Unambiguous Catalytic Computation

Chetan Gupta
Indian Institute of Technology Kanpur, India
gchetan@cse.iitk.ac.in

Rahul Jain
Indian Institute of Technology Kanpur, India
jain@cse.iitk.ac.in

Vimal Raj Sharma
Indian Institute of Technology Kanpur, India
vimalraj@cse.iitk.ac.in

Raghunath Tewari
Indian Institute of Technology Kanpur, India
rtewari@cse.iitk.ac.in

Abstract

The catalytic Turing machine is a model of computation defined by Buhrman, Cleve, Koucký, Loff, and Speelman (STOC 2014). Compared to the classical space-bounded Turing machine, this model has an extra space which is filled with arbitrary content in addition to the clean space. In such a model we study if this additional filled space can be used to increase the power of computation or not, with the condition that the initial content of this extra filled space must be restored at the end of the computation.

In this paper, we define the notion of unambiguous catalytic Turing machine and prove that under a standard derandomization assumption, the class of problems solved by an unambiguous catalytic Turing machine is same as the class of problems solved by a general nondeterministic catalytic Turing machine in the logspace setting.

2012 ACM Subject Classification Mathematics of computing

Keywords and phrases Catalytic computation, Logspace, Reinhardt-Allender

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2019.16

Funding *Rahul Jain*: Ministry of Human Resource Development, Government of India
Raghunath Tewari: DST Inspire Faculty Grant, Visvesvaraya Young Faculty Fellowship

Acknowledgements The fourth author would like to thank Michal Koucký for valuable discussions and for suggesting key ideas which led to the proof of the main result in this paper. The first and third author would like to thank Ministry of Electronics and IT, India for supporting this research through the Visvesvaraya PhD. The authors would also like to thank the anonymous reviewers for their valuable comments which helped in improving the presentation of this paper and suggesting an alternative proof of CNL = coCNL as a corollary of our result.

1 Introduction

The catalytic computational model was first introduced by Buhrman et al. [2]. It is a computational model constructed by equipping a standard Turing machine with a large *auxiliary tape* in addition to its work tape. This auxiliary tape is filled with arbitrary data which must be restored at the end of the computation. A catalytic Turing machine with a workspace of size $s(n)$ can be assumed to have auxiliary space of size $2^{s(n)}$. The question that arises is, whether having access to this additional tape increases the power of computation or not. At first, this extra filled space seemed to be of no use. However, surprisingly, Buhrman et al. [2] showed that there exist some problems which can be solved by a deterministic

catalytic logspace Turing machine (CL) but are not known to be solvable by a standard deterministic logspace Turing machine (L), or even nondeterministic logspace NL. More precisely, they showed that $\mathsf{uniformTC}^1 \subseteq \mathsf{CL} \subseteq \mathsf{ZPP}$ and $\mathsf{uniformTC}^1$ is known to contain NL which is believed to be different from L. This result gives the motivation to explore the power of the catalytic Turing machine further.

In a subsequent result, Buhrman et al. [3] defined a nondeterministic catalytic computational model. In a nondeterministic catalytic Turing machine, the content of the auxiliary tape must be restored to its initial content for every sequence of nondeterministic choices. The nondeterministic equivalent of CL is called CNL. Using the same observation as in [2] they showed that $\mathsf{CNL} \subseteq \mathsf{ZPP}$. They also showed that, under a standard derandomization assumption, the class of problems solvable by a nondeterministic logspace catalytic Turing machine (CNL) is closed under complement, that is $\mathsf{CNL} = \mathsf{coCNL}$. To prove this, they first show that on a specific input x most of the configuration graphs of a CNL machine will be of polynomial size. They use the pseudorandom generator of [5] to obtain a polynomial size configuration graph. However, having access to a polynomial size graph is not enough because the size of a vertex in the graph is still exponentially larger than the size of the workspace. To circumvent this problem, they use a *hash function* picked from a hash family which maps these vertices injectively to smaller values. After that, they apply the inductive counting technique of Immerman and Szelepcsényi on this smaller size graph to obtain the final result [4, 8].

In this paper, we define a variant of nondeterministic catalytic Turing machine called *unambiguous* catalytic Turing machine. Analogous to the standard Turing machine, an unambiguous catalytic Turing machine is a nondeterministic catalytic Turing machine which has at most one accepting path on each input.

We show that under the same derandomization assumption as that of [3], in the logspace setting, unambiguous catalytic Turing machine (CUL) and nondeterministic catalytic Turing machine are equivalent in power. This is stated formally in the following theorem.

▶ **Theorem 1** (Main Theorem). *If there exists a constant* $\epsilon > 0$ *such that* $\mathsf{DSPACE}(n) \nsubseteq \mathsf{SIZE}(2^{\epsilon n})$*, then* $\mathsf{CNL} = \mathsf{CUL}$.

We prove Theorem 1 by giving an unambiguous logspace catalytic algorithm which answers if in the configuration graph of a CNL machine, the accepting vertex is reachable from the starting vertex or not. For this, we use (i) the pseudorandom generator used by [1] and [3] to obtain a small size min-unique weighted configuration graph of the CNL machine, (ii) the hashing scheme of [3] which maps the vertices of the configuration graph to smaller values, and (iii) the double counting technique of [7]. Our result is analogous to a result of [1] in the traditional Turing machine model, where authors prove that, if there are problems in $\mathsf{DSPACE}(n)$ which require exponential size circuits, then $\mathsf{UL} = \mathsf{NL}$.

The rest of the paper is organized as follows. Section 2 contains definitions of nondeterministic and unambiguous catalytic computation. We state the derandomization assumption under which our result holds, pseudorandom generators and the hashing scheme that we have used. In Section 3, we prove the main result $\mathsf{CUL} = \mathsf{CNL}$.

