under } w_i])^{n^2} \leq \\
2^{n^2}(\mathbf{Pr}[\exists_{v\in V_G,e\in E_G} e \text{ is on one but not another shortest path from } s \text{ to } v])^{n^2} \leq \\
2^{n^2}(\#(V_G\times E_G)\ \mathbf{Pr}[e \text{ is on one but not another shortest path from } s \text{ to } v])^{n^2} = \\
2^{n^2}(n^3\mathbf{Pr}[e \text{ is on one but not another shortest path from } s \text{ to } v])^{n^2} \leq (*) \\
2^{n^2}\left(n^3\frac{1}{4n^3}\right)^{n^2} = \\
2^{n^2}\left(\frac{1}{4}\right)^{n^2} = \frac{1}{2^{n^2}},
\end{aligned}$$

where $(*)$ holds because once we fix the weights of all edges except e, there will be at most one possible weight for e that makes it possible.

As the probability of all functions being "bad" is smaller than 1 (and in fact approaches zero quite rapidly as n grows), there must exist a "good" weight function. ■

The above argument is non-constructive. However, the family of the functions whose existence is shown does not depend on the graph and only on its size n. Therefore the information can be used as the advice in non-uniform computation model. Now given a specific graph, we can iterate over individual weight functions, using Algorithm 3.3.9 to discover whether $w_i(G) \in \mathbf{1MA}$, and to solve **Reach** if it does.

Corollary A.2 $\mathbf{NL/poly} = \mathbf{UL/poly}$.

(For a proof see [30].)

With almost all (probabilistically speaking) weight functions being able to disambiguate log-space computation, it seems reasonable to expect that one should be able to compute at least some of them in **FL**. We present a candidate for such a family of functions and provide some evidence (though no proof so far) that it indeed has the desired properties. It is our hope that further analysis of this approach may yield some fruit.

First, let us note that to show $\mathbf{NL} = \mathbf{UL}$, we do not have to disambiguate all possible graphs. We can first modify the configuration graph by adding a step counter. Thus the graph can be transformed into a *layered dag* (with vertices partitioned into layers and all edges going between consecutive layers only) while preserving the

property that the accepting configuration (which we can force to be the last node in the graph) is reachable from the initial one (again, easily made the first node) iff the **NL** machine being analyzed accepts its input.

If we identify vertices with their numbers and assign weight $2 * 3^v$ (simply using 2^v would work here as well, but the base of 3 and the multiplier are used in later arguments) to any edge $\langle u, v \rangle$, the total weight of any path π originating from the source s will be equal to

$$w(\pi) = -2 * 3^s + 2 * \sum_{v \in \pi} 3^v.$$

Now for any two paths π_1, π_2 of equal weight we have

$$\begin{gathered} w(\pi_1) = w(\pi_2) \Rightarrow \\ \sum_{v \in \pi_1} 3^v = \sum_{v \in \pi_2} 3^v \Rightarrow \\ v \in \pi_1 \iff v \in \pi_2, \end{gathered}$$

which in a dag yields $\pi_1 = \pi_2$.

Of course, this weight function w has an exponential range, so it cannot be used directly. We will therefore force it into a polynomial range by choosing a suitable constant t and taking all the weights modulo i (thus creating a different function w_i) for every $i \in \{2, \ldots, n^t\}$:

$$w_i(u, v) := (2 * 3^v) \mod i.$$

Let us assume that this function family does not work, i.e., that there exists an infinite family of graphs $\{G_n\}$ ($\#V_{G_n} = n$) such that every weight function in our family fails to disambiguate it:

$$\forall_n \forall_{i \in \{2,\ldots,n^t\}} \exists_{u,v,\pi_1,\pi_2} \pi_1 \text{ and } \pi_2 \text{ are minimum-weight from } u \text{ to } v \text{ in } w_i(G_n).$$

Furthermore, without loss of generality, we can restrict the paths π_1 and π_2 not to share any vertices except u and v. To achieve this, simply make u the first point at which the paths diverge, and v—the first point at which they meet again; as subpaths of paths of minimum weight, the restricted paths will have to be minimum-weight as well.

