Electronic Colloquium on Computational Complexity, Report No. 343 (2026)

The Complexity of Unpredictability

Aryan Agarwala
Max-Planck-Institut für Informatik
aryan@agarwalas.in

Dani Dorfman
Max-Planck-Institut für Informatik
ddorfman@mpi-inf.mpg.de

Abstract

In this work, we initiate the study of middle-bit predictors: functions that take as input a string with exactly one missing bit and attempt to predict the value of that bit. These are closely related to next-bit predictors, a classical object in pseudorandomness.

We study the deterministic complexity of the problem of computing an unpredictable distribution (UNPRED): given as input a list of middle-bit predictors in circuit form and a parameter $\epsilon > 0$, output a multiset of strings such that each predictor succeeds with probability $< 1/2 + \epsilon$ on a uniformly sampled string from this multiset. We denote the version in which the predictors are restricted to be TC^1 circuits by $\mathsf{UNPRED}^{\mathsf{TC}^1}$.

We show that UNPRED offers a unified route towards resolving several major open problems in *catalytic computation*. The most important of these is understanding the relationship between catalytic logspace (CL and SearchCL) and deterministic polynomial time (P and FP). It is widely believed that $\mathsf{SearchCL} \subseteq \mathsf{FP}$. We prove the following implications:

$$\mathsf{UNPRED} \in \mathsf{FP} \implies \mathsf{SearchCL} \subseteq \mathsf{FP} \implies \mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{FP}.$$

The questions of whether $\mathsf{UNPRED} \in \mathsf{FP}$ and whether $\mathsf{SearchCL} \subseteq \mathsf{FP}$ are therefore almost equivalent, up to the restriction to TC^1 circuits.

Furthermore, we show that any deterministic polynomial-time algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ must exploit properties of Boolean TC^1 circuits, either in the algorithm itself or in its analysis. We are not aware of any such algorithm for any natural problem. Thus, a deterministic polynomial-time algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ would not only make progress in catalytic computation but would also advance our understanding of Boolean circuits.

Together, our results provide strong evidence that middle-bit predictors could be the key to unlocking several major open problems in catalytic computation, pseudorandomness, and Boolean circuit complexity.

ISSN 1433-8092

Contents

1 Introduction **1**
- 1.1 Catalytic Computation . . . 1
- 1.2 Our Results: UNPRED . . . 2
- 1.3 Catalytic Computation: Comparison with Past Work and Connections to Other Problems . . . 3
- 1.4 Connection to Pseudorandomness . . . 6
- 1.5 Our Results: NextBitUNPRED . . . 7
- 1.6 Future Directions and Open Problems . . . 9
- 1.7 Proof Overview . . . 10
- 1.8 Organization of the Paper . . . 16

2 Preliminaries **17**
- 2.1 Notation . . . 17
- 2.2 Standard Complexity Classes . . . 17
- 2.3 Predictors and UNPRED . . . 18
- 2.4 Catalytic and In-Place Computation . . . 20
- 2.5 Oracle Model . . . 22

3 Catalytic Logspace Algorithms for UNPRED **23**

4 Catalytic Logspace Reduces to UNPRED **28**
- 4.1 WeakExcess Reduces to UNPRED . . . 29
- 4.2 LossyCoding Reduces to WeakExcess . . . 33
- 4.3 SearchCL Reduces to UNPRED . . . 38

5 Lower Bounds for $\mathsf{UNPRED}^{\mathcal{O}_P}$ and $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ **39**
- 5.1 Lower Bound for $\mathsf{UNPRED}^{\mathcal{O}_P}$ . . . 40
- 5.2 Lower Bound for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ . . . 42

6 StableCL versus SearchCL **46**

7 Next-Bit Predictors, Pessimistic Estimators, and Graph Sparsification **49**
- 7.1 Computing Approximate Solutions to Pessimistic Estimator Instances Reduces to NextBitUNPRED . . . 51
- 7.2 Graph Sparsification Reduces to $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1}$ . . . 53
- 7.3 NextBitUNPRED Reduces to Computing Approximate Solutions to Pessimistic Estimator Instances . . . 54

8 BlockLossyCoding Reduces to UNPRED **58**

9 Artificial Intelligence Disclosure **62**

Acknowledgements **63**

References **64**

A Online Hypercube Graph Balancing Problem **68**

B Online Prefix-Tree Balancing Problem **69**

C A Uniform Version of UNPRED **70**

1 Introduction

In this work, we initiate the study of middle-bit predictors. A middle-bit predictor over n-bit strings at an index $i \in [n]$ is a function $P : \{0,1\}^{n-1} \to \{0,1\}$. P is said to *succeed* on a string $s \in \{0,1\}^n$ if, given as input the entirety of s except its ith bit, P guesses the ith bit of s correctly. Middle-bit predictors are closely related to next-bit predictors, which are a classical object in pseudorandomness, going back to the work of Blum and Micali [BM84] and Yao [Yao82].

Understanding the strength of middle-bit predictors was explicitly posed as an open problem by Pyne and Tell in their recent survey on derandomizing low-space algorithms [PT26b]. The recent independent work of Doron, Moshkovitz, Oh, and Zuckerman [DMOZ26] also studies middle-bit predictors in a slightly different setting; we describe their work in more detail later.

In this work, we define the problem UNPRED: given as input a list of m middle-bit predictors[1] over $\{0,1\}^n$ and a parameter $0 < \epsilon < 1/2$, output a multiset $S \subseteq \{0,1\}^n$ such that each of these m predictors succeeds with probability less than $1/2 + \epsilon$ on a string s sampled uniformly at random from S. Observe that a random multiset S of size $\Theta(\log m/\epsilon^2)$ is a valid output with constant probability; thus, UNPRED is easy with randomized algorithms, and in this work we seek deterministic algorithms.

The motivation for our work comes from two separate but deeply related fields: *catalytic computation* and *pseudorandomness.*

1.1 Catalytic Computation

The framework of catalytic computation was introduced by Buhrman, Cleve, Koucký, Loff, and Speelman [BCK+14] in order to study the power of space that contains important but irrelevant information. In this model a space-bounded Turing machine is given access to a much longer read-write 'catalytic' tape. The catalytic tape initially contains an arbitrary string chosen by an adversary, and upon termination it must contain *exactly* the same string.

Catalytic computation has recently been at the center of several breakthroughs in space-bounded complexity. Examples include the Cook-Mertz algorithm for the *tree evaluation problem* [CM24], Williams' simulation of time using low space [Wil25], and the OpenAI preprint claiming a proof that $\mathsf{L} = \mathsf{RL} = \mathsf{BPL}$ [Ope26] [2]. Furthermore, techniques from catalytic computation have also recently been applied to *hardness versus randomness* results in low-space derandomization [DPT24, LPT24, DPTW25, PT26a]. We refer interested readers to the survey of Pyne and Tell [PT26b]. Thus, obtaining a complete understanding of catalytic computation could be crucial to future progress in space-bounded complexity theory.

[BCK+14] introduced the class *catalytic logspace* (CL), in which a logspace machine is augmented with a polynomial-length catalytic tape. Surprisingly, they gave strong evidence that CL is significantly more powerful than ordinary logspace (L), by showing that a CL machine can simulate log-depth threshold circuits (TC^1). TC^1 is believed to be stronger than L, and in particular contains nondeterministic logspace (NL). On the other hand, while L is contained in deterministic polynomial time (P), [BCK+14] were unable to prove that $\mathsf{CL} \subseteq \mathsf{P}$. Instead, they showed that any catalytic logspace algorithm can be simulated by zero-error *randomized* polynomial-time algorithms (ZPP).

[1] Each predictor is given as a Boolean circuit, along with the index that it is predicting as an integer from 1 to n.

[2] At this point it is unclear whether there is a human who understands OpenAI's proof of $\mathsf{L} = \mathsf{RL} = \mathsf{BPL}$. However, catalytic computation seems to feature prominently in the preprint.

Theorem 1.1 ([BCK+14])**.**

$$\mathsf{TC}^1 \subseteq \mathsf{CL} \subseteq \mathsf{ZPP}.$$

The following problem naturally arises from their work.

Problem 1.2 ([Kou16, Mer23, CLMP25])**.**

$$\text{Is } \mathsf{CL} \subseteq \mathsf{P}?$$

Standard derandomization assumptions [IW97] would imply that $\mathsf{P} = \mathsf{ZPP}$, and thus $\mathsf{CL} \subseteq \mathsf{P}$. Therefore, Problem 1.2 should be viewed as a derandomization problem. We show that the problem of designing a deterministic polynomial-time (FP) algorithm for UNPRED is almost equivalent to the search analogue of Problem 1.2 ($\mathsf{SearchCL} \subseteq \mathsf{FP}$).

1.2 Our Results: UNPRED

1.2.1 SearchCL $\subseteq$ FP and UNPRED $\in$ FP Are Almost Equivalent

Our first result concerns an oracle version of UNPRED, which we call $\mathsf{UNPRED}^{\mathcal{O}_P}$. In this model, the predictors are not explicitly given to the algorithm as circuits, but can only be queried as black-box oracles (see Section 2.5 for the catalytic oracle model). We show that there exists a catalytic logspace algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$ in this oracle model.

Theorem 1.3 (Theorem 3.3)**.**

$$\mathsf{UNPRED}^{\mathcal{O}_P} \in \mathsf{SearchCL}^{\mathcal{O}_P}.$$

A corollary of this theorem is that the version of UNPRED where the predictors are explicitly given as input in the form of TC^1 circuits can be solved in $\mathsf{SearchCL}$. We refer to this problem as $\mathsf{UNPRED}^{\mathsf{TC}^1}$.

Corollary 1.4 (Corollary 3.4)**.**

$$\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{SearchCL}.$$

This corollary follows by simply replacing each oracle query in the algorithm of Theorem 1.3 with an explicit evaluation of the TC^1 circuit of the predictor, using the CL algorithm of Buhrman, Cleve, Koucký, Loff, and Speelman [BCK+14] to do so.

Corollary 1.4 shows that a deterministic polynomial-time algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ is necessary for establishing $\mathsf{SearchCL} \subseteq \mathsf{FP}$. The following theorem shows that a deterministic polynomial-time algorithm for UNPRED is sufficient.

Theorem 1.5.

$$\mathsf{SearchCL} \leq_{\mathsf{P}} \mathsf{UNPRED}.$$

Thus, Corollary 1.4 and Theorem 1.5 together place the search version of the CL versus P problem between two versions of UNPRED:

$$\mathsf{UNPRED} \in \mathsf{FP} \implies \mathsf{SearchCL} \subseteq \mathsf{FP} \implies \mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{FP}.$$

It is very surprising to us that this weak equivalence is possible, since middle-bit predictors have no superficial connection to the catalytic condition of restoring space.