2 Preliminaries

In this section, we present the necessary definitions, notations, and lemmas. We start with the definition of a nondeterministic catalytic Turing machine as defined in [3].

▶ **Definition 2.** Let $\mathcal{M}$ be a nondeterministic Turing machine with three tapes: one input tape, one work tape, and one *auxiliary tape*.

Let $x \in \{0,1\}^n$ be an input, and $w \in \{0,1\}^{s_a(n)}$ be the initial content of the auxiliary tape. We say that $\mathcal{M}(x,w)$ accepts x if there exists a sequence of nondeterministic choices that makes the machine accept. If for all possible sequences of nondeterministic choices $\mathcal{M}(x,w)$ does not accept, the machine rejects x.

Then $\mathcal{M}$ is said to be a *catalytic nondeterministic* Turing machine using workspace $s(n)$ and auxiliary space $s_a(n)$ if for all inputs, the following three properties hold.

1. **Space bound**. The machine $\mathcal{M}(x,w)$ uses space $s(n)$ on its work tape and space $s_a(n)$ on its auxiliary tape.
2. **Catalytic condition**. $\mathcal{M}(x,w)$ halts with w on its auxiliary tape, irrespective of its nondeterministic choices.
3. **Consistency**. The outcome of the computation is consistent among all initial aux-tape content w. $\mathcal{M}(x,w)$ should either accept for all choices of w — in which case we say $\mathcal{M}$ accepts x — or it rejects for all possible w — $\mathcal{M}$ rejects x. However, the specific nondeterministic choices that make $\mathcal{M}(x,w)$ go one way or the other may depend on w.

▶ **Definition 3.** $\mathsf{CNSPACE}(s(n))$ is the set of decision problems that can be solved by a nondeterministic catalytic Turing machine with at most $s(n)$ size workspace and $2^{s(n)}$ size auxiliary space. CNL denotes the class $\mathsf{CNSPACE}(O(\log n))$.

Unambiguous computation is a natural restriction of nondeterministic computation where on every input the Turing machine can have at most one nondeterministic path which accepts the input. In the domain of catalytic computation, the definition naturally extends as follows.

▶ **Definition 4.** An *unambiguous* catalytic Turing machine is a nondeterministic catalytic Turing machine which on every input produces at most one sequence of nondeterministic choices where the machine accepts.

▶ **Definition 5.** $\mathsf{CUSPACE}(s(n))$ is the set of decision problems that can be solved by an unambiguous catalytic Turing machine with at most $s(n)$ size workspace and $2^{s(n)}$ size auxiliary space. CUL denotes the class $\mathsf{CUSPACE}(O(\log n))$.

In order to present our result, we will use the notion of *configuration graph*. Configuration graphs of a classical Turing machine are used heavily in proving space-bounded computation results. A modified version of configuration graph was used for catalytic computations by Buhrman et al. in [3]. They defined the configuration graph in the context of catalytic computation in the following way: Let $\mathcal{M}$ be a nondeterministic catalytic Turing machine with $c\log n$ size workspace and n^c size auxiliary space. Then, $\mathcal{G}_{\mathcal{M},x,w}$ denotes the configuration graph of a nondeterministic catalytic Turing machine $\mathcal{M}$ on input x and initial auxiliary content w. Every vertex of $\mathcal{G}_{\mathcal{M},x,w}$ corresponds to a configuration of $\mathcal{M}$ reachable from the initial configuration of $\mathcal{M}$ which consists of the content of the work tape and the auxiliary tape, head positions of all three tapes and the current state. The graph has a directed edge from a vertex ver_1 to a vertex ver_2 if the configuration corresponding to ver_2 can be reached from the configuration corresponding to ver_1 in one step in $\mathcal{M}$. We will denote the number of the vertices in a configuration graph $\mathcal{G}_{\mathcal{M},x,w}$ by $|\mathcal{G}_{\mathcal{M},x,w}|$.

A configuration of a nondeterministic catalytic Turing machine $\mathcal{M}$ with $c\log n$ size workspace and n^c size auxiliary space can be described with at most $c\log n + n^c + \log n + \log(c\log n) + \log n^c + O(1) = O(n^c)$ bits, where we need $c\log n + n^c$ bits for work and auxiliary tape content, $\log n + \log(c\log n) + \log n^c$ bits for the tape heads, and $O(1)$ bits for the state information. Thus, the total number of configurations of $\mathcal{M}$ can be upper bounded by $2^{O(n^c)}$, which also implies $|\mathcal{G}_{\mathcal{M},x,w}| \leq 2^{O(n^c)}$ for an input x and initial auxiliary content w.

In Section 3, we will prove $\mathsf{CUL} = \mathsf{CNL}$ under the same assumption the following derandomization result holds.

▶ **Lemma 6** ([5, 6]). *If there exists a constant $\epsilon > 0$ such that $\mathsf{DSPACE}(n) \nsubseteq \mathsf{SIZE}(2^{\epsilon n})$ then for all constants c there exists a constant c' and a function $G : \{0,1\}^{c' \log n} \to \{0,1\}^n$ computable in $O(\log n)$ space, such that for any circuit C of size n^c*

$$\left| \Pr_{r \in \{0,1\}^n}[C(r) = 1] - \Pr_{s \in \{0,1\}^{c' \log n}}[C(G(s)) = 1] \right| < \frac{1}{n}.$$

Buhrman et al. in [3], showed a way to get a small size configuration graph of a nondeterministic logspace catalytic Turing machine. We will use the following lemma in our result, a stronger version of which was proved in [3].