Let us take any $G \in \{G_n\}$. Noting that in a layered dag any two paths π_1 and π_2 between the same pair of vertices must have the same number of vertices, we can

now construct a new weighted graph H which will "capture" all such possible pairs of paths:

$$V_H := V_G \times V_G,$$
$$E_H := \{\langle\langle u, x\rangle, \langle v, y\rangle\rangle | \langle u, v\rangle, \langle x, y\rangle \in E_G\}.$$

As mentioned, without loss of generality, we can focus on pairs of paths that do not share any vertices (except the common source and target)—we will call such paths in H **proper**. We can now define the weight function on H to calculate the difference of weights of the edges (and thus the paths) from G:

$$w(\langle\langle u, x\rangle, \langle v, y\rangle\rangle) := w(\langle u, x\rangle) - w(\langle v, y\rangle) = 2 * 3^x - 2 * 3^y.$$

It is easy to show that this weight function w is injective on proper paths. Consider the "heaviest" vertex n. The weights of all other vertices can add up (by absolute value) to at most

$$2\sum_{i=0}^{n-1} 3^i = 2\frac{3^n - 1}{3 - 1} < 3^n,$$

and therefore for a proper path π, knowing within which of the ranges $[-3^{n+1}, -3^n)$, $[-3^n, 3^n]$, or $(3^n, 3^{n+1}]$ the value of w falls, determines whether n is on any (and on which) of the "component paths" of π. With that known, one can remove the weight of n from the total and repeat the process to obtain the exact specification of π.

Let us now return to the modular restrictions of w, and extend them naturally to H. If a weight function w_i fails to disambiguate G, we must have a proper path in H whose weight r will be a multiple of i. Among $\Omega(n^t)$ values of i, there are $\Omega(n^{t-1})$ primes. As the values of w fall within the range $[-3^{n+1}, 3^{n+1}]$, a single weight r can be "used" for at most $O(n)$ of the prime moduli—the only value having more prime factors is 0, which corresponds to an empty path in H. Therefore we must have at least $\Omega(n^{t-2})$ distinct values of r and, as w is injective on proper paths, at least $\Omega(n^{t-2})$ distinct proper paths in H.

We conjecture that, given the structure of H, such situation is not possible. Intuitively, if the proper paths in question were short, there could not be many of them. On the other hand, if they were long, many of them would have to share many vertices, which (at least probabilistically) seems to yield a situation in which some edges are at the same time required to be present and absent from G. If this conjecture can be proven, based on the above discussion it would immediately give the long-expected result of **UL** = **NL**.

Bibliography

[1] M. Agrawal, "For completeness, sublogarithmic space is no space," *Information Processing Letters*, vol. 82, no. 6, pp. 321–325, 2002.

[2] E. Allender and K. Lange, "StUSPACE($\log n$) is Contained in DSPACE($\log^2 n / \log\log n$)," in *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 3, 1996.

[3] E. Allender, K. Reinhardt, and S. Zhou, "Isolation, Matching, and Counting: Uniform and Nonuniform Upper Bounds," *Journal of Computer and System Sciences*, vol. 59, no. 2, pp. 164–181, 1999.

[4] C. Àlvarez and B. Jenner, "A Very Hard Log Space Counting Class," in 5^{th} *Annual Conference on Structure in Complexity Theory*, pp. 154–168, IEEE Computer Society Press, 1990.

[5] C. Bourke, R. Tewari, and N. Vinodchandran, "Directed planar reachability is in unambiguous logspace," in *IEEE Conference on Computational Complexity*, pp. 217–221, IEEE Computer Society Press, 2007.

[6] G. Buntrock, C. Damm, U. Hertrampf, and C. Meinel, "Structure and importance of logspace-MOD class," *Theory of Computing Systems*, vol. 25, no. 3, pp. 223–237, 1992.

[7] G. Buntrock, L. Hemachandra, and D. Siefkes, "Using Inductive Counting to Simulate Nondeterministic Computation," *Information and Computation*, vol. 102, no. 1, pp. 102–117, 1993.

[8] G. Buntrock, B. Jenner, K. Lange, and P. Rossmanith, "Unambiguity and Fewness for Logarithmic Space," in 8^{th} *International Conference on Fundamentals of Computation Theory*, vol. 529 of *Lecture Notes in Computer Science*, pp. 168–179, Springer, 1991.

[9] A. Chandra, D. Kozen, and L. Stockmeyer, "Alternation," *Journal of the ACM*, vol. 28, no. 1, pp. 114–133, 1981.

[10] S. A. Cook, "Characterizations of Pushdown Machines in Terms of Time-Bounded Computers," *Journal of the ACM*, vol. 18, no. 1, pp. 4–18, 1971.

[11] O. Goldreich, *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

[12] J. Hopcroft and J. Ullman, "Some Results on Tape-Bounded Turing Machines," *Journal of the ACM*, vol. 16, no. 1, pp. 168–177, 1969.

[13] N. Immerman, "Nondeterministic space is closed under complementation," in *3rd Annual Conference on Structure in Complexity Theory*, pp. 112–115, IEEE Computer Society Press, 1988.

[14] B. Jenner and B. Kirsig, *Alternierung und Logarithmischer Platz*. Dissertation, Universität Hamburg, 1989.