Of course, there is a gap between the two ends of this chain: one uses $\mathsf{UNPRED}^{\mathsf{TC}^1}$, while the other uses UNPRED. This gap can be closed in two ways. First, one can define a uniform version of

UNPRED where the predictors are given as input in the form of polynomial-time Turing machines with certain complicated properties. One can then prove both Corollary 1.4 and Theorem 1.5 using this complicated version of UNPRED, eliminating the gap entirely. An older draft of this paper indeed followed this approach. However, we concluded that this approach is overly complicated, and obfuscates the key technical arguments that make this reduction possible [3]. In Section C, we define this version and state the corresponding theorems without proof. Second, we can actually show that all problems *currently* known to be in CL (or SearchCL) reduce directly to $\mathsf{UNPRED}^{\mathsf{TC}^1}$. We will do this in Sections 7 and 8.

Our takeaway from this set of results is straightforward: any future work on resolving the search analogue of Problem 1.2 can ignore the mysterious catalytic condition of restoring space, and simply try to design a deterministic polynomial-time algorithm for UNPRED. Such an algorithm would immediately imply $\mathsf{SearchCL} \subseteq \mathsf{FP}$ and thus $\mathsf{CL} \subseteq \mathsf{P}$. A natural starting point for this would be to design a deterministic polynomial-time algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$, which is a necessary condition.

1.2.2 A Deterministic Polynomial-Time Algorithm for UNPRED Must Exploit Properties of Boolean Circuits

By Theorem 1.5, an FP algorithm for UNPRED would suffice to prove that $\mathsf{SearchCL} \subseteq \mathsf{FP}$ and $\mathsf{CL} \subseteq \mathsf{P}$. The most direct approach towards this would be to design a deterministic polynomial-time oracle algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$, similar to the catalytic logspace algorithm of Theorem 1.3. Replacing each oracle query in such an algorithm with circuit evaluations would imply that $\mathsf{UNPRED} \in \mathsf{FP}$ and thus $\mathsf{CL} \subseteq \mathsf{P}$. Unfortunately, the following theorem shows that this is unconditionally impossible.

Theorem 1.6 (Corollary 5.4)**.** *Every deterministic oracle algorithm for* $\mathsf{UNPRED}^{\mathcal{O}_P}$*, on instances with* n *predictors over* $\{0,1\}^n$ *and* $\epsilon = 1/10$*, has worst-case running time* $\exp(\Omega(\sqrt{n}))$*. In particular,*

$$\mathsf{UNPRED}^{\mathcal{O}_P} \notin \mathsf{P}^{\mathcal{O}_P}.$$

This shows that any deterministic algorithm for UNPRED or $\mathsf{UNPRED}^{\mathsf{TC}^1}$ which treats the input predictors purely as functions must take exponential time in the worst case. Therefore, any FP algorithm would have to exploit properties of Boolean circuits, either in the algorithm itself or in its analysis. We are not aware of any deterministic polynomial-time algorithm for any comparable problem which is able to exploit such properties of large circuit classes (as large as TC^1). Therefore, a deterministic polynomial-time algorithm for UNPRED or $\mathsf{UNPRED}^{\mathsf{TC}^1}$ would not only make substantial progress on (or completely resolve) Problem 1.2, but would also significantly advance our understanding of Boolean circuits.

A more pessimistic interpretation of this result would be that this oracle barrier is too strong, and thus $\mathsf{SearchCL} \subseteq \mathsf{FP}$ is not solvable with current techniques.

Both interpretations solidify $\mathsf{UNPRED}^{\mathsf{TC}^1}$ as the eventual starting point towards attacking Problem 1.2.

1.3 Catalytic Computation: Comparison with Past Work and Connections to Other Problems

1.3.1 Comparison with Past Work

There are two papers that we would like to compare our work with:

[3] If a reader is interested in this older draft, we would be happy to share it.

- Cook, Ghentiyala, Mertz, Pyne, and Sheffield [CGM+25] gave an oracle O relative to which $\mathsf{CL}^O \not\subseteq \mathsf{P}^O$, ruling out relativizing proofs of $\mathsf{CL} \subseteq \mathsf{P}$. Theorems 1.3 and 1.6 in our work can also be interpreted as a relativization barrier. However, we believe that ours is a more *meaningful* barrier, since it identifies a precise problem, UNPRED, that requires substantially different techniques to be solved in CL and in P. More importantly, our barrier gives a precise way forward: a polynomial-time algorithm for UNPRED with explicitly given predictor circuits would imply $\mathsf{CL} \subseteq \mathsf{P}$.

- Cook, Li, Mertz, and Pyne [CLMP25] show that SearchCL reduces in polynomial time to LossyCoding: the problem of finding a string that a compressor and decompressor circuit fail to accurately reconstruct (see Definition 4.9). Since $\mathsf{UNPRED} \leq_{\mathsf{P}} \mathsf{LossyCoding}$ (by Lemma 3.2), Theorem 1.5 is a strengthening of their result. To see why this strengthening matters, imagine a universe where CL can evaluate any circuit. That is, $\mathsf{P} \subseteq \mathsf{CL}$. Then Theorems 1.3 and 1.5 would imply that UNPRED is SearchCL-complete under polynomial-time reductions. In contrast, even in this extreme case, it is unclear whether $\mathsf{LossyCoding} \in \mathsf{SearchCL}$. More generally, for any circuit class $\mathcal{C}$ such that $\mathcal{C} \subseteq \mathsf{CL}$, Theorem 1.3 implies that $\mathsf{UNPRED}^{\mathcal{C}} \in \mathsf{SearchCL}$, whereas this is entirely unclear for LossyCoding. Thus, UNPRED more accurately captures the strength of CL.

1.3.2 CL versus NC

As stated earlier, we view $\mathsf{CL} \subseteq \mathsf{P}$ (Problem 1.2) as a derandomization problem. Such a viewpoint suggests that CL may itself be able to simulate certain randomized algorithms deterministically. Pyne [Pyn24] and Cook, Li, Mertz, and Pyne [CLMP25] demonstrated this derandomization ability of CL through a technique called *compress-or-random*. At a high level, the idea is that a CL algorithm could use its catalytic tape as the random string for a zero-error randomized algorithm. If the catalytic tape is not a good random string for the randomized algorithm, then the CL algorithm could compress the string on the catalytic tape and create free space. This win-win argument was used to develop an understanding of the structure of catalytic computing [CLMP25, KMPS25, Pyn24]. However, these works did not place a 'new' problem in CL. That is, at this point, no problem was known to be in CL beyond the original TC^1 inclusion of [BCK+14]. TC^1 is contained in the class of problems solvable in parallel with polylogarithmic depth and polynomial work (NC). This led to the following question.

Problem 1.7 ([Kou16, Mer23])**.**

Is $\mathsf{CL} \subseteq \mathsf{NC}$?

Unlike Problem 1.2, there was no reason to believe that this is true, except for the fact that all problems known to be in CL were contained in NC. Agarwala and Mertz [AM25] broke this barrier by showing that bipartite maximum matching can be solved in CL. Agarwala, Alekseev, and Vinciguerra [AAV26] then showed that linear matroid intersection can be solved in CL. Arvind, Chakraborty, and Datta [ACD26] used the same techniques to show that the exact matching problem on planar graphs can be solved in CL. Chakraborty, Datta, Kusre, Mukhopadhyay, and Sinhababu [CDK+26] generalized the bipartite matching algorithm of [AM25] and showed that the maximum matching problem on general graphs can also be solved in CL. Finally, Marcussen, Pyne, and Rubinfeld [MPR26] recently showed that the graph spectral sparsification problem can be solved in CL.

At the time, none of these problems were known to be in NC. Thus, these papers presented their algorithms as barriers towards proving that $\mathsf{CL} \subseteq \mathsf{NC}$ (Problem 1.7). Any proof of $\mathsf{CL} \subseteq \mathsf{NC}$ would immediately imply that all of these problems are contained in NC. For many of these problems, this was a longstanding open question. In a recent breakthrough result, Chatterjee, Ghosh, Gurjar, Raj, and Thierauf [CGG$^+$26] showed that bipartite matching and linear matroid intersection can be solved in NC, destroying the barriers of [AM25, AAV26]. Unfortunately, their algorithm also did not give any insight about how to tackle Problem 1.7. This example shows that barriers to Problems 1.2 and 1.7 based on individual graph problems can be fickle, and if solved do not necessarily provide any insight into the strength of CL. Instead, we will give a barrier to Problem 1.7 which we believe is both stronger and more meaningful than these graph problems. This barrier will be based on a subproblem of UNPRED.

1.3.3 Search Versus Decision for Catalytic Logspace

Many of our theorems concern the search analogues of Problem 1.2 and Problem 1.7. This naturally raises the following question: are the search and decision versions of catalytic logspace equivalent? For example, if one were to obtain the very strong result that $\mathsf{L} = \mathsf{CL}$, would it then imply that $\mathsf{FL} = \mathsf{SearchCL}$? Formally,

Problem 1.8.

$$\text{Is } \mathsf{FL}^{\mathsf{CL}} = \mathsf{SearchCL}?$$

Readers familiar with complexity theory may notice that this is similar to the question of whether $\mathsf{SearchBPP}$ equals $\mathsf{FP}^{\mathsf{BPP}}$. It is known that $\mathsf{FP}^{\mathsf{BPP}}$ is equal to the class of search problems solvable by polynomial time *pseudodeterministic* algorithms [GG11]. These are randomized algorithms which, for any input, output the *same* solution with high probability. These algorithms remove the additional instability in the output of search problems caused by randomness.

In our work, we define $\mathsf{StableCL}$ to be the class of search problems solvable by catalytic logspace algorithms which, for any input, output the *same* solution for all initial catalytic tape configurations. Thus, a stable algorithm removes the additional instability in the output of search problems caused by the catalytic tape. We prove that $\mathsf{StableCL}$ has the same relationship with FL and CL that $\mathsf{PseudodeterministicBPP}$ has with FP and BPP.

Theorem 1.9 (Corollary 6.5)**.**

$$\mathsf{FL}^{\mathsf{CL}} = \mathsf{StableCL}.$$

Therefore, Problem 1.8 is equivalent to the following.

Problem 1.10.

$$\text{Is } \mathsf{StableCL} = \mathsf{SearchCL}?$$

The $\mathsf{SearchCL}$ algorithms for bipartite matching [AM25], general matching [CDK$^+$26], linear matroid intersection [AAV26], and graph sparsification [MPR26] are *not* stable, and it is unclear whether these algorithms can be made stable.

We prove in Sections 7 and 8 that all four of these $\mathsf{SearchCL}$ algorithms can be interpreted as TC^1 reductions to $\mathsf{UNPRED}^{\mathsf{TC}^1}$. Therefore, the instability in these algorithms is due to the fact that our $\mathsf{SearchCL}$ algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ (Corollary 1.4) is not stable.

It turns out that there is a good reason for this. Our $\mathsf{SearchCL}$ algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ has a special property: there is a point in the algorithm where its entire output is explicitly

written on the catalytic tape. We call such algorithms *restorable*. In Corollary 6.9, we show *unconditionally* that $\mathsf{UNPRED}^{\mathsf{TC}^1}$ does not have a restorable catalytic logspace algorithm which is additionally stable. Thus, a $\mathsf{StableCL}$ algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ (which is a necessary condition for resolving Problem 1.10 positively) would require substantially new ideas: in particular, it must not be restorable. We emphasize that restorability is a natural by-product of the compress-or-random technique, and is not an artificial feature of our algorithm. We elaborate significantly in Section 6.

