Unambiguous Functions in Logarithmic Space

Grzegorz Herman
Michael Soltys
Department of Computing and Software, McMaster University
email: `gherman@tcs.uj.edu.pl, soltys@mcmaster.ca`

November 21, 2011

Abstract

We investigate different variants of unambiguity in the context of computing multi-valued functions. We propose a modification to the standard computation models of Turing machines and configuration graphs, which allows for unambiguity-preserving composition. We define a notion of reductions (based on function composition), which allows nondeterminism but controls its level of ambiguity. In light of this framework we establish reductions between different variants of path counting problems. We obtain improvements of results related to inductive counting.

Keywords: Logarithmic Space, Nondeterminism, Reduction, Unambiguity

1 Introduction

The notion of nondeterminism is a fundamental notion in theoretical computer science. Numerous restrictions of nondeterminism have been introduced and studied. A well known restriction is that of *unambiguity*, in which the machine is not "aware" of the existence of an accepting path (and makes nondeterministic choices), but the path itself is required to be unique.

The unambiguous version of logarithmic space, called **UL**, has been explicitly considered for the first time in [1] and [2]. In the latter paper, **UL** variants allowing 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 were 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 [3], 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 [4]). In [8], Lange has shown a problem complete for **ReachUL**.

Finally, inductive counting has been used again by Reinhardt and Allender in [9] 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 [10] for details).

The contributions of the present work are threefold:

1. 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, as opposed to functions based on properties of computation trees of nondeterministic machines

(as, for example, the function classes **#L**, **GapL**, etc.; see [1]). This type of computation has been previously defined ([11]), but it was based on *deterministic* machines having access to an oracle for a language from a possibly nondeterministic class. Our model agrees with this approach when full power of nondeterminism is allowed, but has the advantage of being easily adaptable to classes of limited ambiguity.

2. We introduce the notion of nondeterministic, unambiguous reductions. In our model, oracles are used only as a tool of function composition, and not as sources of additional computational power. Therefore our reductions work in a similar way to many-one reductions, and are well suited for the purpose of comparing the ambiguity-complexity of functions. Within this framework, we analyze variants of the path-counting problem—in particular, by Propositions 27 and 28 we obtain the equivalence of counting up to any constant number of (arbitrary or simple) paths.

3. Finally, we take a closer look on the inductive counting technique of [12, 13], which allows us to combine the results of [3] and [9] into Algorithm 37: an unambiguous algorithm for reachability on graphs with restricted ambiguity of shortest paths.

This research has been completed during the first author's doctoral studies at McMaster University; see [5]. Also, a conference version of this paper was presented at the "Computability in Europe 2009," where it appeared in the abstract booklet; see [6].

2 Querying Turing machines

In the familiar definition of Turing machines there is a slight technical difficulty when composing computations with sub-linear space bounds. The difficulty is that the input and output tapes, which are not subject to the space restriction, become work-tapes of the new machine, and thus may violate the space bounds.

There exists an easy fix: instead of producing a (possibly long) output in its entirety, the machine computes just a single character at a requested index (if the index happens to point beyond the output, the answer will be a blank symbol). Moreover, we use the same approach to access the machine's input—the machine writes the index of the input character it is interested in on one of its tapes, and queries an oracle (by entering a special state).

Thus modified, the machine can be seen as rewriting **requests** about its **output** to (sequences of) **queries** about its **input**, and generating an **answer** (by entering a special answer state) based on the results of these queries. We call such a (nondeterministic) machine a **Querying Turing machine (QTM)**. Henceforth, we use the terms: input, output, request, query, and answer, to mean precisely the roles described here.

Definition 1 *A k-tape* **Querying Turing machine** *consists of:*

- *a finite* **alphabet**[1] Σ *(*$|\Sigma| \geq 2$*),*
- *a finite set of* **states** Γ*,*
- *an* **initial state** $q_{init} \in \Gamma$*,*

[1]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.

- *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** $\Delta \subseteq \Gamma \times \Sigma^k \times \Gamma \times \Sigma^k \times \{\leftarrow, -, \rightarrow\}^k$.