▶ **Lemma 7** ([3]). *Let $\mathcal{M}$ be a nondeterministic catalytic Turing machine using $c \log n$ size workspace and n^c size auxiliary space. If there exists a constant $\epsilon > 0$ such that $\mathsf{DSPACE}(n) \nsubseteq \mathsf{SIZE}(2^{\epsilon n})$, then there exists a function $G : \{0,1\}^{O(\log n)} \to \{0,1\}^{n^c}$, such that on every input x and initial auxiliary content w, for at least one seed $s \in \{0,1\}^{O(\log n)}$, $|\mathcal{G}_{\mathcal{M},x,w \oplus G(s)}| \leq n^{2c+3}$. Moreover, G is logspace computable. ($w \oplus G(s)$ represents the bitwise XOR of w and $G(s)$)*

Let $\mathcal{G}$ be a directed graph, with vertex set $V(\mathcal{G})$ and edge set $E(\mathcal{G})$. Then, a weight function for $\mathcal{G}$ is a map $W : E(\mathcal{G}) \to \mathbb{N}$ which maps every edge in $E(\mathcal{G})$ to a natural number. Let $\mathcal{G}_W$ denote the weighted graph with respect to the weight function W. We say a weight function is a k-bit weight function if every edge in $E(\mathcal{G})$ gets a weight in the range $[0, 2^k - 1]$. A k-bit weight function for a graph $\mathcal{G}$ of n vertices can be thought of as a kn^2 length binary string $b = b_1 b_2 \dots b_{k.n^2}$. In such a representation, the weight assigned to the ith edge e_i of $\mathcal{G}$ is $W(e_i) = \textsc{Dec}(b_{j+1} b_{j+2} \dots b_{j+k})$, where $j = k.(i-1)$ and $\textsc{Dec}(x)$ is the natural number whose decimal representation is the binary string x.

We say $\mathcal{G}_W$ is *min-unique*, if there is a unique minimum weight path between every pair of vertices in $\mathcal{G}_W$. For any two vertices u and v in $V(\mathcal{G}_W)$, we denote the weight of the minimum weight path from u to v by $\mathsf{dist}(u, v)$. The following lemma implicit in [1] shows that under the assumption of Lemma 6 there exists a logspace computable pseudorandom generator which gives an $O(\log n)$-bit min-unique weight function for any graph of n vertices.

▶ **Lemma 8.** *If there exists a constant $\epsilon > 0$ such that $\mathsf{DSPACE}(n) \nsubseteq \mathsf{SIZE}(2^{\epsilon n})$, then there exists a logspace computable function $W : \{0,1\}^{O(\log n)} \to \{0,1\}^{c' n^2 \log n}$, such that for any directed graph $\mathcal{G}$ of n vertices there exists at least one seed $s' \in \{0,1\}^{O(\log n)}$ for which $\mathcal{G}_{W(s')}$ is min-unique.*

We also borrow the following lemma about the existence of a hash family from [3].

▶ **Lemma 9** ([3]). *For every n, there exists a family of hash functions $\{h_k\}_{k=1}^{n^3}$, with each h_k a function $\{0,1\}^n \to \{0,1\}^{4 \log n}$, such that the following properties hold. First, h_k is computable in space $O(\log n)$ for every k, and second, for every set $S \subset \{0,1\}^n$ with $|S| \leq n$ there is a hash function in the family that is injective on S.*

▶ **Definition 10.** Let $\mathcal{G}$ be a directed graph, $h : V(\mathcal{G}) \to \{1, 2, \dots, n\}$ be a hash function and W be a weight function for a graph of n vertices. Then, the *hashed-weighted* graph denoted by $\mathcal{G}_{h,W}$ is a weighted graph, such that every edge $uv \in E(\mathcal{G}_{h,W})$ has weight $W(uv) = W(h(u)h(v))$.

3 Reinhardt-Allender's Double Counting in the Catalytic Setting

In this section, we will prove Theorem 1 by constructing a CUL machine $\mathcal{M}'$ which accepts the same language as that of a given CNL machine $\mathcal{M}$. The core idea is to use the double counting technique of Reinhardt and Allender [7] on the configuration graph of $\mathcal{M}$. However, there are few hurdles to implement it.

Firstly note that, as shown by Buhrman et al. [3], the configuration graph of a CNL machine $\mathcal{M}$ can be of exponential size. Therefore, it is not possible for $\mathcal{M}'$ to do double counting on this graph in its workspace, which is just logarithmic in size. To handle this problem, we use the pseudorandom generator of Lemma 7 to get a small size configuration graph. But this does not solve the problem completely as the size of a vertex in the configuration graph is still very large. To solve this, we use the family of hash functions described in Lemma 9. One of these functions injectively maps the vertices of the configuration graph to small hashed values. Both the pseudorandom generator of Lemma 7 and hash function family of Lemma 9 were also used in [3] for performing inductive counting. In our result, to do double counting we need to make the configuration graph min-unique, therefore, we also use the pseudorandom generator of Lemma 8.

In $\mathcal{M}'$, we do the double counting on the configuration graph of $\mathcal{M}$ for every possible triplet consisting of seeds of the pseudorandom generators of Lemma 7 and 8 and a hash function from the hash family of Lemma 9. During the double counting we move on to the next triplet if the hash function doesn't map the vertices injectively or the configuration graph is not min-unique, otherwise, after finishing double counting we accept if the accepting node in the configuration graph was encountered at some point during the process.

We detect if the configuration graph is not min-unique the same way Reinhardt and Allender do it in [7]. Detecting if a hash function doesn't map the vertices injectively is tricky. Note that, to check whether two vertices of the configuration graph have been mapped to the same value or not cannot be done directly by storing them in the workspace of $\mathcal{M}'$. This is because the size of those vertices can be large. Therefore, we perform a clever bit by bit comparison of these vertices to check if they have been mapped to the same value or not. We outline this procedure in Algorithm 3.