Thus, UNPRED provides a unified route towards resolving three major questions related to catalytic computation: P versus CL (Problem 1.2), NC versus CL (Problem 1.7), and $\mathsf{SearchCL}$ versus $\mathsf{StableCL}$ (Problem 1.10).

1.4 Connection to Pseudorandomness

A *next-bit predictor* over n-bit strings at an index $i \in [n]$ is a function $P : \{0,1\}^{i-1} \to \{0,1\}$. P is said to *succeed* on a string $s \in \{0,1\}^n$ if, given as input the length $i-1$ prefix of s, it produces as output the ith bit of s. Next-bit predictors are a classical object in pseudorandomness, going back to the work of Blum and Micali [BM84] and Yao [Yao82]. We refer to the version of UNPRED where the input consists of next-bit predictors, instead of middle-bit predictors, as $\mathsf{NextBitUNPRED}$. Observe that every next-bit predictor is also a middle-bit predictor which simply ignores the suffix of its input after the index i. Thus, middle-bit predictors are a stronger object than next-bit predictors.

A central tool in the hardness-to-randomness paradigm [Yao82, BM84, NW94, IW97], which aims to show that $\mathsf{BPP} = \mathsf{P}$, is a *distinguish-to-predict* (D2P) transformation. Yao's classical distinguish-to-predict lemma [Yao82] constructs, for any randomized polynomial-time algorithm $\mathcal{A}$ and input $x \in \{0,1\}^n$, a list of next-bit predictors $\mathcal{P}$ and a parameter $\epsilon \geq 1/\mathrm{poly}(|x|)$ such that any polynomial-size solution to $\mathsf{NextBitUNPRED}$ on $(\mathcal{P}, \epsilon)$ would suffice to derandomize $\mathcal{A}$ on input x. This argument *almost* reduces the problem of derandomizing $\mathcal{A}$ to $\mathsf{NextBitUNPRED}$, except for the fact that Yao's argument is *probabilistic*, and a naive derandomization of his argument results in exponentially many next-bit predictors. Li, Pyne, and Tell [LPT24] provide an alternative argument that results in only polynomially many next-bit predictors; however, their construction of these next-bit predictors is non-uniform.

It makes intuitive sense that this reduction, which we call *next-bit D2P*, is so difficult: Goldreich and Wigderson [GW00] and Li, Pyne, and Tell [LPT24] showed that $\mathsf{NextBitUNPRED}$ has a deterministic polynomial-time algorithm [4]. Thus, an efficient next-bit D2P algorithm would simply derandomize $\mathcal{A}$. Li, Pyne, and Tell [LPT24] formalize this framework of derandomization: for a fixed algorithm $\mathcal{A}$, given as input $x \in \{0,1\}^*$, first solve next-bit D2P, and then solve $\mathsf{NextBitUNPRED}$. In this framework, however, the second step is easy: the deterministic polynomial-time algorithms for $\mathsf{NextBitUNPRED}$ in [GW00, LPT24] are fairly straightforward and do not use any technical machinery. Thus, all of the hardness lies in the first step: next-bit D2P.

We believe that middle-bit predictors may offer a more promising route towards derandomization. Consider the same framework as above, but with middle-bit predictors instead of next-bit predictors. This would reduce derandomization to two problems: *middle-bit D2P* and UNPRED. Since middle-bit predictors are more powerful than next-bit predictors, UNPRED should be harder than $\mathsf{NextBitUNPRED}$, and thus middle-bit D2P should be easier than next-bit D2P. Intuitively, this framework with middle-bit predictors seems to be more balanced than the framework with next-bit predictors.

[4] In fact, they gave deterministic polynomial-time algorithms for the oracle version $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$.

Our results support this conclusion: Theorem 1.6 rules out deterministic polynomial-time oracle algorithms for $\mathsf{UNPRED}^{\mathcal{O}_P}$. This is in sharp contrast to $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$, which has simple deterministic polynomial-time oracle algorithms due to Goldreich and Wigderson [GW00] and Li, Pyne, and Tell [LPT24]. Thus, middle-bit predictors are unconditionally stronger than next-bit predictors in the oracle versions of the problems. Similarly, Theorem 1.5 is precisely a middle-bit D2P for $\mathsf{SearchCL}$ algorithms, whereas a next-bit D2P for $\mathsf{SearchCL}$ is not known, and would imply that $\mathsf{SearchCL} \subseteq \mathsf{FP}$. Finally, we note that the smallest class of polynomial-time randomized algorithms that has been formally studied in the context of derandomization, as far as we are aware, is FLOSSY [Kor22, CLMP25]. We do not define this class here, but we note that UNPRED belongs to this class, since $\mathsf{UNPRED} \leq_{\mathsf{P}} \mathsf{LossyCoding}$. Thus, while Theorems 1.6 and 1.5 show that UNPRED is harder than $\mathsf{NextBitUNPRED}$, UNPRED lies in the smallest class of derandomization problems that has been studied, which shows that it is not unreasonably hard either.

The recent work of Doron, Moshkovitz, Oh, and Zuckerman [DMOZ26] also supports our conclusion. As far as we are aware, this is the only other work that studies middle-bit predictors. They give a middle-bit D2P for a class of randomized algorithms that they call 'insensitive algorithms' with parameters that improve on the hybrid argument. A long line of work on 'beating the hybrid argument' has aimed to achieve such an improvement for next-bit predictors [FSUV12, DMOZ22, CT21, SV22]. Thus, their paper, along with ours, provides promising evidence that middle-bit predictors warrant further study in the context of pseudorandomness.

In this work, in addition to the new problem UNPRED, we also study the classical problem $\mathsf{NextBitUNPRED}$, and provide a complete characterization of it.

1.5 Our Results: $\mathsf{NextBitUNPRED}$

1.5.1 Sequential Algorithms Are Optimal for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$

The deterministic $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ algorithms of [GW00, LPT24] are inherently sequential: they construct the output strings one bit at a time, using n rounds of queries. It remained open whether there are deterministic parallel algorithms for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$. We answer this question negatively, showing that sequential algorithms are essentially optimal for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$.

Theorem 1.11 (Corollary 5.8)**.** *Let $0 < \delta \leq 1$. Every deterministic parallel oracle algorithm for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ of depth $O(n^{1-\delta})$, on instances with n predictors over $\{0,1\}^n$ with $\epsilon = 1/10$, has worst-case work $\exp(\Omega(n^{\delta}))$. In particular,*

$$\mathsf{NextBitUNPRED}^{\mathcal{O}_P} \notin \mathsf{NC}^{\mathcal{O}_P}.$$

Now, while a $\mathsf{SearchCL} \subseteq \mathsf{FP}$ proof would have to show that $\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{FP}$, a $\mathsf{SearchCL} \subseteq \mathsf{NC}$ proof would have to additionally show that $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1} \in \mathsf{NC}$. This gives a new barrier towards proving the search analogue of $\mathsf{CL} \subseteq \mathsf{NC}$ (Problem 1.7). Crucially, $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1} \in \mathsf{NC}$ is incomparable to the problem of $\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{FP}$: UNPRED is a harder problem than $\mathsf{NextBitUNPRED}$, but the parallel model is more restrictive than the sequential one. Therefore, future work on Problems 1.2 and 1.7 could start with trying to design an NC algorithm for $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1}$. Since next-bit predictors are classically studied, this problem may be more approachable than $\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{FP}$. Furthermore, a positive solution to this problem would give the first algorithm for $\mathsf{NextBitUNPRED}$ or UNPRED which exploits properties of circuits. Thus, this would likely give valuable insight towards proving that $\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{FP}$.

1.5.2 NextBitUNPRED Is Equivalent to Computing Approximate Solutions to Pessimistic Estimator Instances

The SearchCL algorithms for bipartite matching [AM25], linear matroid intersection [AAV26], and general matching [CDK+26] are very similar to one another. In particular, these three algorithms use the compress-or-random framework in a manner which allows the compression and decompression procedures to be implemented in NC. This was specifically noted by Agarwala and Mertz [AM25].

However, the SearchCL algorithm for graph sparsification by Marcussen, Pyne, and Rubinfeld [MPR26] is quite different from these. It is unclear whether the compression and decompression subroutines in [MPR26] can be implemented in NC. This is explicitly listed as an open problem in their work[5].

Their algorithm works by defining a TC^1-computable *pessimistic estimator* over strings of length $\text{poly}(n)$, and then computing an approximate solution to this pessimistic estimator instance. We refer readers unfamiliar with pessimistic estimators to Section 7 for a definition. For now, it suffices to know that pessimistic estimators, introduced by Raghavan [Rag88], are a generalization of the method of conditional expectation (also known as the method of conditional probabilities [ES73, Spe87]). It is a classical technique in derandomization and can be found in various textbooks, such as the ones by Alon and Spencer [AS16] and by Motwani and Raghavan [MR95].

We show that computing an approximate solution to a pessimistic estimator instance reduces to NextBitUNPRED, and vice versa. Both reductions involve only minor changes in the parameters: the circuits grow polynomially in size and by a constant factor in depth. Thus, NextBitUNPRED is equivalent to the algorithmic task underlying this classical technique.

Theorem 1.12 (Informal; Theorem 7.6, Corollary 7.12, and Observation 7.13)**.** NextBitUNPRED *is equivalent to computing approximate solutions to pessimistic estimator instances.*

Applying this theorem to the pessimistic estimator instance of Marcussen, Pyne, and Rubinfeld [MPR26] immediately gives the following corollary.

Corollary 1.13 (Theorem 7.8)**.**

$$\mathsf{GraphSparsification} \leq_{\mathsf{L\text{-}uniform}\ \mathsf{TC}^1} \mathsf{NextBitUNPRED}^{\mathsf{TC}^1}.$$

$\mathsf{UNPRED}^{\mathsf{TC}^1}$ (and hence also $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1}$) can be solved by a compress-or-random procedure in which both the compression and the decompression can be implemented in logspace-uniform TC^1 (and hence in NC). We do not state this formally, but it follows from the observation that our compression and decompression procedures in Section 3 (Lemma 3.2) can be implemented in TC^1 when the predictors are TC^1 circuits. Together with Corollary 1.13, this answers the open question posed by Marcussen, Pyne, and Rubinfeld [MPR26]. This is another example of how NextBitUNPRED and UNPRED can shed light on catalytic computation.

We summarize our results in Tables 1, 2, and 3. We omit the results related to the StableCL versus SearchCL problem (Problem 1.10), since they require additional definitions in order to be accurately stated. This will be done in Section 6.

[5]They state this question as whether GraphSparsification reduces to LossyCoding over NC circuits. We define and discuss the LossyCoding problem later in Definition 4.9.