On the request-tape we provide the index (given as an integer in binary) of the output symbol that we wish to compute. The actual input-string to the function is provided as an oracle; when the machine wants to query the i-th symbol of the input-string (by writing i in binary on the query-tape, and entering the query state), the oracle provides this symbol. If the index happens to point beyond the string, the answer will be a blank symbol.

To see a QTM computation intuitively, imagine that on input $(j)_b$ (integer j in binary) provided to the QTM on its request-tape, the machine enters a possibly lengthy computation. In this computation it wishes to compute the j-th symbol of $f(w)$ on some input-string w. This input-string w is provided to the QTM in the form of an oracle. In order to compute the j-th symbol of $f(w)$, the machine queries the different bits of w; it does so by writing the index of the bit it wishes to see, for example i. So it writes $(i)_b$ on the query-tape, and by entering the query-state it prods the oracle to give it the i-th symbol of w. In general, all the bits of w are queried at some point, perhaps repeatedly, until the QTM is ready to output the j-th symbol of $f(w)$, which, for symbol a, it does by entering the corresponding response state $\hat{q}_a$. By providing the input as an oracle, and by computing a single symbol of the output, the machine's usage of space is circumscribed by the lenghts of the indices, making logarithmic bounds natural and easy to enforce.

We use the familiar notion of a configuration of a Turing machine: a configuration is a tuple consisting of the current state, together with the contents of all the tapes (note that this does include the request tape, but not the input, as the latter is only available via an oracle) and the positions of the heads. Any configuration with an answer state is called an answer-configuration.

Definition 2 *Given a QTM M with input oracle O, we say that the pair (M, O)* **yields** *a on request i if an answer-configuration with state $\hat{q}_a$ is reachable from the initial configuration. Note that the initial configuration specifies that M contains i on the query tape.*

Using the power of nondeterministic guesses 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. We make such **failures** explicitly distinct from any possible answer the machine might give.

We represent failure by "dead-end" configurations, that is, configurations that do not yield new ones. We also do not require answer states to be final—terminating or continuing the computation is seen as yet another aspect of nondeterminism (this will significantly simplify the formal definition of composition for computation graphs).

As we wish to do "composition" of machines, the input oracle will often be substituted with another QTM. Thus, we allow oracles to be multi-valued ("nondeterministic"), as well as to fail to give an answer, and we assume all the failures to be unrecoverable.

With the above in place, the only visible difference between a Querying 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** Querying Turing machines—those with a specific input "plugged in."

Clearly, a deterministic Querying Turing machine computes a well-defined function. However, when guesses are allowed, different branches might yield different answers, or fail altogether. One way of dealing with this is to define our machines to compute a relation, rather than a function. However, this gives rise to reductions which [7] has shown to be very strong: applying them to **NL** yields all of **NP**! Thus, instead we interpret "ambiguity" as "computing" multiple functions:

Definition 3 *Given an oracle O, we say that O* **consistently computes** *(we also say* **returns***) a string $X \in \Sigma^*$ if the following condition holds: for any query i, O either outputs $X[i]$ or fails. Here $X[i]$ denotes the i-th symbol of X, and we emphasize that "failing" is always a legal option.*

Definition 4 *Given a function $\phi : \alpha \to \beta$, we say that a QTM M is* **sound** *for ϕ if the following condition holds: if M is supplied with an input oracle that consistently returns $X \in \alpha$, then on any request i, M either yields $\phi(X)[i]$ or fails. Recall from definition 2 that "yields" means that one of the computation branches returns a.*

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 define:

Definition 5 *Given a function $\phi : \alpha \to \beta$, and given an $a \in \Sigma$, we say that a QTM M is* **a-total** *for ϕ if the following holds: if*

- *M is provided with an input oracle that consistenly returns $X \in \alpha$* without ever failing*, and*
- *given a request i to M such that the i-th character of $\phi(X)$ is a,*

then M yields a. Finally, we say that M is **total** *for ϕ if M is a-total for every $a \in \Sigma$.*

Suppose that we consider QTM deciders with output in $\Sigma = \{0, 1\}$. In order to relate this definition to the traditional nondeterministc deciders, we observe that if we replace failure with a "reject" answer, then a sound 1-total QTM corresponds to existential acceptance, i.e., there is a path that accepts. Analogously, if we replace failure with an "accept" answer, then a sound 1-total QTM corresponds to universal acceptance, i.e., all the paths accept.