In the following lemma, we prove the existence of the pseudorandom generators and the family of hash functions in the context of a configuration graph of a CNL machine.

▶ **Lemma 11.** *Let $\mathcal{M}$ be a nondeterministic catalytic Turing machine using $c \log n$ size workspace and n^c size auxiliary space. For an input x and auxiliary content w, let G be the pseudorandom generator as given in Lemma 7 and s be a seed such that, $|\mathcal{G}_{\mathcal{M},x,w\oplus G(s)}| \leq N$, where $N = n^{2c+3}$. Then,*

1. *there exists a family of logspace computable hash functions $H = \{h_k\}_{k=1}^{O(N^3)}$, such that for each k we have $h_k : \{0,1\}^N \to \{0,1\}^{4 \log N}$, and at least one $h_k \in H$ injectively maps $V(\mathcal{G}_{\mathcal{M},x,w\oplus G(s)})$ to $\{0,1\}^{4 \log N}$.*
2. *if there exists a constant $\epsilon > 0$ such that $\mathsf{DSPACE}(n) \not\subseteq \mathsf{SIZE}(2^{\epsilon n})$, then there exists a logspace computable function $W : \{0,1\}^{O(\log N^4)} \to \{0,1\}^{c' N^8 \log N^4}$, such that for at least one seed $s' \in \{0,1\}^{O(\log N^4)}$, the hashed-weighted graph $\mathcal{G}_{\mathcal{M},x,w\oplus G(s),h_k,W(s')}$ is min-unique, where h_k injectively maps $V(\mathcal{G}_{\mathcal{M},x,w\oplus G(s)})$ to $\{0,1\}^{4 \log N}$.*

Proof. We know that the size of a vertex(configuration) in $\mathcal{G}_{\mathcal{M},x,w\oplus G(s)}$ can be upper bounded by $O(n^c)$. For the sake of simplicity, we assume that $O(n^c) \leq N$. Then, 1 follows directly from Lemma 9 if you take the set S(of Lemma 9) as $V(\mathcal{G}_{\mathcal{M},x,w\oplus G(s)})$.

Now, consider the graph $\mathcal{G}_{\mathcal{M},x,w\oplus G(s)}$ where every vertex is hashed by h_k to some value in the range $[0, N^4 - 1]$ injectively. If we treat this hashed graph as a graph of N^4 many vertices, then 2 follows from Lemma 8. ◀

3.1 Proof of Main Theorem

Since $\mathsf{CUL} \subseteq \mathsf{CNL}$ follows by definition, we only need to show that $\mathsf{CNL} \subseteq \mathsf{CUL}$. Let $\mathcal{M}$ be a nondeterministic catalytic Turing machine with $c \log n$ size workspace and n^c size auxiliary space. We will prove $\mathsf{CNL} \subseteq \mathsf{CUL}$ by showing that there exists an unambiguous catalytic Turing machine $\mathcal{M}'$ with $c' \log n$ size workspace and $n^{c'}$ size auxiliary space, where c' is sufficiently larger than c, such that on every input x and auxiliary content w, $\mathcal{M}(x, w)$ accepts if and only if $\mathcal{M}'(x, w)$ accepts. For the sake of simplicity, we assume that $\mathcal{M}$ has a unique configuration when it accepts an input. Let acc_w and $start_w$ denote the accept and start configurations of $\mathcal{M}$ on input x and auxiliary content w respectively.

Let G, H, W and N be as given in Lemma 11. For $s \in \{0,1\}^{O(\log n)}$, $h_k \in H$, and $s' \in \{0,1\}^{O(\log N^4)}$, we say a triplet $\langle s, h_k, s' \rangle$ is a *good* triplet, if (1) h_k injectively maps the vertices of $\mathcal{G}_{\mathcal{M},x,w\oplus G(s)}$ to $\{0,1\}^{4 \log N}$ and (2) $\mathcal{G}_{\mathcal{M},x,w\oplus G(s),h_k,W(s')}$ is min-unique. Otherwise, we call it a *bad* triplet. Existence of a good triplet follows directly from Lemma 11.

In our algorithm for $\mathcal{M}'$, we iterate over all possible combinations of s, h_k, and s'. In each iteration we work with the hashed-weighted configuration graph $\mathcal{G}_{\mathcal{M},x,w\oplus G(s),h_k,W(s')}$. For a good triplet $\langle s, h_k, s' \rangle$, our algorithm **Accept**s if there is a path from $start_{w\oplus G(s)}$ to $acc_{w\oplus G(s)}$. Otherwise, for a bad triplet $\langle s, h_k, s' \rangle$ the algorithm moves on to the next triplet.

To perform the double counting technique on $\mathcal{G}_{\mathcal{M},x,w\oplus G(s),h_k,W(s')}$, we need to identify the vertices which are at distance i from $start_{w\oplus G(s)}$. For this, our algorithm uses an unambiguous procedure Reachable. Another unambiguous procedure Badgraph is used to check if h_k maps $V(\mathcal{G}_{\mathcal{M},x,w\oplus G(s),h_k,W(s')})$ to $\{0,1\}^{4 \log N}$ injectively or not.

Algorithm 1 outlines the main algorithm of $\mathcal{M}'$, Algorithm 2 and Algorithm 3 outline the procedures Reachable and Badgraph respectively.

3.1.1 Description of the Algorithm 1

Let x be the input and w be the auxiliary content of $\mathcal{M}'$. We iterate over all triplets $\langle s, h_k, s' \rangle$ using the loop of line 2. In line 3, we set w to $w \oplus G(s)$ and weight function wt to $W(s')$. For the sake of simplicity, we assume that the function wt assigns weight one to every edge. If not we can always split an edge with weight l to an l length path, similar to how it is done in Lemma 2.1 of [7].