[15] N. Jones, "Space-bounded reducibility among combinatorial problems," *Journal of Computer and System Sciences*, vol. 11, no. 1, pp. 68–85, 1975.

[16] R. Kannan, "Alternation and the power of nondeterminism," in *15th Annual ACM Symposium on Theory of Computing*, pp. 344–346, ACM, 1983.

[17] R. Karp, "Reducibility among combinatorial problems," *Complexity of Computer Computations*, vol. 43, pp. 85–103, 1972.

[18] R. Ladner and N. Lynch, "Relativization of Questions About Log Space Computability," *Theory of Computing Systems*, vol. 10, no. 1, pp. 19–32, 1976.

[19] K. Lange, "Nondeterministic Logspace Reductions," in *11th Symposium on Mathematical Foundations of Computer Science*, vol. 176 of *Lecture Notes in Computer Science*, pp. 378–388, Springer, 9 1984.

[20] K. Lange, "An Unambiguous Class Possessing a Complete Set," in *14th Annual Symposium on Theoretical Aspects of Computer Science*, vol. 1200 of *Lecture Notes in Computer Science*, pp. 339–350, Springer, 1997.

[21] H. Lewis and C. Papadimitriou, "Symmetric Space-Bounded Computation," *Theoretical Computer Science*, vol. 19, pp. 161–187, 1982.

[22] M. Liśkiewicz and R. Reischuk, "The Sublogarithmic Alternating Space World," *SIAM Journal on Computing*, vol. 25, p. 828, 1996.

[23] B. Litow and I. Sudborough, "On non-erasing oracle tapes in space bounded reducibility," *SIGACT News*, vol. 10, no. 2, pp. 53–57, 1978.

[24] N. Lynch, "Log Space Machines with Multiple Oracle Tapes," *Theoretical Computer Science*, vol. 6, pp. 25–39, 1978.

[25] K. Mulmuley, U. Vazirani, and V. Vazirani, "Matching is as easy as matrix inversion," *Combinatorica*, vol. 7, no. 1, pp. 105–113, 1987.

[26] R. Niedermeier and P. Rossmanith, "Unambiguous auxiliary pushdown automata and semi-unbounded fan-in circuits," *Information and computation*, vol. 118, no. 2, pp. 227–245, 1995.

[27] C. Papadimitriou, *Computational complexity*. Addison-Wesley, 1994.

[28] O. Reingold, "Undirected ST-connectivity in log-space," in *37^{th} Annual ACM Symposium on Theory of Computing*, pp. 376–385, ACM, 2005.

[29] O. Reingold, S. Vadhan, and A. Wigderson, "Entropy waves, the zig-zag graph product, and new constant-degree expanders," *Annals of Mathematics*, vol. 155, no. 1, pp. 157–187, 2002.

[30] K. Reinhardt and E. Allender, "Making Nondeterminism Unambiguous," in *38^{th} Annual Symposium on Foundations of Computer Science*, pp. 244–253, IEEE Computer Society Press, 1997.

[31] W. Ruzzo, J. Simon, and M. Tompa, "Space-bounded hierarchies and probabilistic computations," in *14^{th} Annual ACM Symposium on Theory of Computing*, pp. 215–223, ACM, 1982.

[32] W. Rytter, "On the recognition of context-free languages," in *5^{th} International Conference on Fundamentals of Computation Theory*, vol. 208 of *Lecture Notes in Computer Science*, pp. 318–325, Springer, 1984.

[33] J. Savitch, "Relationship between nondeterministic and deterministic tape complexities," *Journal of Computer and System Sciences*, vol. 4, no. 2, pp. 177–192, 1970.

[34] R. Stearns, J. Hartmanis, and P. Lewis, "Hierarchies of memory limited computations," in *6^{th} Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, pp. 179–190, IEEE Computer Society Press, 1965.

[35] I. Sudborough, "On the Tape Complexity of Deterministic Context-Free Languages," *Journal of the ACM (JACM)*, vol. 25, no. 3, pp. 405–414, 1978.

[36] R. Szelepcsényi, "The method of forced enumeration for nondeterministic automata," *Acta Informatica*, vol. 26, no. 3, pp. 279–284, 1988.

[37] A. Szepietowski, "If Deterministic and Nondeterministic Space Complexities are Equal for $\log\log n$ then they are also Equal for $\log n$," in *6th Annual Symposium on Theoretical Aspects of Computer Science*, vol. 349 of *Lecture Notes in Computer Science*, pp. 251–255, Springer, 1989.

[38] A. Szepietowski, *Turing Machines with Sublogarithmic Space*. Springer, 1994.

[39] L. Valiant, "Relative Complexity of Checking and Evaluating," *Information Processing Letters*, vol. 5, no. 1, pp. 20–23, 1976.