In a QTM the input is not given explicitly, nor is the output produced explicitely. Thus we need to give a meaningful definition of space. Keeping in mind that whenever an oracle is queried on an index beyond its output (and only then), it answers with a blank symbol, we can define our space bounds as follows:

Definition 6 *The* **size (length)** *of an oracle is the smallest value of a query to which the oracle responds with a blank.*

Definition 7 *A QTM M* **operates in space** *$f(n)$ if whenever supplied with an input oracle of size n, it uses at most the first $f(n)$ cells on each one of its tapes, including the oracle tape.*

Note that the above definition measures the space with respect to the size of the oracle output; not with respect to the size of the query. The benefit of this measure is that it deals well with the atypical case where the request is very short compared to the input. On the other hand, algorithms where the space bound is given explicitly, i.e., it is computed before the machine is allowed to process the input, this is not an issue, as they can discover the size of their input and return a blank if the request is beyond the bound. See for example [18] where space-constructible bounds are discussed. Finally, just issuing oracle queries requires space logarithmic in the oracle size, and hence we will not consider sub-logarithmic space bounds.

The following definitions of complexity classes arise naturally from the above:

Definition 8 *Given a function $f(n)$, $f(n) \geq \log(n)$, the class* **QFunc**$(f(n))$ *consists of all functions that have sound and total QTMs operating in space $O(f(n))$.*

Definition 9 *Given a function $f(n)$, $f(n) \geq \log(n)$, the class* **QSpace**$(f(n))$ *(and* **co-QSpace**$(f(n))$*) consists of those languages that have characteristic functions with the following property: they are computed by QTMs that are sound, 1-total (respectively, 0-total) and operate in space $O(f(n))$.*

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

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

It has been shown in [11] that several definitions of **FNL** are equivalent; here we use the following:

Definition 10 *A function ϕ is in* **FNL** *if:*

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

In general, for a function $\phi \in \mathbf{FNSpace}(f(n))$ we insist that L_ϕ, the bit-graph of ϕ, is in **NSpace**$(f(n))$ and $|\phi(X)|$ is bounded by a polynomial. With that in mind we are ready to prove the following proposition.

Proposition 11 **QFunc**$(f(n))$ = **FNSpace**$(f(n))$.

Proof: We will only show that **QFunc**$(\log(n))$ = **FNL**—the result generalizes easily to larger space bounds.

Observe that the logarithmic space bound applies to all the tapes of the Querying Turing machine, and hence in particular to the size of the request tape. This in turn imposes a polynomial bound on the length of the output.

The **NL** machine for L_ϕ can be turned into a Querying Turing machine for ϕ by guessing the appropriate a, verifying the guess, and (if guessed correctly), answering "a". Turning a Querying 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 shows a close relationship between the function classes and the intersections of "existential" and "universal" language classes. This relationship can be captured as follows:

Corollary 12 *For every space bound f:*

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

Recall that L_ϕ is the bit-graph of ϕ; see definition 10.

Composing QTM computations will be done 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 13 *If M and N are QTMs sound for ϕ and ψ, respectively, then their composition, as described in the paragraph above, is sound for $\phi \circ \psi$. Moreover, if they are total, so is the composition.*

PROOF: The soundness of the composition can be seen directly from the definition 4. 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 QTMs:

Proposition 14 *For any pair of QTMs M and N, which run in space $f(n)$ and $g(n)$, respectively, their composition operates within space $O(f(2^{O(g(n))}))$. Recall our working assumption that $f(n), g(n) \geq \log(n)$.*

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. □

3 Querying Computation Graphs and Ambiguity

We extend the concept of configuration graphs to allow processing oracles queries.