Note that from now onwards, we will denote the hashed-weighted graph for the fixed triplet $\langle s, h_k, s' \rangle$ by $\mathcal{G}_{\mathcal{M},x,w,h_k,wt}$. We define two sets $C_{=i}$ and $C_{<i}$ for $\mathcal{G}_{\mathcal{M},x,w,h_k,wt}$ as follows:

- $C_{=i} = \{ver \in V(\mathcal{G}_{\mathcal{M},x,w,h_k,wt}) \mid \mathsf{dist}(start_w, ver) = i\}$,
- $C_{<i} = \bigcup_{j=0}^{i-1} C_{=j}$.

For applying double counting technique, we use two counters c_i and d_i, where, $c_i = |C_{<i+1}|$ and $d_i = \Sigma_{ver \in C_{<i+1}} \mathsf{dist}(start_w, ver)$. Clearly, $c_0 = 1$ and $d_0 = 0$. From lines 5 to 19, we compute the counters c_i and d_i iteratively from the values of c_{i-1} and d_{i-1}. Since for a good triplet $\langle s, h_k, s' \rangle$, $|\mathcal{G}_{\mathcal{M},x,w,h_k,wt}|$ can not be more than N^4, we compute c_i's and d_i's for $i = 1$ to M, where $M = N^4$. Note that we set $M = N^4$ for a special case where wt assigns weight one to every edge, otherwise, its value can be different and need to be set accordingly.

Since h_k maps the vertices of $\mathcal{G}_{\mathcal{M},x,w,h_k,wt}$ on $\{0,1\}^{4 \log N}$, we go over all possible hashed values v from 0 to $M - 1$ and check if there is a vertex ver in $\mathcal{G}_{\mathcal{M},x,w,h_k,wt}$ such that $\mathsf{dist}(start_w, ver) = i$ and $h_k(ver) = v$, using the procedure Reachable. If there exists such a vertex ver than we increment c_i and d_i accordingly in line 10. Moreover, if ver is an accepting node then we store this information in the $final$ variable.

Algorithm 1 Algorithm of $\mathcal{M}'$.

G and W are as described in Lemma 11. S and S' are the set of seeds for G and W respectively and H is the hash function family. $M = N^4$ is the maximum size of the configuration graph for a good triplet $\langle s, h_k, s' \rangle$.

```
1:  procedure UNAMBIGUOUSSIMULATION(Input x, Auxiliary Content w)
2:      for ⟨s, h_k, s'⟩ ∈ S × H × S' do
3:          w ← w ⊕ G(s), wt ← W(s'), final ← FALSE
4:          c_0 ← 1, d_0 ← 0
5:          for i = 1 to M do
6:              c_i ← c_{i−1}, d_i ← d_{i−1}
7:              for v = 0 to M − 1 do
8:                  (found, finalreach) ← REACHABLE(v, i, h_k, wt, c_{i−1}, d_{i−1}, s)
9:                  if found = TRUE then
10:                     c_i ← c_i + 1, d_i ← d_i + i
11:                     if finalreach = TRUE then
12:                         final ← TRUE
13:                     end if
14:                 else if found = BAD then
15:                     w ← w ⊕ G(s)
16:                     Jump to line 2
17:                 end if
18:             end for
19:         end for
20:         w ← w ⊕ G(s)
21:         if final = TRUE then
22:             Accept
23:         end if
24:     end for
25:     Reject
26: end procedure
```

If the triplet $\langle s, h_k, s' \rangle$ is bad, we catch it while computing the values of c_i's and d_i's using REACHABLE and move to the next triplet after restoring the initial auxiliary content w in line 15 by XORing it again with $G(s)$. If it is good, then the loop of line 2 terminates normally. After which we restore w in line 20 and **Accept** if $final$ is TRUE in line 22 or move to the next triplet if $final$ is FALSE.

Finally, if acc_w is not reachable from $start_w$ for any good triplet $\langle s, h_k, s' \rangle$, we **Reject** in line 25.

3.1.2 Description of the Algorithm 2

REACHABLE$(v, i, h_k, wt, c_{i-1}, d_{i-1}, s)$ is called only if the following conditions are satisfied:

- **Condition A**: All the vertices in $C_{<i}$ have a unique minimum weight path from $start_w$.
- **Condition B**: All the vertices in $C_{<i}$ are injectively mapped to the set $\{0,1\}^{4 \log N}$.

We define the following conditions based on which REACHABLE detects a bad triplet:

- **Condition I**: There exists a vertex $ver1$ in $C_{<i}$ and a vertex $ver2$ in $C_{=i}$, such that $h_k(ver1) = h_k(ver2) = v$. ($h_k$ *doesn't map* $V(\mathcal{G}_{\mathcal{M},x,w,h_k,wt})$ *to* $\{0,1\}^{4 \log n}$ *injectively.*)

- **Condition II**: There exist vertices $ver1$ and $ver2$ in $C_{=i}$ such that $h_k(ver1) = h_k(ver2) = v$. ($h_k$ *doesn't map* $V(\mathcal{G}_{\mathcal{M},x,w,h_k,wt})$ *to* $\{0,1\}^{4\log n}$ *injectively.*)
- **Condition III**: There exists a vertex ver in $C_{=i}$ such that $h_k(ver) = v$ and ver has more than one minimum weight paths from $start_w$. ($\mathcal{G}_{\mathcal{M},x,w,h_k,wt}$ *is not min-unique.*)

Reachable is a nondeterministic procedure which **Reject**s on all sequences of nondeterministic choices except one, where it returns one of the following pair of values:

- (BAD, FALSE) if at least one of the **Condition I, II,** and **III** is satisfied.
- (TRUE, TRUE) if none of the **Condition I, II,** or **III** are satisfied and there exists a vertex ver in $C_{=i}$, such that $h_k(ver) = v$ and $ver = acc_w$.
- (TRUE, FALSE) if none of the **Condition I, II,** or **III** are satisfied and there exists a vertex ver in $C_{=i}$, such that $h_k(ver) = v$ but $ver \neq acc_w$.
- (FALSE, FALSE) if none of the **Condition I, II,** or **III** are satisfied and there does not exist a vertex ver in $C_{=i}$, such that $h_k(ver) = v$.