Table 1: Characterization of UNPRED and NextBitUNPRED.

UNPRED	Informal: UNPRED is polynomial time equivalent to SearchCL, up to circuit evaluation. $\mathsf{UNPRED}^{\mathcal{O}_P} \in \mathsf{SearchCL}^{\mathcal{O}_P}$ and $\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{SearchCL} \leq_{\mathsf{P}} \mathsf{UNPRED}$
NextBitUNPRED	Informal: NextBitUNPRED is logspace equivalent to computing approximate solutions to pessimistic estimator instances.

Table 2: All problems currently known to be in SearchCL reduce to $\mathsf{UNPRED}^{\mathsf{TC}^1}$.

Reduction
$\mathsf{BipMATCH} \leq_{\mathsf{L\text{-}uniform}\ \mathsf{TC}^1} \mathsf{UNPRED}^{\mathsf{TC}^1}$
$\mathsf{LinearMatroidIntersection} \leq_{\mathsf{L\text{-}uniform}\ \mathsf{TC}^1} \mathsf{UNPRED}^{\mathsf{TC}^1}$
$\mathsf{MATCH} \leq_{\mathsf{L\text{-}uniform}\ \mathsf{TC}^1} \mathsf{UNPRED}^{\mathsf{TC}^1}$
$\mathsf{GraphSparsification} \leq_{\mathsf{L\text{-}uniform}\ \mathsf{TC}^1} \mathsf{NextBitUNPRED}^{\mathsf{TC}^1}$

Table 3: Oracle hardness of UNPRED and NextBitUNPRED.

	Lower bound	**Separation**
$\mathsf{UNPRED}^{\mathcal{O}_P}$	Deterministic algorithm requires $\exp(\Omega(\sqrt{n}))$ time.	$\mathsf{UNPRED}^{\mathcal{O}_P} \notin \mathsf{P}^{\mathcal{O}_P}$
$\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$	Deterministic parallel algorithm with depth $O(n^{1-\delta})$ requires $\exp(\Omega(n^{\delta}))$ work.	$\mathsf{NextBitUNPRED}^{\mathcal{O}_P} \notin \mathsf{NC}^{\mathcal{O}_P}$

Bounds are worst-case, for deterministic oracle algorithms with n predictors over $\{0,1\}^n$, $\epsilon = 1/10$, and fixed $0 < \delta \leq 1$.

1.6 Future Directions and Open Problems

We believe that our contribution in this work is providing strong evidence that UNPRED (and middle-bit predictors generally) warrants further study, specifically because it may hold the key to solving several major open problems in catalytic computation and pseudorandomness. In this section we discuss directions that future work should explore.

One ambitious goal to pursue is the following.

Problem 1.14. Is $\mathsf{UNPRED} \in \mathsf{FP}$?

A positive solution to this problem would immediately imply $\mathsf{CL} \subseteq \mathsf{P}$ and $\mathsf{SearchCL} \subseteq \mathsf{FP}$ (Problem 1.2). By Theorem 1.6, any deterministic polynomial-time UNPRED algorithm would need to avoid the oracle barrier of $\mathsf{UNPRED}^{\mathcal{O}_P} \notin \mathsf{P}^{\mathcal{O}_P}$. Thus, we begin by asking a simpler question.

Problem 1.15. Is $\mathsf{UNPRED}^{\mathsf{NC}^1} \in \mathsf{FP}$?

A positive solution to this weaker problem would provide the first UNPRED algorithm that does not simply treat the predictors as black-box functions. In particular, it would need to use

properties of Boolean circuits. It is likely that this would provide a substantial clue about how to solve Problem 1.14.

Similarly, future work could focus on studying parallel algorithms for NextBitUNPRED. As we said earlier, these would be incomparable to sequential algorithms for UNPRED.

Problem 1.16. Is $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1} \in \mathsf{NC}$?

A positive solution to this problem would immediately imply several NC inclusions which are not known, such as $\mathsf{GraphSparsification} \in \mathsf{NC}$ (due to our reduction from Section 7.2).

Again, it would also be valuable to solve Problem 1.16 for weak circuits. This would serve as a blueprint to overcome the oracle lower bound from Theorem 1.11.

Problem 1.17. Is $\mathsf{NextBitUNPRED}^{\mathsf{NC}^1} \in \mathsf{NC}$?

Note that a positive solution to the search analogue of Problem 1.2 would require first solving Problem 1.15. Similarly, a positive solution to the search analogue of Problem 1.7 would require first solving Problem 1.17.

Now, we state a problem related to the $\mathsf{FL}^{\mathsf{CL}}$ versus SearchCL problem (Problem 1.8).

Problem 1.18. Is $\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{StableCL}$?

A positive solution to this problem would immediately give StableCL algorithms for *every* problem currently known to be in SearchCL. Corollary 6.9 shows that a StableCL algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ cannot be restorable. Thus, we state the following subproblem.

Problem 1.19. Construct a non-restorable SearchCL algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$.

Of course, any algorithm can be made intentionally non-restorable. The spirit of this problem is to find an algorithm that cannot be made restorable.

Finally, we look at future directions in pseudorandomness. In our work we show an efficient middle-bit D2P for SearchCL algorithms. Doron, Moshkovitz, Oh, and Zuckerman [DMOZ26] show a middle-bit D2P for insensitive algorithms that beats the hybrid argument. In both cases, a next-bit D2P seems hard. Future work should explore how far these new predictors can go.

Problem 1.20. Construct efficient middle-bit D2Ps for classes of algorithms.

Here, efficiency refers to both computational efficiency, as in our work, and parameter efficiency, as in [DMOZ26]. Since $\mathsf{UNPRED}^{\mathcal{O}_P} \in \mathsf{SearchCL}^{\mathcal{O}_P}$, positive solutions to this problem may place new problems in SearchCL.

1.7 Proof Overview

We now give a brief overview of our main ideas. Each section additionally begins with a far more detailed discussion.

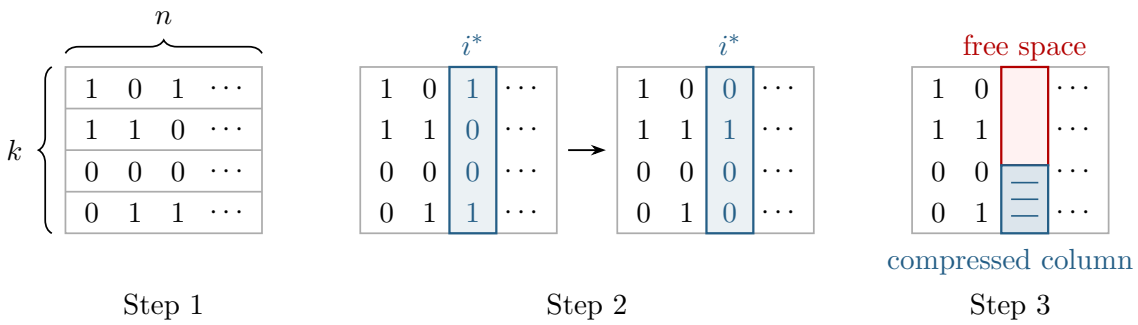

Figure 1: A compress-or-random algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$. Step 1: interpret the catalytic tape as a multiset S of k strings in $\{0,1\}^n$. Step 2: If there is a predictor P^* at index i^* which succeeds on too many strings, replace the $(i^*)^{th}$ bit of each string s with 0 if P^* succeeds on s, and with 1 otherwise. This makes the $(i^*)^{th}$ column sparse. Step 3: compress the sparse string in the $(i^*)^{th}$ column.

Catalytic Logspace Algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$. Our catalytic logspace algorithm for UNPRED in the oracle model, i.e., $\mathsf{UNPRED}^{\mathcal{O}_P}$, follows the *compress-or-random* framework briefly described in the introduction. The idea is as follows: the algorithm interprets the catalytic tape itself as a multiset of $k = \Theta(\log(2m)/\epsilon^2)$ n-bit strings. First, the algorithm checks whether this multiset is a valid output, by counting, for each predictor, the number of strings on which it succeeds. If every count is below $(1/2+\epsilon)k$, the algorithm outputs the multiset and halts. Otherwise, some predictor P^*, at an index $i^* \in [n]$, succeeds on too many strings. The algorithm replaces the $(i^*)^{th}$ bit of each string with 0 if P^* succeeds on the string, and 1 otherwise. The amount of information on the catalytic tape is exactly the same as before, since the $(i^*)^{th}$ bit of each string can be recovered by evaluating P^* on the string and flipping its output if the bit stores 1. Since P^* succeeds on too many strings, the $(i^*)^{th}$ bit of most strings is set to 0. This sparsity allows us to compress the catalytic tape. Thus, we simply iterate this process until either the catalytic tape contains a good output for the problem or until this repeated compression frees up enough space for the algorithm to perform a brute-force search over all multisets of size k. Finally, the algorithm decompresses, restoring the catalytic tape. This is our entire algorithm. Replacing our oracle queries with circuit evaluations directly gives our $\mathsf{UNPRED}^{\mathsf{TC}^1}$ algorithm. See Figure 1 for a visual representation of the compression step.

Reduction from SearchCL to UNPRED. Cook, Li, Mertz, and Pyne [CLMP25] show that SearchCL reduces to LossyCoding: given a compressor $C : \{0,1\}^n \to \{0,1\}^{n-1}$ and a decompressor $D : \{0,1\}^{n-1} \to \{0,1\}^n$, find x such that $D(C(x)) \neq x$. In their instances, C is computable in polynomial time and D is computable in polynomial time by an inplaceFL machine [CGM+25] M. Such a machine has a read-write input tape and a read-write work tape of logarithmic size, and must halt with its output written on the read-write input tape. We exploit this in-place structure. To ensure that the input and output of M have the same size, the input of M is padded with an extra bit which is simply ignored by the computation. Consider the multigraph G on $\{0,1\}^n$ which contains, for every non-solution y (i.e., y such that $D(C(y)) = y$) and every bit b, the walk traced by the input tape of the in-place machine when it is run on $C(y) \circ b$: every step of the computation adds an edge, which is labelled by the $O(\log n)$-bit internal state of the machine (and

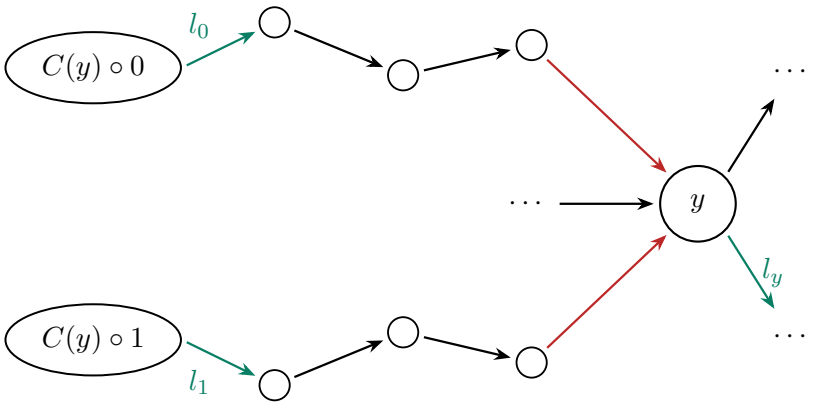

Figure 2: A non-solution $y \in \{0,1\}^n$ in the graph G. Vertex labels correspond to the string on the input tape of M and edge labels correspond to the rest of its Turing machine configuration. The first edge of each walk is coloured green, the last edge is coloured red, and intermediate edges are coloured black. Each vertex has at most one outgoing green edge. Non-solutions, such as y, have at least two incoming red edges. Each vertex has an equal number of incoming and outgoing black edges.