Definition 15 *A* **Querying Computation Graph (QCG)** *$\langle V, E, S, c\rangle$ is a directed graph with a distinguished subset*[2] *of* **source vertices** *$S \subseteq V$, together with a coloring function $c : V \cup E \to \Sigma \cup \{\bot\}$, where $\bot$ denotes "no color," such that there is at most one vertex of every color, i.e., if it is the case that $c(u) = c(v) \neq \bot$ then $u = v$, but there may be many edges of a single color.*

Intuitively, we take the usual configuration graph of a nondeterministic computation on a given input, and we add colored vertices and edges to represent answer configurations and transitions dependent on oracle queries. The requirement of there being a unique answer-configuration for any specific answer can be easily fulfilled by making the machine erase the contents of all tapes before entering an answer state.

Oracles (and, equivalently, closed QTMs) do not issue any input queries. Thus their operation can be modeled with the following restriction of QCGs:

Definition 16 *A* **Closed Computation Graph (CCG)** *is a Querying 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 QCGs. Let us define it as follows:

Definition 17 *Given QCGs $G = \langle V_G, E_G, S_G, c_G\rangle$, $H = \langle V_H, E_H, S_H, c_H\rangle$, and a function f :*

[2]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.

$V_G \to S_H$, the **f-composition** *of G and H (denoted $G \circ_f H$) is a QCG $\langle V, E, S, c\rangle$ with:*

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

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 QTM as an input oracle of another is as follows. The function f represents the way of extracting the oracle query (and so the initial configuration of the inner machine) from the configuration of the outer machine (usually f is as simple as taking a segment of the configuration corresponding to the contents of the oracle tape, in which case we will omit it 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 are expected consequences of our definitions:

Observation 18 *The composition of two QCGs is a QCG. The composition of a QCG and a CCG is a CCG.*

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 19 *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:

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

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

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

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

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 QCGs to QTMs, 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 QTM, 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 QCG 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 20 *A CCG, corresponding to a QTM working in space $f(n)$ and supplied with an input of size n, has size $2^{O(f(n))}$.*

To capture the degree of ambiguity of a computation, we look at the shape of its CCGs:

Definition 21 *For a family $\mathcal{C}$ of CCGs, we say that a QTM M is a* **$\mathcal{C}$-machine** *iff, when supplied with any consistent input, its CCG belongs to $\mathcal{C}$.*

Definition 22 *The class* **$\mathcal{C}$-QFunc**$(f(n))$ *consists of all functions that have sound, total $\mathcal{C}$-machines operating in space $O(f(n))$. The classes* **$\mathcal{C}$-QSpace**$(f(n))$ *and* **co-$\mathcal{C}$-QSpace**$(f(n))$ *can be defined analogously.*

To be able to talk about classical deterministic and non-deterministic algorithms, we introduce two classes of CCGs: **D**—those of out-degree 1, and **N**—the class of all CCGs. Computational (un)ambiguity is enforced by limiting the number of distinct ways to reach (from a source vertex) a node in the CCG. 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),

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:
    - $\mathbf{A}$ = all nodes,
    - $\mathbf{F}$ = colored ("final") nodes only.

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

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

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

We denote ($\mathbb{REM}$ contains graph classes closed under edge removal):

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

4 Reductions

To talk about the relative complexity of different problems (functions), we would like to introduce a notion analogous to many-one log-space reductions, a notion that would allow nondeterminism but at the same time limit its level of ambiguity. The very nature of QTMs (being queried multiple times about different characters of their output) suggests employing some variant of Turing reductions, but these have been shown (see [14, 15, 16, 17]) to be very sensitive to the exact definition.