Reachable in line 3-31, guesses the vertices of $C_{<i}$ in ascending order of their hashed values from h_k. In every iteration, it first cleans a portion of $\mathcal{M}'$'s workspace say z and selects $l \leq i-1$ nondeterministically. Then using the workspace z and auxiliary content w of $\mathcal{M}'$ it simulates the machine $\mathcal{M}$ on x and w for l steps.

During a simulation, we denote the current configuration of $\mathcal{M}$ by $(z, w, pos, state)$, where z denotes the work tape content, w denotes the auxiliary content, pos denotes the head positions on the different tapes and $state$ denotes the current state.

To ensure the ascending order, we use the variable h which is initially set to -1. After every simulation, we compare the hashed value of the current configuration to h in line 7. If the order is violated, we continue the simulation until $\mathcal{M}$ halts, restore w and **Reject**. If not, we assign $h_k(z, w, pos, state)$ to h and use it in the next iteration.

In line 13, we store the sum of the distance of all c_{i-1} many guessed vertices from $start_w$ in the variable d. Variable $v_{present}$ intends to store the information about the existence of a vertex ver in $C_{<i}$, such that $h_k(ver) = v$. In line 15, we set $v_{present}$ to TRUE if $h = v$.

In line 17-29, if $l = i-1$ then we increment cnt for every neighbour of the current configuration which hashes to v. We also set $finalreach$ to TRUE if a neighbour of the current configuration is an accepting node acc_w. Later, we use variables cnt and $v_{present}$ to decide the returning value of Reachable. In line 30, we continue the simulation until a halting state is reached to restore the auxiliary content.

Outside the loop, in line 32, we compare d with d_{i-1} and **Reject** if $d \neq d_{i-1}$. Since vertices in $C_{<i}$ have unique minimum weight path from $start_w$ and they were all guessed in ascending order of their hashed value, $d = d_{i-1}$ holds only for one sequence of nondeterministic choices. For a more detailed proof of why $d = d_{i-1}$ holds only for one sequence of nondeterministic choices, one can refer to Theorem 2.2 of [7].

$v_{present}$ = FALSE implies that there is no vertex ver in $C_{<i}$, such that $h_k(ver) = v$. In such a case, we return the appropriate value based on the value of cnt. $cnt = 0$ implies that there is no vertex ver in $C_{=i}$, such that $h_k(ver) = v$, therefore, we return (FALSE, FALSE). $cnt = 1$ implies that there is exactly one vertex ver in $C_{=i}$, such that $h_k(ver) = v$, therefore, we return (TRUE, $finalreach$). $cnt > 1$ implies that either the **Condition II** or **III** satisfies, therefore, we return (BAD, FALSE).

$v_{present}$ = TRUE implies that there is a vertex ver in $C_{<i}$, such that $h_k(ver) = v$. Here again, if $cnt = 0$ we return (FALSE, FALSE). But if $cnt > 0$, we need to check if **Condition I** is satisfied i.e. there is a vertex $ver' \neq ver$ for which we incremented cnt in line 22 when it was encountered through a path of weight i. Note that, in Reinhardt-Allender's algorithm we do not need to check this because there we do not work with hashed graphs.

We call the procedure BADGRAPH to check if the **Condition I** is satisfied or not. If BADGRAPH returns TRUE i.e. **Condition I** is satisfied, we return (BAD, FALSE). If BADGRAPH returns FALSE, then that means that all the vertices for which we incremented *cnt* in line 22 were actually the vertex *ver* encountered through a different path of weight i, hence we return (FALSE, FALSE).

3.1.3 Description of the Algorithm 3

BADGRAPH is also a nondeterministic procedure which **Reject**s on all sequences of nondeterministic choices except one, where it returns TRUE if **Condition I** is satisfied, else it returns FALSE.

BADGRAPH is called from REACHABLE if there is a vertex *ver* in $C_{<i}$, such that $h_k(ver) = v$ and $cnt > 0$. Let F denote the set of all the vertices for which *cnt* was incremented in the line 22 of REACHABLE.

In BADGRAPH we compare *ver* with every vertex in F one bit at a time because we cannot store all the bits due to the limited workspace of $\mathcal{M}'$. In line 2, we set g to be the index of the vertex in F we intend to compare with *ver*. In line 3, we set t to be the index of the bits that we intend to compare. Since a vertex is basically a configuration of machine $\mathcal{M}$ on input x and auxiliary content w, we keep $T = O(n^c)$.

From line 4 to 40, we compare the two bits by guessing the vertices of $C_{<i}$ in the same manner as we do in RECHABLE. In line 17, we store the tth bit of *ver* in $bit1$. To get the gth vertex of F we use the variable cnt' which is set to 0 initially in line 4. We increment cnt' by one every time $l = i - 1$ and neighbour of the current configuration hashes to v. Thus, $cnt' = g$ in line 25 implies that we have the gth vertex of F and we store the tth bit of that vertex in $bit2$.

In line 38, we compare both bits $bit1$ and $bit2$ and if they are unequal then that means that there is at least one vertex in F which is different from *ver* but both have the same hash value. That implies that **Condition I** is satisfied and hence we return TRUE.

If we never encounter unequal bits in line 38, then that means that all the vertices in F are actually the vertex *ver*. Therefore, we return FALSE in line 43.