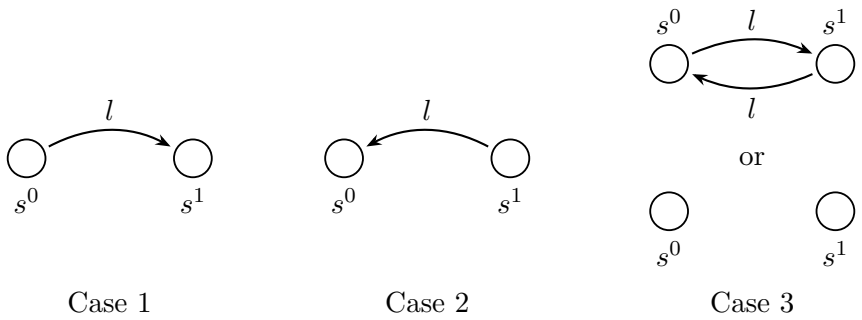

Figure 3: Reduction from WeakExcess to UNPRED. Case 1: the predictor outputs 1. Case 2: the predictor outputs 0. Case 3: the predictor outputs 0 or 1 with equal probability.

b). This walk starts at the vertex $C(y) \circ b$ and ends at the vertex $D(C(y)) = y$. Since the machine is deterministic, a non-solution y is the start vertex of at most one walk but the end vertex of two: the walks starting at $C(y) \circ 0$ and $C(y) \circ 1$. Hence, y has strictly larger in-degree than out-degree. Thus, every vertex of G with out-degree $\geq$ in-degree is a solution; we call the problem of finding such a vertex WeakExcess. See Figure 2 for a visual representation of this reduction.

We now briefly describe our reduction from WeakExcess to UNPRED. For every edge e of the graph G, the two endpoints of the edge differ in at most one bit. Moreover, the position of this bit is determined by the label l of e. For a label l, we refer to this position as $\mathsf{ind}(l) \in [n]$. For each label l, we define a middle-bit predictor P_l over $\{0,1\}^n$ at index $\mathsf{ind}(l)$ as follows. Given as input a string $s \in \{0,1\}^{n-1}$, P_l must output either 0 or 1. Let $s^0 \in \{0,1\}^n$ be the string obtained from s by inserting a 0 in the $\mathsf{ind}(l)^{th}$ position, and s^1 be the string obtained by inserting a 1. If P_l outputs 0 then it would succeed on s^0, and if it outputs 1 then it would succeed on s^1. In order to decide between the two cases, the predictor P_l looks at the edges between s^0 and s^1 with label l in the graph G. If there are more such edges directed from s^0 to s^1 than from s^1 to s^0, then P_l outputs 1. Else, if there are more such edges directed from s^1 to s^0, then P_l outputs 0. Else, if the numbers of edges in both directions are equal, P_l outputs 0 or 1, each with probability half [6]. See Figure 3 for a visual representation of these predictors. Since the labels l have $O(\log n)$ many bits, there are a total of poly(n) many predictors. A solution to UNPRED on these middle-bit predictors (with a small enough parameter ϵ) is a set of vertices such that each middle-bit predictor succeeds on very few vertices from this set. An averaging argument shows that this set must contain a single vertex on which very few of these middle-bit predictors succeed. Intuitively, these predictors are defined so that they are more likely to succeed on the head of an edge than its tail. Thus, if a vertex has higher in-degree than out-degree, many of these predictors will succeed on that vertex. The contrapositive of this statement is that a vertex on which very few of these predictors succeed should have out-degree $\geq$ in-degree. Thus, it would be a solution to WeakExcess. This is an informal description. It remains to analyze the computational complexity of these predictors, to give exact thresholds for the terms 'very few' and 'many' in our argument, and to show that a solution to WeakExcess can actually be efficiently extracted from a solution to UNPRED. We address these details in Section 4.1.

Lower Bounds for $\mathsf{UNPRED}^{\mathcal{O}_P}$ and $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$. We obtain both lower bounds by constructing predictor oracles as adversaries. Consider an UNPRED instance over $\{0,1\}^n$ which has exactly one predictor P_i at each index $i \in [n]$. Let $s \in \{0,1\}^{n-1}$ be a query to the predictor P_i. Let $s^0 \in \{0,1\}^n$ be the string obtained from s by inserting a 0 in its ith position, and similarly s^1 be the string obtained by inserting a 1. If P_i outputs 0 on the query s, then it succeeds on s^0 and fails on s^1. On the other hand, if P_i outputs 1, then it fails on s^0 and succeeds on s^1. This can be thought of as the predictor P_i assigning an orientation to the edge (s^0, s^1) in the n-dimensional hypercube. If it fails on s^0 then the edge is oriented from s^1 to s^0, and if it fails on s^1 then the edge is oriented from s^0 to s^1. The goal of an $\mathsf{UNPRED}^{\mathcal{O}_P}$ algorithm is to compute a set of strings such that each predictor succeeds on very few strings from this set. A subproblem of this is to identify a single string such that very few predictors succeed on that string. In our hypercube interpretation, this is analogous to computing a single vertex with large in-degree. Thus, the adversary wants to answer

[6] We require predictors to be deterministic. Thus, we simulate this random guessing deterministically by constructing two predictors for each label, one which always outputs 1 in this equality case, and the other which always outputs 0.

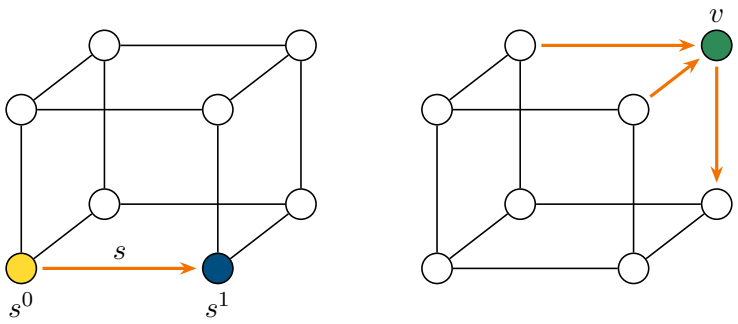

Figure 4: Hypercube interpretation of $\mathsf{UNPRED}^{\mathcal{O}_P}$. The diagram on the left is a visual representation of the case when the predictor outputs 0 on the query string s. The edge s is oriented from s^0 to s^1, indicating that the predictor fails on the string s^1. The diagram on the right is a visual representation of the vertex v that the $\mathsf{UNPRED}^{\mathcal{O}_P}$ algorithm wants to find: a vertex with high in-degree.

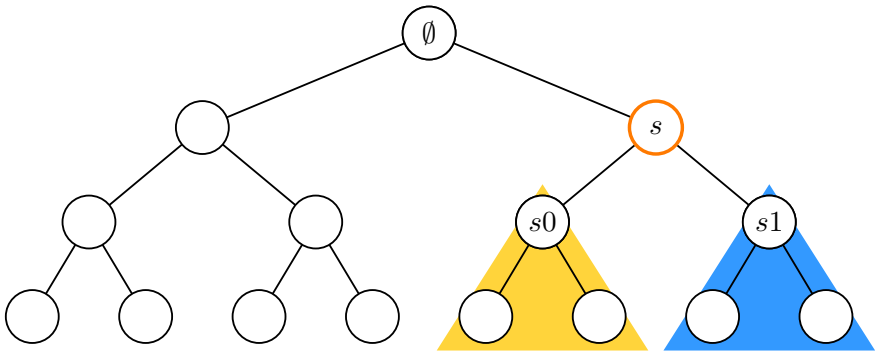

Figure 5: Binary tree interpretation of $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$. A query s to a predictor corresponds to an internal vertex of this tree. Depending on the answer of the predictor, it either fails on the entire subtree of $s0$ or the entire subtree of $s1$.

queries in a manner which maximizes the number of queries that the algorithm needs to make in order to reveal a vertex with large in-degree. We formalize the problem that the adversary needs to solve as an *online* problem: given as input a sequence of hypercube edges, assign an orientation to each edge before seeing the next one, such that the orientation minimizes the maximum in-degree of any vertex in the subgraph formed by queried edges. We give a simple greedy algorithm for this online problem which, when implemented by the adversary, results in the algorithm needing to make $\exp(\Omega(\sqrt{n}))$ queries in total to identify a single vertex with large in-degree. Our greedy algorithm is very simple: it can answer each query in $\mathrm{poly}(n)$ time (note that the query itself has $n-1$ bits). This concludes our lower bound on $\mathsf{UNPRED}^{\mathcal{O}_P}$. This idea is formalized in Section 5.1. See Figure 4 for a visual representation of this hypercube argument.

We use a similar approach for our lower bound on $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$. As $\mathsf{UNPRED}^{\mathcal{O}_P}$ naturally corresponds to the n-dimensional hypercube, $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ naturally corresponds to the complete binary tree of depth n. We label each vertex at the ith level of this tree with a string $s \in \{0,1\}^i$, and the two children of this vertex with the strings $s0$ and $s1$ (the root corresponds to the empty string $\emptyset$). A query $s \in \{0,1\}^{i-1}$ to the ith next-bit predictor P_i naturally corresponds to

the vertex in the $(i-1)^{th}$ level of this tree with label s. If P_i outputs 1 on input s, then it succeeds on every string in the subtree of $s1$ and fails on every string in the subtree of $s0$. Else, if P_i outputs 0, then it fails on every string in the subtree of $s1$ and succeeds on every string in the subtree of $s0$. This can be viewed as P_i placing a *token* on the child of vertex s that it fails on. The number of predictors that fail on a string $s \in \{0,1\}^n$ is equal to the number of tokens on the path from the root to the leaf s in the binary tree. Thus, an adversary wants to answer queries in a manner which minimizes the maximum number of tokens on any root-to-leaf path. We once again formulate the problem that the adversary wants to solve as an online problem, and then give a simple greedy algorithm for this online problem which results in the exponential lower bound that we want. This is done in Section 5.2. See Figure 5 for a visual representation of this binary tree argument.