We have decided to take a path similar to that of [17]: we allow the original input to be transformed in a parametrized way, requiring that the transformation can be performed in one of the classes $\mathcal{C}$-$\mathbf{QFunc}(\log(n))$, and the "parameters" fit within the desired space bound. 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 23 *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 family of functions $\theta_i : \alpha \to \gamma$, and a function $\xi : \delta^* \to \beta$ such that:*

- *taking $\theta(X) := \langle\theta_i(X)\rangle_i$ we have $\theta \in \mathcal{C}$-$\mathbf{QFunc}(\log(n))$ (i.e., the functions θ_i can be "uniformly" computed in $\mathcal{C}$-$\mathbf{QFunc}(\log(n))$),*
- *$\xi \in \mathcal{D}$-$\mathbf{QFunc}(\log(n))$,*
- *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, extending Proposition 13, is the key to making use of unambiguous, nondeterministic reductions:

Lemma 24 *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 $\phi \circ \psi$, 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 Proposition 13. It remains to show how to obtain the desired (un)ambiguity properties. First, let us make the following simple observation about the composition:

Observation 25 *Every path in the f-composition of Querying 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 Querying 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 25. 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. □

Using the above lemma we can justify the definition of our notion of reduction, showing that the right properties of computation graphs are maintained after the reduction is applied:

Proposition 26 *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**$(f(n))$.*

5 Counting up to a constant number

We are now going to consider the (functional) problem of **path counting**. **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 this work we are going to focus on bounded version(s) of counting—the problem **Count**$k\mathcal{X}$ will be the one of counting *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 $\mathcal{X}$.[3] The canonical problem of **reachability** (denoted **Reach**) is of course equivalent to counting "up to one" path.

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 $\mathcal{C}$.

How do counting problems for different values of k relate to each other? Obviously, decreasing the counter limit can only make the problem easier, as we can simply glue together the previously distinct answers. In the other direction, the following can be shown (recall that $\mathbb{REM}$ is the family of CCG classes closed under edge removal):

Proposition 27 *For any class $\mathcal{C} \in \mathbb{REM}$ and constant $k \geq 1$,*
$\mathcal{C}$-**Count**$(k+1)$**SF** $\preceq_{\mathbf{D}}^{\mathcal{C}\text{-}\mathbf{Count1SF}}$ $\mathcal{C}$-**Count**k**SF**.