3.1.4 Correctness of Algorithm 1

We divide the proof of correctness of the Algorithm 1 into two cases:

Case 1 - Triplet $\langle s, h_k, s' \rangle$ is good: We first prove that if triplet $\langle s, h_k, s' \rangle$ is good then given the correct values of c_{i-1} and d_{i-1} the ith iteration of the loop of line 5 correctly computes values of c_i and d_i.

First notice that, since triplet $\langle s, h_k, s' \rangle$ is good, REACHABLE will never return BAD for any of the v chosen in line 7. Now, for every vertex *ver* in $C_{=i}$, a call to REACHABLE$(h_k(ver), i, h_k, wt, c_{i-1}, d_{i-1}, s)$ will return (TRUE, $finalreach$) after which we update the values c_i and d_i accordingly in line 10. And for any vertex *ver* not in $C_{=i}$, a call to REACHABLE$(h_k(ver), i, h_k, wt, c_{i-1}, d_{i-1}, s)$ will return (FALSE, FALSE). Thus at the end of the ith iteration we will have the correct values of c_i and d_i.

Since we start with the correct values of c_0 and d_0, we can say that the loop of line 5 terminates normally with the correct values of c_M and d_M. Now, if the vertex acc_w is present in the graph $\mathcal{G}_{\mathcal{M},x,w,h_k,wt}$ such that $\mathsf{dist}(start_w, acc_w) = i$, then $final$ is set to TRUE in the ith iteration of the loop of line 5 when REACHABLE$(h_k(acc_w), i, h_k, wt, c_{i-1}, d_{i-1}, s)$ is called and it returns (TRUE, TRUE). Following which we halt and **Accept** in line 22 after restoring the initial auxiliary content of $\mathcal{M}'$ by XORing it with $G(s)$. □

Algorithm 2 The REACHABLE procedure.

REACHABLE$(v, i, h_k, wt, c_{i-1}, d_{i-1}, s)$ is called only if **Condition A** and **Condition B** are satisfied. The procedure checks if there exists a $ver \in V(\mathcal{G}_{\mathcal{M},x,w,h_k,wt})$ such that $h_k(ver) = v$, $\mathsf{dist}(start_w, ver) = i$ and $ver = acc_w$.

procedure REACHABLE$(v, i, h_k, wt, c_{i-1}, d_{i-1}, s)$
  $d \leftarrow 0$, $h \leftarrow -1$, $v_{present} \leftarrow$ FALSE, $cnt \leftarrow 0$, $finalreach \leftarrow$ FALSE
  for $j = 1$ to c_{i-1} **do**
    Clean the workspace z for simulation of $\mathcal{M}$.
    Nondeterministically guess $l \leq i - 1$.
    Simulate $\mathcal{M}$ on (x, w) using z as workspace for l steps.
    if $h_k(z, w, pos, state) \leq h$ **then**
      Continue the simulation until a halting state is reached.
      $w \leftarrow w \oplus G(s)$
      Reject
    end if
    $h \leftarrow h_k(z, w, pos, state)$
    $d \leftarrow d + l$
    if $h = v$ **then**
      $v_{present} \leftarrow$ TRUE
    end if
    if $l = i - 1$ **then**
      $q =$ Number of configurations reachable from $(z, w, pos, state)$ in one step
      for $r = 1$ to q **do**
        Simulate one more step.
        if $h_k(z, w, pos, state) = v$ **then**
          $cnt \leftarrow cnt + 1$
          if $(z, w, pos, state) = acc_w$ **then**
            $finalreach \leftarrow$ TRUE
          end if
        end if
        Simulate a step back.
      end for
    end if
    Continue the simulation until a halting state is reached.
  end for
  if $d \neq d_{i-1}$ **then**
    $w \leftarrow w \oplus G(s)$
    Reject
  end if
  if $v_{present} =$ FALSE **then**
    if $cnt = 0$ **then** return (FALSE, FALSE)
    else if $cnt = 1$ **then** return (TRUE, $finalreach$)
    else if $cnt > 1$ **then** return (BAD, FALSE)
    end if
  else
    if $cnt = 0$ **then** return (FALSE, FALSE)
    else if BADGRAPH$(v, i, h_k, wt, c_{i-1}, d_{i-1}, s, cnt) =$ TRUE **then**
      return (BAD, FALSE)
    else return (FALSE, FALSE)
    end if
  end if
end procedure

Algorithm 3 The BADGRAPH procedure.

BADGRAPH$(v, i, h_k, wt, c_{i-1}, d_{i-1}, s, cnt)$ is called only if **Condition A** and **Condition B** are satisfied. The procedure checks if h_k maps $V(\mathcal{G}_{\mathcal{M},x,w,h_k,wt})$ to $\{0,1\}^{4\log n}$ injectively or not.

procedure BADGRAPH$(v, i, h_k, wt, c_{i-1}, d_{i-1}, s, cnt)$
  for $g = 1$ to cnt **do**
    for $t = 1$ to T **do**
      $d \leftarrow 0$, $h \leftarrow -1$, $bit1 \leftarrow 0$, $bit2 \leftarrow 0$, $cnt' \leftarrow 0$
      for $j = 1$ to c_{i-1} **do**
        Clean the workspace z for simulation of $\mathcal{M}$.
        Nondeterministically guess $l \leq i - 1$.
        Simulate $\mathcal{M}$ on (x, w) using z as workspace for l steps.
        if $h_k(z, w, pos, state) \leq h$ **then**
          Continue the simulation until a halting state is reached.
          $w \leftarrow w \oplus G(s)$
          Reject
        end if
        $h \leftarrow h_k(z, w, pos, state)$
        $d \leftarrow d + l$
        if $h = v$ **then**
          Store the tth bit of $(z, w, pos, state)$ in $bit1$.
        end if
        if $l = i - 1$ **then**
          $q =$ Number of configurations reachable from $(z, w, pos, state)$
          for $r = 1$ to q **do**
            Simulate one more step.
            if $h_k(z, w, pos, state) = v$ **then**
              $cnt' \leftarrow cnt' + 1$
              if $cnt' = g$ **then**
                Store the tth bit of $(z, w, pos, state)$ in $bit2$.
              end if
            end if
            Simulate a step back.
          end for
        end if
        Continue the simulation until a halting state is reached.
      end for
      if $d \neq d_{i-1}$ **then**
        $w \leftarrow w \oplus G(s)$
        Reject
      end if
      if $bit1 \neq bit2$ **then**
        return TRUE
      end if
    end for
  end for
  return FALSE
end procedure