StableCL versus SearchCL. Our main contribution here is to show that $\mathsf{UNPRED}^{\mathsf{TC}^1}$ unconditionally does not have a stable restorable catalytic logspace algorithm. Our proof is via a simple counting argument. Assume for the sake of contradiction that $\mathsf{UNPRED}^{\mathsf{TC}^1}$ is computable by a stable restorable catalytic logspace Turing machine M. Fix an input x to M describing an $\mathsf{UNPRED}^{\mathsf{TC}^1}$ instance over $\{0,1\}^n$, where $|x| \leq \text{poly}(n)$. For all initial configurations of the catalytic tape, M reaches a state where its output is explicitly written on the catalytic tape. This follows from the fact that M is restorable. Due to the fact that M is stable, this output must be the *same* for all initial catalytic tape configurations. The output consists of at least one string, and thus consumes $\geq n$ bits on the catalytic tape. Let the catalytic tape have length c. At the point when the output is explicitly written on the catalytic tape, the machine M could have at most $2^{c-n} \cdot 2^{O(\log n)}$ different configurations. This is due to the fact that n bits of the catalytic tape are fixed, so the catalytic tape could have at most 2^{c-n} different configurations, and the work tape has length $O(\log |x|) = O(\log n)$. The remaining state of M, i.e., tape head pointers, etc., can also be described using $O(\log n)$ many bits (assuming $c \leq \text{poly}(n)$, which is true because a catalytic logspace machine has a polynomial-length catalytic tape). Since M is a catalytic machine, it must eventually revert the catalytic tape back to its original configuration. There are 2^c different initial configurations of the catalytic tape. Thus, by injectivity, we must have $2^{c-n+O(\log n)} \geq 2^c$. For large enough n, this is a contradiction. Thus, $\mathsf{UNPRED}^{\mathsf{TC}^1}$ does not have a stable restorable catalytic logspace algorithm. This argument is formalized in Section 6.

Next-Bit Predictors and Pessimistic Estimators. In Section 7 we show that the problem of computing an approximate solution to a pessimistic estimator instance is equivalent to $\mathsf{NextBitUNPRED}$. The reduction from $\mathsf{NextBitUNPRED}$ to the problem of computing an approximate solution to a pessimistic estimator instance is very technical, and closely resembles an argument of Wigderson and Xiao [WX08]. We defer an explanation of this reduction to Section 7.3. The intuition for the other direction is as follows. A pessimistic estimator F over $\{0,1\}^n$ is a function which takes as input a string s of length at most n, and produces a positive real number. The function has the property that for any string s of length less than n, $(F(s0) + F(s1))/2 \leq F(s)$. Therefore, appending a uniformly random bit to s *does not increase* the value of F in expectation. The goal of this problem is to compute a string $s \in \{0,1\}^n$ such that $F(s) \leq \alpha F(\emptyset)$, where $\alpha \geq 1 + 1/\text{poly}(n)$ is an approximation parameter. Given a pessimistic estimator F, we construct a $\mathsf{NextBitUNPRED}$ instance over $\{0,1\}^n$ such that any solution contains an approximate solution to the pessimistic estimator instance. This instance consists of one next-bit predictor P_i at each index $i \in [n]$. Given as input a string $s \in \{0,1\}^{i-1}$, P_i outputs 1 with probability

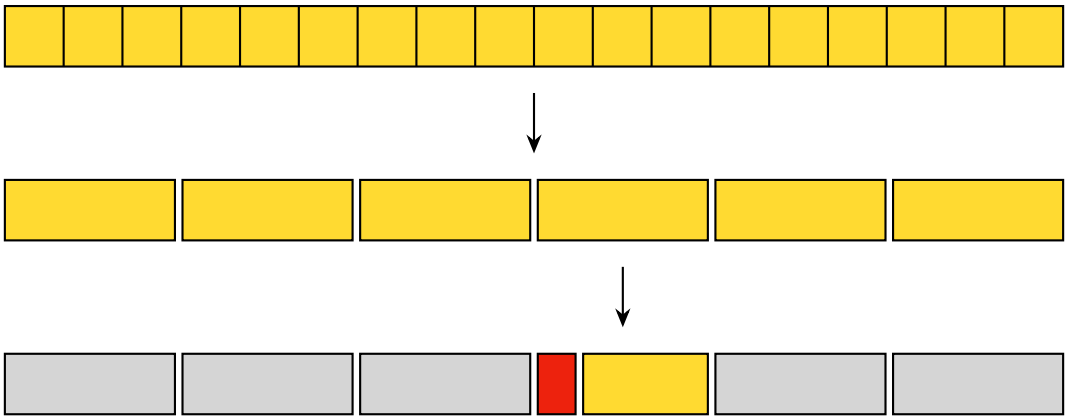

Figure 6: BlockLossyCoding. The input is split into blocks. A single block is chosen and compressed. The red region represents free space.

$F(s1)/(F(s0)+F(s1))$, and outputs 0 with probability $F(s0)/(F(s0)+F(s1))$ [7]. Thus, for a string $s \in \{0,1\}^n$, if $F(s_{\leq i}) > F(s_{<i})$, the predictor P_i is likely to succeed on s. Any string s such that $F(s) > \alpha F(\emptyset)$ must have many such indices, or have a few indices on which the inequality is very strong, i.e., on which $F(s_{\leq i})$ is much larger than $F(s_{<i})$. Thus, on any string that is not an α-approximate solution to the pessimistic estimator instance, these predictors are likely to succeed often. As a contrapositive, any string $s \in \{0,1\}^n$ on which these predictors do not succeed often is an approximate solution to the pessimistic estimator instance. This is exactly what we wanted from our reduction. This argument is formalized in Section 7.1. As a corollary of this reduction, along with the algorithm by Marcussen, Pyne, and Rubinfeld [MPR26], we obtain a reduction from graph sparsification to $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1}$.

Reductions from Bipartite Matching, Linear Matroid Intersection, and General Matching to $\mathsf{UNPRED}^{\mathsf{TC}^1}$. The compress-or-random algorithms of [AM25, AAV26, CDK+26] share a common structure: the catalytic tape is split into blocks of $\ell = O(\log n)$ bits each, and the compressor selects one block, depending on the tape, and compresses only that block (BlockLossyCoding, Definition 8.1, Figure 6). We give a simple reduction from BlockLossyCoding to UNPRED. This reduction first reduces BlockLossyCoding to a version of UNPRED over alphabets of size $\text{poly}(n)$. That is, instead of binary strings we have strings over an alphabet set A, and the predictors try to guess a character of this string with probability $\geq 1/|A| + \epsilon$. We show that this large alphabet version of UNPRED reduces to the binary version. This is done in Section 8.

1.8 Organization of the Paper

In Section 2 we give preliminaries. In Section 3 we give catalytic logspace algorithms for $\mathsf{UNPRED}^{\mathcal{O}_P}$ and $\mathsf{UNPRED}^{\mathsf{TC}^1}$ (Theorem 1.3 and Corollary 1.4). In Section 4 we reduce SearchCL to UNPRED (Theorem 1.5). In Section 5 we prove oracle lower bounds for $\mathsf{UNPRED}^{\mathcal{O}_P}$ and $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$.

[7] We require predictors to be deterministic. We approximate this random guessing by constructing T deterministic copies of each predictor, where T is a positive integer. Copy $j \in [T]$ guesses 1 exactly when $F(s1)/(F(s0)+F(s1)) \geq (2j-1)/(2T)$.

In Section 6 we discuss StableCL versus SearchCL. In Section 7 we prove the equivalence between NextBitUNPRED and pessimistic estimators, and its consequence for graph sparsification. In Section 8 we reduce bipartite matching, linear matroid intersection, and general matching to $\mathsf{UNPRED}^{\mathsf{TC}^1}$. Section 9 contains our AI disclosure.

2 Preliminaries

2.1 Notation

Numbers and Logic. $\mathbb{N}$ refers to the natural numbers $\{1, \dots\}$, $\mathbb{Z}$ refers to the integers, and $\mathbb{R}$ refers to the real numbers. We use notation such as $\mathbb{Z}_{\geq 0}$ to refer to the set of nonnegative integers. For $n \in \mathbb{Z}_{\geq 0}$, we define $[n] \coloneqq \{1, \dots, n\}$. We use log to refer to logarithms in base 2, and ln for logarithms in base e. $\exp(x)$ refers to the function e^x. For a Boolean proposition p, we use $\mathbb{1}[p]$ as an indicator function, which is 1 if p is true and 0 otherwise.

Strings, Tuples, and Functions. We use $\emptyset$ to refer to the empty string. Let s be a string of length n and $i \in [n]$ be an index. We use s_i to refer to the ith character of s. We use $s_{-i} \coloneqq s_1 \dots s_{i-1} s_{i+1} \dots s_n$ to refer to the string s without its ith character. We use $s_{\leq i} \coloneqq s_1 \dots s_i$ to refer to the length i prefix of s, and $s_{<i} \coloneqq s_1 \dots s_{i-1}$ to refer to the length $i-1$ prefix of s. We set $s_{\leq 0} \coloneqq \emptyset$ for every string s. For two strings s and t, $s \circ t$ is the concatenation of s and t. We identify tuples of strings with their concatenation. Similarly, for a tuple s of n elements, s_i refers to the ith element in the tuple. For a function $f : X \to X$ and $i \in \mathbb{Z}_{\geq 0}$, we use $f^{(i)}$ to refer to the function f composed with itself i times. That is, on an input x, $f^{(i)}(x) = f(f(\dots f(x)))$, where f is applied i times.

Graphs. All graphs in this paper are finite. For a graph G, we use $V(G)$ to refer to its vertex set, and $E(G)$ for its edge set. A simple undirected graph has edges that are unordered pairs of distinct vertices, with no repeated pairs. A simple directed graph has edges that are ordered pairs of distinct vertices, with no repeated pairs. A directed multigraph may have multiple edges with the same ordered endpoints, including self-loops. In a multigraph, in-degrees and out-degrees count edge copies, and each self-loop contributes one to both.

Multisets. All multisets in this paper are finite. For a multiset S, $|S|$ counts the number of elements in S *with* multiplicity, and $s \in S$ means that s occurs in S with non-zero multiplicity. For a set X, we write $S \subseteq X$ if every element of S belongs to X. For a non-empty multiset S, $U(S)$ is the uniform distribution over the elements of S, where an element of S is sampled with probability proportional to its multiplicity. We also write $s \in S$ in probability subscripts as shorthand for $s \sim U(S)$.

2.2 Standard Complexity Classes

Languages and Search Problems. A language L is a subset of $\{0,1\}^*$. A search problem P is defined to be a subset of $\{0,1\}^* \times \{0,1\}^*$. An algorithm solves a search problem P if, on every input x, it computes y such that $(x, y) \in P$.

Machine Classes. P is the class of languages decidable by a Turing machine in deterministic polynomial time. L is the class of languages decidable by a Turing machine with logarithmic work space. FP and FL are their search problem counterparts.

Randomized Classes. BPP and BPL are the classes of languages decidable by randomized Turing machines with error at most $1/3$ on every input, with BPP machines restricted to run in polynomial time, and BPL machines restricted to use at most logarithmic work space. ZPP is the class of languages decidable by randomized Turing machines with zero error in expected polynomial time. $\mathsf{SearchBPP}$ consists of search problems solvable by randomized polynomial-time algorithms that output a valid solution with probability at least $2/3$ on every input.