In words, we show that given a graph G from a class $\mathcal{C} \in \mathbb{REM}$, and an algorithm for $\mathcal{C}$-**Count**1**SF**, 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 $\langle \mathcal{C}$-**Count**k**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$(G')$ (note that 2 really means $2^+$)
```

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 $\mathcal{C}$-**Count**1**SF**$(G - e) = 0$.

Now, 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

[3]In this notation, a result of Allender, Reinhardt and Zhou (Theorem 5.1 in [10]) implies that for a polynomial p, **Count**p**SF** $\in$ **QFunc**$(\log(n))$ (i.e., limited counting *can* be solved non-deterministically in logarithmic space, but with *no* bounds on ambiguity).

is removing edges and we have chosen $\mathcal{C}$ to be one of the classes closed under this operations, all calls to $\mathcal{C}$-**Count**1**SF** 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. □

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

Proposition 28 *For any class $\mathcal{C} \in \mathbb{REM}$ and constant $k \geq 1$,*
$\mathcal{C}$-**Count**$(k+1)$**AF** $\preceq_{\mathbf{D}}^{\mathcal{C}\text{-}\mathbf{Count1AF}}$ $\mathcal{C}$-**Count**k**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 27 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 $\mathcal{C}$-**Count**1**AF** to check whether y is reachable from z. □

Corollary 29 *For any classes $\mathcal{C} \in \mathbb{REM}$, $\mathcal{D} \in \mathbb{UNI}$, and constant $k \geq 1$,*
$\mathcal{C}$-**Reach** $\in \mathcal{D}$-**QFunc**$(\log(n)) \iff \mathcal{C}$-**Count**$k$**AF** $\in \mathcal{D}$-**QFunc**$(\log(n))$,
$\mathcal{C}$-**Reach** $\in \mathcal{D}$-**QFunc**$(\log(n)) \iff \mathcal{C}$-**Count**$k$**SF** $\in \mathcal{D}$-**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 30 *For any class $\mathcal{D} \in \mathbb{UNI}$, bound p, and constant k,*

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

6 Inductive Counting

In this section we revisit the 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. Counting the vertices in these layers allows us to solve **Reach**—it is enough to compare the counts of L_n for the graph with the target vertex present and removed.

Let us start with the breakthrough due to Immerman and Szelepcsényi (see [12, 13]). Denoting the number of vertices in L_k by C_k, we can calculate C_{k+1} from C_k within **QFunc**(log(n))(**guess** denotes making a nondeterministic choice).

Algorithm 31 (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 see that on all of the nondeterministic branches that have not failed, the value of C_{k+1} has been computed correctly (again, see [12, 13] for details). Analyzing more carefully the nondeterministic branches on which the above algorithm may succeed, we can show more:

Proposition 32 [4] *1**AA-Reach** $\in$ 1**AF-QFunc**(log(n)), 1**SA-Reach** $\in$ 1**SF-QFunc**(log(n)) and 1**MA-Reach** $\in$ 1**MF-QFunc**(log(n)).*

Proof: 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 31 guesses paths between the same pairs of vertices over and over again, and thus the result does not immediately extend to higher path counts. However, we can modify the algorithm following Buntrock, Hemachandra and Siefkes (see [3]) to guess and verify *all* the paths to every reachable vertex. First, let us note that a graph can be easily (deterministically in log-space) transformed to a one which is equivalent w.r.t. the number of paths between vertices, and has out-degree 2. Having done this, and assuming that we have guessed the number p of distinct paths (of length at most l) between vertices u and v, we can use the following procedure to verify their existence:

Algorithm 33

[4]Note that the result of [8] is actually stronger, showing that 1**AA-Reach** $\in$ 1**AA-QFunc**(log(n)). However, it is based on a different algorithm. Here we are trying to show that a careful analysis of *the same* algorithm in light of our framework might provide stronger results "for free".

```
**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). However, we can keep track of the *collective* number of paths T_k to all vertices in L_k, and use it to cut off these "unfortunate" branches:

Algorithm 34

```
1.  set $T_{k+1} := 1$
2.  for every $v \in V$:
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 $T_{k+1} := T_{k+1} + r$
```

It is not difficult to show that the value of T_{k+1} will be correctly computed on exactly one nondeterministic branch, and that all other branches will fail. Moreover, a smart use of tail recursion makes it possible to carry **guesspaths** (and thus the whole Algorithm 34) within space $O(\log(np)\log(p))$. Thus Buntrock et al. obtain the following:

Proposition 35 p**AA-Reach** $\in 1$**AF-QFunc**$(\log(np)\log(p))$.

Algorithm 31 considers *all* paths to every vertex. It is however possible to adapt it to consider only the *minimal length* paths (following Allender and Reinhardt, see [9]), by counting 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). We refer the reader to [9] for details on this algorithm.

Knowing Σ_k in addition to C_k does not seem to have any interesting consequences. However, 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, Allender and Reinhardt conclude what in our terms can be expressed as the following:

Corollary 36 *1**MA-Reach** $\in$ 1**AF-QFunc**$(\log(n))$.*

As it turns out, the same "double counting" technique can be applied to Algorithm 34. First, we can make sure that **guesspaths()** considers only paths of length *exactly* l (instead of up to l). Then, making T_k (Σ_k) denote the collective number (sum of lengths, respectively) of all 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 37

```
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 $T'_k := T'_k + p$, $\Sigma'_k := \Sigma'_k + lp$
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 $\Sigma_{k+1} := \Sigma_{k+1} + dr$
```

Corollary 38 *p**MA-Reach** $\in$ 1**AF-QFunc**$(\log(np)\log(p))$.*

Corollary 39 *For a constant k, k**MA-Reach** $\in$ 1**AF-QFunc**$(\log(n))$.*

7 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 here. Thus we would like to point out at least the following directions for future work:

1. Trying to improve the results outlined in Section 5 from constant to polynomial bounds on path count (this would 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. Incorporating strong unambiguity (and possibly other restrictions), into the framework.