Case 2 - Triplet $\langle s, h_k, s' \rangle$ is bad: A triplet $\langle s, h_k, s' \rangle$ is bad if

- **Violation I**: h_k does not injectively map the vertices of $\mathcal{G}_{\mathcal{M},x,w,h_k,wt}$ to $\{0,1\}^{4\log N}$.
- **Violation II**: $\mathcal{G}_{\mathcal{M},x,w,h_k,wt}$ is not min-unique.

We will show that if both violations occur simultaneously then Algorithm 1 moves to the next triplet without finishing all M iterations of the loop of line 5. The other cases where only one violation occurs can be analysed similarly.

Let ver_1 and ver_2 be two vertices of $\mathcal{G}_{\mathcal{M},x,w,h_k,wt}$ such that (1) $\mathsf{dist}(start_w, ver_1) \leq \mathsf{dist}(start_w, ver_2)$, (2) $h_k(ver_1) = h_k(ver_2)$, and (3) there does not exist any other pair of vertices say ver_3 and ver_4 such that $h_k(ver_3) = h_k(ver_4)$ and $\mathsf{dist}(start_w, ver_3) \leq \mathsf{dist}(start_w, ver_4) < \mathsf{dist}(start_w, ver_2)$. ver_1 and ver_2 exist due to **Violation I**.

Let ver be a vertex which has more than one minimum weight paths from $start_w$ such that there is no other vertex ver' with more than one minimum weight paths from $start_w$ and $\mathsf{dist}(start_w, ver') < \mathsf{dist}(start_w, ver)$. ver exists due to **Violation II**.

Let $\mathsf{dist}(start_w, ver_2) = i$ and $\mathsf{dist}(start_w, ver) = j$. First note that $i \leq M$, because if $i > M$ then the first $M+1$ vertices on the shortest path from $start_w$ to ver_2 are all injevtively mapped to $\{0,1\}^{4\log N}$ which is not possible because $M = N^4 = |\{0,1\}^{4\log N}|$.

Let $i \leq j$, then both **Condition A** and **Condition B** are satisfied for $C_{<i}$, therefore, the first $i-1$ iterations of the loop of line 5 will terminate normally with correct values of c_{i-1} and d_{i-1}. But on the ith iteration $\textsc{Reachable}(h_k(ver_2), i, h_k, wt, c_{i-1}, d_{i-1}, s)$ will return (BAD, FALSE) as **Condition I** or **II** are satisfied and Algorithm 1 will move on to the next triplet. The case of $j < i$ is similar. □

Finally, if $acc_w \notin \mathcal{G}_{\mathcal{M},x,w,h_k,wt}$ for any good triplet $\langle s, h_k, s' \rangle$ then the value of $final$ is never set to TRUE, therefore, after going over all triplets we **Reject** in line 25.

3.2 coCUL and an alternative proof of CNL = coCNL

Note that, if in line 22 of Algorithm 1 we **Reject** instead of **Accept** after finding an accepting node in the configuration graph for a good triplet $\langle s, h_k, s' \rangle$ and in line 25 we finally **Accept** instead of **Reject** after not finding the accepting node in any of the configuration graph for a good triplet $\langle s, h_k, s' \rangle$, then $L(\mathcal{M}') = \overline{L(\mathcal{M})}$. This proves that $\mathsf{coCNL} \subseteq \mathsf{CUL}(= \mathsf{CNL})$, which implies that $\mathsf{CUL} = \mathsf{CNL} = \mathsf{coCNL} = \mathsf{coCUL}$.

References

1 Eric Allender, Klaus Reinhardt, and Shiyu Zhou. Isolation, Matching, and Counting Uniform and Nonuniform Upper Bounds. *J. Comput. Syst. Sci.*, 59(2):164–181, October 1999. doi:10.1006/jcss.1999.1646.

2 Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a Full Memory: Catalytic Space. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 857–866, New York, NY, USA, 2014. ACM. doi:10.1145/2591796.2591874.

3 Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. Catalytic Space: Nondeterminism and Hierarchy. *Theory of Computing Systems*, 62(1):116–135, January 2018. doi:10.1007/s00224-017-9784-7.

4 Neil Immerman. Nondeterministic Space is Closed Under Complement. *SIAM Journal on Computing*, 17:935–938, 1988.

5 Russell Impagliazzo and Avi Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 220–229, New York, NY, USA, 1997. ACM. doi:10.1145/258533.258590.

6 Adam R. Klivans and Dieter van Melkebeek. Graph Nonisomorphism Has Subexponential Size Proofs Unless the Polynomial-Time Hierarchy Collapses. *SIAM J. Comput.*, 31(5):1501–1526, May 2002. doi:10.1137/S0097539700389652.

7 Klaus Reinhardt and Eric Allender. Making Nondeterminism Unambiguous. *SIAM J. Comput.*, 29(4):1118–1131, February 2000. doi:10.1137/S0097539798339041.

8 Robert Szelepcsényi. The Method of Forced Enumeration for Nondeterministic Automata. *Acta Informatica*, 26:279–284, 1988.