Circuits. Unless stated otherwise, circuits use AND, OR, NOT, and $\mathsf{THRESHOLD}$ gates. AND, OR, and $\mathsf{THRESHOLD}$ gates have unbounded fan-in, and NOT gates have fan-in 1. Circuits may have multiple output bits. For a circuit C, $\mathsf{size}(C)$ is the number of gates of C, including input gates. $\mathsf{depth}(C)$ is the maximum number of gates on any directed path of C. TC^1 is the class of problems solvable by logspace-uniform families of polynomial-size, $O(\log n)$-depth circuits. We write $\mathsf{L\text{-}uniform}\ \mathsf{TC}^1$ when making this uniformity explicit. NC is the class of problems solvable by logspace-uniform families of polynomial-size, $\mathrm{polylog}(n)$-depth circuits.

Measures of Computation. For a sequential algorithm $\mathcal{A}$, $\mathsf{time}(\mathcal{A})$ is the maximum number of steps that $\mathcal{A}$ takes on any given input, as a function of its input size. For a fixed input $x \in \{0,1\}^*$, $\mathsf{time}(\mathcal{A}(x))$ is the number of steps that $\mathcal{A}$ takes on input x. For a parallel algorithm $\mathcal{A}$, $\mathsf{depth}(\mathcal{A})$ denotes the maximum number of sequential steps of $\mathcal{A}$ on any given processor, and $\mathsf{work}(\mathcal{A})$ denotes the total number of operations of $\mathcal{A}$ across all processors.

Search Reductions. Let $\mathcal{C}$ be a computational class. For search problems A and B, $A \leq_{\mathcal{C}} B$ means that there is (1) an instance map f which maps instances of A to instances of B; and (2) a solution map g which takes as input an instance of A and a solution of B, and outputs a solution of A, such that (1) both f and g are computable in $\mathcal{C}$; and (2) for every $x \in \{0,1\}^*$, if $(f(x), y) \in B$; that is, y is a solution to the instance $f(x)$ for the search problem B, then $(x, g(x,y)) \in A$; that is, $g(x,y)$ is a solution to the instance x for the search problem A. This definition traces a reduction as follows: in order to solve the problem A, a machine can compute $f(x)$, solve B on $f(x)$ to obtain a solution y, and then compute $g(x,y)$ to obtain a solution for x. In particular, $\mathcal{C} = \mathsf{P}$ means that both maps are computable in deterministic polynomial time, and $\mathcal{C} = \mathsf{L\text{-}uniform}\ \mathsf{TC}^1$ means that both maps are computable by logspace-uniform TC^1 circuit families.

2.3 Predictors and UNPRED

Definition 2.1 (Next-Bit Predictor)**.** Let $n \in \mathbb{N}$. A next-bit predictor over $\{0,1\}^n$ at index $\mathsf{ind} \in [n]$ is a function $P : \{0,1\}^{\mathsf{ind}-1} \to \{0,1\}$. P succeeds on a string $s \in \{0,1\}^n$ if

$$P(s_{<\mathsf{ind}}) = s_{\mathsf{ind}}.$$

That is, given a length $\mathsf{ind} - 1$ prefix of the string, P predicts its next bit.

Definition 2.2 (Middle-Bit Predictor)**.** Let $n \in \mathbb{N}$. A middle-bit predictor over $\{0,1\}^n$ at index $\mathsf{ind} \in [n]$ is a function $P : \{0,1\}^{n-1} \to \{0,1\}$. P succeeds on a string $s \in \{0,1\}^n$ if

$$P(s_{-\mathsf{ind}}) = s_{\mathsf{ind}}.$$

That is, given the entire string except its ind^{th} bit, P predicts the remaining bit.

Definition 2.3 (Predictor Success and Advantage)**.** Let $n \in \mathbb{N}$, let P be a (next or middle)-bit predictor over $\{0,1\}^n$, and let $\mathbf{D}$ be a probability distribution over $\{0,1\}^n$. The *success* of P on $\mathbf{D}$ is

$$\mathsf{succ}_{\mathbf{D}}(P) \coloneqq \Pr_{s\sim\mathbf{D}}[P \text{ succeeds on } s].$$

The *advantage* of P on $\mathbf{D}$ is

$$\mathsf{adv}_{\mathbf{D}}(P) \coloneqq \max\{0, \mathsf{succ}_{\mathbf{D}}(P) - \frac{1}{2}\}.$$

Let $\mathcal{P}$ be a non-empty family of predictors. We overload the notation above by defining

$$\mathsf{succ}_{\mathbf{D}}(\mathcal{P}) \coloneqq \max_{P\in\mathcal{P}} \mathsf{succ}_{\mathbf{D}}(P).$$
$$\mathsf{adv}_{\mathbf{D}}(\mathcal{P}) \coloneqq \max_{P\in\mathcal{P}} \mathsf{adv}_{\mathbf{D}}(P).$$

Definition 2.4 (UNPRED)**.** UNPRED is the search problem defined as follows:

1. **Input:** $n, m \in \mathbb{N}$, $0 < \epsilon \leq 1$, Boolean circuits $P_1, \ldots, P_m$, and $\mathsf{ind}_1, \ldots, \mathsf{ind}_m \in [n]$.
   For all $i \in [m]$, $P_i : \{0,1\}^{n-1} \to \{0,1\}$ is a middle-bit predictor over $\{0,1\}^n$ at index $\mathsf{ind}_i \in [n]$.
2. **Output:** A non-empty multiset $S \subseteq \{0,1\}^n$ such that
   $$\mathsf{adv}_{U(S)}(\{P_1, \ldots, P_m\}) < \epsilon.$$

To avoid precision issues with ϵ, we assume that we are given an integer r **in unary** and $\epsilon \coloneqq 1/r$.[8] We assume that n is also given in unary.

Variants of UNPRED**.** Let $\mathcal{C}$ be a circuit class, such as TC^1. $\mathsf{UNPRED}^{\mathcal{C}}$ refers to a version of UNPRED where the predictor circuits belong to the class $\mathcal{C}$. $\mathsf{UNPRED}^{\mathcal{O}_P}$ refers to the version where the predictors can only be accessed through queries to a predictor oracle $\mathcal{O}_P$; we define this oracle model formally in Section 2.5. We also consider the subproblem of UNPRED where each predictor circuit, instead of being a middle-bit predictor, is a next-bit predictor. We refer to the unrestricted version of this as NextBitUNPRED, the version restricted to circuit class $\mathcal{C}$ as $\mathsf{NextBitUNPRED}^{\mathcal{C}}$, and the version with oracles as $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$.

Output Size. Definition 2.4 does not bound the output size of UNPRED. As is the case with hitting sets, there is typically no need to bound it beyond the resource bounds of the algorithm: for a time-bounded algorithm, the size of the output is naturally bounded by the runtime. Catalytic logspace algorithms, however, are not bounded by polynomial runtime in the worst case. Thus, we will be explicit about output size bounds in our algorithms in this case. In fact, our catalytic algorithms write their entire output on the catalytic tape itself, whose size is polynomially bounded.

[8] In all reductions to UNPRED we implicitly convert ϵ to $1/\lceil 1/\epsilon \rceil$.

Parameters. If $\epsilon > 1/2$, then the problem is trivial, since $\mathsf{adv}_{\mathbf{D}}(P) \leq 1/2$ for every distribution $\mathbf{D}$ and predictor P; thus, we will assume that $\epsilon \leq 1/2$ when convenient. We require $\epsilon > 0$ because we view UNPRED as a total derandomization problem: if $0 < \delta < 1$, then the Chernoff bound guarantees that a random multiset of size $\geq \lceil \ln(m/\delta)/(2\epsilon^2) \rceil$ (where elements of $\{0,1\}^n$ are sampled independently with replacement) constitutes a valid output with probability at least $1-\delta$. At $\epsilon = 0$, no valid output exists[9]. We assume that ϵ is encoded in unary so that the output size of this randomized algorithm, $O(\ln(m/\delta)/\epsilon^2)$, is polynomial in the input size. We assume that n is encoded in unary since, in the oracle version, the input size could otherwise be much smaller than n, and a logspace algorithm could not even index over the length of the strings that it needs to output.

Averaging Argument. The following is a simple fact that is used many times in this paper. It states that for any family of predictors $\mathcal{P}$ and any multiset S such that $\mathsf{adv}_{U(S)}(\mathcal{P})$ is small, there exists a string $s \in S$ such that not too many predictors in $\mathcal{P}$ succeed on s.

Lemma 2.5. *Let $n, m \in \mathbb{N}$ and $\epsilon > 0$. Let $\mathcal{P} \coloneqq \{P_1, \ldots, P_m\}$ be a family of (middle or next)-bit predictors over $\{0,1\}^n$. Let $S \subseteq \{0,1\}^n$ be a non-empty multiset such that*

$$\mathsf{adv}_{U(S)}(\mathcal{P}) < \epsilon.$$

Then there exists $s \in S$ such that $\Pr_{P \in \mathcal{P}}[P \text{ succeeds on } s] < 1/2 + \epsilon$.

Proof. Assume to the contrary that for all $s \in S$, $\Pr_{P \in \mathcal{P}}[P \text{ succeeds on } s] \geq 1/2 + \epsilon$. Then

$$\begin{aligned} &\Pr_{s \in S, P \in \mathcal{P}}[P \text{ succeeds on } s] \geq 1/2 + \epsilon \\ \implies &\exists P \in \mathcal{P} \text{ such that } \Pr_{s \in S}[P \text{ succeeds on } s] \geq 1/2 + \epsilon \\ &\implies \mathsf{adv}_{U(S)}(\mathcal{P}) \geq \epsilon. \end{aligned}$$

This completes the proof. □

2.4 Catalytic and In-Place Computation

Our main computational model in this paper is the catalytic space model:

Definition 2.6 (Catalytic Machines)**.** Let $s \coloneqq s(n)$ and $c \coloneqq c(n)$. A catalytic Turing machine with space s and catalytic space c is a Turing machine M with a read-only input tape of length n, a write-only output tape, a read-write work tape of length s, and a second read-write work tape of length c called the catalytic tape, which will be initialized to an adversarial string τ.

Definition 2.7 (Catalytic Computation of a Language)**.** A catalytic Turing machine M decides a language $L \subseteq \{0,1\}^*$ if for every $x \in \{0,1\}^n$ and $\tau \in \{0,1\}^c$, the result of executing M on input x with initial catalytic tape τ (i.e., $M(x,\tau)$) fulfils two properties: 1) M terminates with 1 written on the output tape if $x \in L$, and with 0 otherwise; and 2) M halts with τ written on the catalytic tape.

[9] Even the natural alternative requirement $\mathsf{succ}_{U(S)}(\mathcal{P}) < 1/2$ cannot be met if $\mathcal{P}$ contains both constant predictors at some index.