3. Enriching the framework by means of limiting the number of oracle queries allowed (on one hand it would allow a stronger version of Proposition 26; on the other, it is not clear whether at most logarithmically many queries can be of any advantage—in particular, such a situation does not even allow the machine to read the whole input).

4. Considering larger space bounds, in particular $O(n)$.

References

[1] Àlvarez, C., Jenner, B.: A Very Hard Log Space Counting Class, In: 5th Annual Conference on Structure in Complexity Theory, pp. 154–168 (1990)

[2] Buntrock, G., Jenner, B., Lange, K., Rossmanith, P.: Unambiguity and Fewness for Logarithmic Space, In: 8th International Conference on Fundamentals of Computation Theory, Volume 529 of Lecture Notes in Computer Science, pp. 168–179 (1991)

[3] Buntrock, G., Hemachandra, L., Siefkes, D.: Using Inductive Counting to Simulate Nondeterministic Computation, Information and Computation 102 (1), pp. 102–117 (1993)

[4] Allender, E., Lange, K.: StUSPACE($\log n$) is Contained in DSPACE($\log^2 n/\log\log n$), In: Electronic Colloquium on Computational Complexity (ECCC), Volume 3 (1996)

[5] Herman, G.: Unambiguous functions in logarithmic space. PhD thesis, McMaster University (2009).

[6] Herman, G., Soltys, M.: Unambiguous functions in logarithmic space. In: Wolfgang Merkle Klaus Ambos-Spies, Benedikt Löwe, editor, Mathematical Theory and Computational Practice. 5th Conference on Computability in Europe, CiE'09, University of Heidelberg, pp. 162-175 (2009).

[7] Lange, K.: Nondeterministic Logspace Reductions, In: 11th Symposium on Mathematical Foundations of Computer Science, Volume 176 of Lecture Notes in Computer Science, pp. 378–388 (1984)

[8] Lange, K.: An Unambiguous Class Possessing a Complete Set, In: 14th Annual Symposium on Theoretical Aspects of Computer Science, Volume 1200 of Lecture Notes in Computer Science, pp. 339–350 (1997)

[9] Reinhardt, K., Allender, E.: Making Nondeterminism Unambiguous, In: 38th Annual Symposium on Foundations of Computer Science, pp. 244–253 (1997)

[10] Allender, E., Reinhardt, K., Zhou, S.: Isolation, Matching, and Counting: Uniform and Nonuniform Upper Bounds, Journal of Computer and System Sciences 59 (2), pp. 164–181 (1999)

[11] Jenner, B., Kirsig, B.: Alternierung und Logarithmischer Platz, Dissertation, Universität Hamburg (1989)

[12] Immerman, N.: Nondeterministic space is closed under complementation, In: 3rd Annual Conference on Structure in Complexity Theory, pp. 112–115 (1988)

[13] Szelepcsényi, R.: The method of forced enumeration for nondeterministic automata, Acta Informatica 26 (3), pp. 279–284 (1988)

[14] Ladner, R., Lynch, N.: Relativization of Questions About Log Space Computability, Theory of Computing Systems 10 (1), pp. 19–32 (1976)

[15] Litow, B., Sudborough, I.: On non-erasing oracle tapes in space bounded reducibility, SIGACT News 10 (2), pp. 53–57 (1978)

[16] Lynch, N.: Log Space machines with Multiple Oracle Tapes, Theoretical Computer Science 6, pp. 25–39 (1978)

[17] Ruzzo, W., Simon, J., Tompa, M.: Space-bounded hierarchies and probabilistic computations, In: 14^{th} Annual ACM Symposium on Theory of Computing, pp. 215–223 (1982)

[18] Stearns, R.E., Hartmanis, J., Lewis, P.M.: Hierarchies of memory limited computations, In: 6^{th} Annual IEEE Symposium on Switching Circuit Theory and Logical Design (1965)