Definition 2.8 (Catalytic Decision Classes)**.** We define $\mathsf{CSPACE}[s, c]$ to be the family of languages decidable by catalytic Turing machines with space s and catalytic space c. We also define catalytic logspace as

$$\mathsf{CL} := \bigcup_{d \in \mathbb{N}} \mathsf{CSPACE}[d \log n, dn^d].$$

Past work defines the search class SearchCL with the following semantics:

Definition 2.9 (Catalytic Computation of a Search Problem)**.** A catalytic Turing machine M computes a search problem $P \subseteq \{0,1\}^* \times \{0,1\}^*$ if for every $x \in \{0,1\}^n$ and $\tau \in \{0,1\}^c$, the result of executing M on input x with initial catalytic tape τ (i.e., $M(x, \tau)$) fulfils two properties: 1) M terminates with y such that $(x, y) \in P$ written on the output tape; and 2) M halts with the catalytic tape in state τ.

Definition 2.10 (Catalytic Search Classes)**.** We define $\mathsf{SearchCSPACE}[s, c]$ to be the family of search problems computable by catalytic Turing machines with space s and catalytic space c. We also define SearchCL as

$$\mathsf{SearchCL} := \bigcup_{d \in \mathbb{N}} \mathsf{SearchCSPACE}[d \log n, dn^d].$$

In this work, we introduce a stronger definition. To see why we need it, suppose that a catalytic logspace machine M_1 decides a language L on input x whenever the string on its catalytic tape belongs to a set $A(x)$, and that a catalytic logspace machine M_2 computes the search problem $\{(x, y) \mid y \in A(x)\}$. To decide L, a catalytic machine M could try to use M_2 to compute some $\tau \in A(x)$, write τ on its catalytic tape, and then run M_1. However, Definition 2.9 does not guarantee that M can afterwards restore its catalytic tape to its original state. We therefore introduce 'restorable computation', which requires the catalytic computation to explicitly write the output to the search problem on its catalytic tape, and then restore it back to its original configuration.

Definition 2.11 (Restorable Catalytic Computation of a Search Problem)**.** A restorable catalytic Turing machine M consists of two procedures: $\mathsf{Compute}_M$ and $\mathsf{Restore}_M$, which share a read-only input tape, a read-write work tape, and a read-write catalytic tape. M computes a search problem $P \subseteq \{0,1\}^* \times \{0,1\}^*$ if, for every input x and every initial catalytic tape τ, it satisfies the following two properties: 1) $\mathsf{Compute}_M(x, \tau)$ halts with $y \in \{0,1\}^*$ such that $(x, y) \in P$ written on the first $|y|$ bits of its catalytic tape; and 2) $\mathsf{Restore}_M$ starts from the work tape and catalytic tape configuration that $\mathsf{Compute}_M(x, \tau)$ halted with, and halts with τ written on the catalytic tape. We refer to the string y as $M(x, \tau)$.

Similar to the classes SearchCSPACE and SearchCL, we define the classes RestorableSearchCSPACE and RestorableSearchCL.

Definition 2.12. We define $\mathsf{RestorableSearchCSPACE}[s, c]$ to be the family of search problems restorably computable by catalytic Turing machines with space s and catalytic space c. We also define RestorableSearchCL as

$$\mathsf{RestorableSearchCL} := \bigcup_{d \in \mathbb{N}} \mathsf{RestorableSearchCSPACE}[d \log n, dn^d].$$

Continuing the example above Definition 2.11, if the machine M_2 *restorably* computed τ, then indeed M could first use $\mathsf{Compute}_{M_2}$ to obtain $\tau \in A(x)$ on the catalytic tape, then run the machine M_1 to decide L, and finally use the $\mathsf{Restore}_{M_2}$ procedure to obtain the original catalytic tape again. Previous work on catalytic computation [CLMP25, AM25, AAV26, CDK$^+$26] used pairs of 'catalytic subroutines' which compress and decompress the catalytic tape, which can be seen as restorable catalytic computation of compression functions. However, this previous work required all the information needed for restoration to be written on the catalytic tape. Our definition allows this information to be stored on the work tape, since $\mathsf{Restore}_M$ has access to both the work tape and the catalytic tape of $\mathsf{Compute}_M$, not just the catalytic tape.

Definition 2.11 does not specify how the machine knows *which* prefix of the catalytic tape contains y. In a catalytic logspace machine $|y|$ would be polynomially bounded, so this could be done in many ways, for example by writing $|y|$ on the work tape. This detail does not affect any of our results.

In-Place Computation. Recently, Cook, Ghentiyala, Mertz, Pyne, and Sheffield [CGM$^+$25] introduced in-place computation, in which the input tape is read-write, instead of being read-only. This model captures transformations that a catalytic Turing machine might make on the catalytic tape. For example, suppose the catalytic tape has a matrix M written on it, and the algorithm needs to replace it with M^{-1}. Even though each entry of M^{-1} can be computed in catalytic logspace, the algorithm cannot simply overwrite the entries of M one at a time, since each overwritten entry may be needed to compute the remaining entries of M^{-1}, and M is much larger than the work tape. In order to study this situation, Cook et al. introduced the class $\mathsf{inplaceFL}$.

Definition 2.13 ([CGM$^+$25]; $\mathsf{inplaceFL}$)**.** An in-place Turing machine M has a read-write input tape and a read-write work tape. M is defined to compute a search problem $P \subseteq \{0,1\}^* \times \{0,1\}^*$ if, for every $x \in \{0,1\}^*$, the Turing machine M when executed on input x terminates with y such that $(x, y) \in P$ written on the first $|y|$ bits of the input tape.

Moreover, we say that $P \in \mathsf{inplaceFL}$ if there exist an in-place Turing machine M computing P and $d \in \mathbb{N}$ such that, on any input x, M uses at most $d \log |x|$ work space.

We emphasize that, in the above definition, the input tape has size exactly $|x|$, and thus the output, y, must have size at most $|x|$ as well.

2.5 Oracle Model

We will describe both algorithms and lower bounds in a query access model, where the predictors are not available to the algorithm in the form of their circuit representations. In this case, an algorithm is explicitly given as input only n, m, ϵ, and $\mathsf{ind}_1, \ldots, \mathsf{ind}_m$. For any $i \in [m]$, the algorithm may access the predictor P_i at a specific input $s \in \{0,1\}^{n-1}$ by producing the tuple (i, s). In response, the algorithm receives the bit $P_i(s)$ on a length 1 read-only tape. Every time there is a query, this tape is overwritten with the result. This tape only provides 1 bit of additional space to the algorithm, which is insignificant for the purpose of this work. We refer to this version of the problem simply as $\mathsf{UNPRED}^{\mathcal{O}_P}$, and define $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ analogously. $\mathcal{O}_P$ here refers to the predictor oracle, i.e., the function which takes as input the tuple (i, s) and produces as output $P_i(s)$.

It remains to specify where the tuple (i, s) is written. For time-bounded computation, we have an infinite-length work tape. Thus, the oracle query can simply be written on this work tape. For circuit computation, we define an extra 'oracle gate', where input wires to this gate carry the

input to the oracle, and the wires leaving this gate carry the output of the oracle. In the case of space-bounded computation, the tuple can be encoded using $\lceil \log(m+1) \rceil + n - 1$ bits, which will typically be too large to explicitly write on the work tape. In the catalytic computation model, **we require any oracle query to be explicitly written on read-write tape(s)**.

Thus, we do not allow an imaginary oracle query tape in our catalytic models. We are being deliberately vague about what it means for the query to be written down. Formally, there should be a logspace-computable injective function which takes as input an index i, and produces as output the location of the ith bit of the oracle query. However, for the sake of readability, we will not consider this extra detail. In all of our oracle algorithms, it will be clear where the query is written. Moreover, none of our oracle lower bounds are in space-bounded models, and thus this detail is never exploited.

3 Catalytic Logspace Algorithms for UNPRED

This section contains our catalytic logspace algorithms for $\mathsf{UNPRED}^{\mathcal{O}_P}$ and $\mathsf{UNPRED}^{\mathsf{TC}^1}$.

We will first give a $\mathsf{RestorableSearchCL}^{\mathcal{O}_P}$ algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$ in Theorem 3.3. This is the main algorithm for the section and contains our key ideas. Then, by replacing the oracle $\mathcal{O}_P$ with explicit evaluation of TC^1 circuits, we obtain a RestorableSearchCL algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ in Corollary 3.4.

Now, let us describe our $\mathsf{RestorableSearchCL}^{\mathcal{O}_P}$ algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$. Our algorithm uses the compress-or-random framework in a way similar to Pyne's algorithm for simulating BPL in CL [Pyn24].

In the $\mathsf{UNPRED}^{\mathcal{O}_P}$ problem, we receive as input an integer n, indicating that the predictors are over $\{0,1\}^n$, an integer m, indicating that there are m predictors, m integers $\mathsf{ind}_1, \ldots, \mathsf{ind}_m \in [n]$ where ind_i is the index that the ith predictor is trying to predict, and finally the accuracy parameter ϵ. We have query access to the predictors $\mathcal{P} = \{P_1, \ldots, P_m\}$.

We know that a uniformly random multiset of $k := \Theta(\ln(2m)/\epsilon^2)$ many strings in $\{0,1\}^n$ constitutes a valid output for $\mathsf{UNPRED}^{\mathcal{O}_P}$ with constant probability. The compress-or-random framework suggests that we should treat the catalytic tape as a random string. Thus, we first check whether the bits on the catalytic tape constitute a valid output for the problem. That is, we interpret the first kn bits on the tape as a multiset $S \subseteq \{0,1\}^n$, compute $\mathsf{adv}_{U(S)}(\mathcal{P})$, and thus check whether $\mathsf{adv}_{U(S)}(\mathcal{P}) < \epsilon$. If this is true, then we already have a valid output on the catalytic tape, so we do not need to do anything more.

What if it turns out that S is *not* a valid output? The compress-or-random framework dictates that in such a case one should try to compress the representation of S. The fact that S is not a valid output for this instance means that we have a predictor P_i such that $\mathsf{adv}_{U(S)}(P_i) \geq \epsilon$. In other words, the predictor P_i performs *too* well on S. Our approach towards compression is as follows: let $S = \{s^1, \ldots, s^k\}$ where each s^j is a string in $\{0,1\}^n$. For every $j \in [k]$, we replace the $(\mathsf{ind}_i)^{th}$ bit of s^j, i.e., $s^j_{\mathsf{ind}_i}$, with a bit s'_j which we define to be 0 if and only if P_i succeeds on the string s^j. Observe that the original bit $s^j_{\mathsf{ind}_i}$ can be recovered by first computing the result of P_i on the remaining bits of s^j, and then taking the XOR of the result with s'_j.

Clearly, what we just did is not compression. For each $j \in [k]$, we replaced a single bit $s^j_{\mathsf{ind}_i}$ with another bit s'_j. Moreover, in order to recover the original bit $s^j_{\mathsf{ind}_i}$, we now need to remember i as well. Thus, not only did we not save space, but we now need extra space to store i!

The key observation here is as follows: consider the length k string $s' := s'_1 \dots s'_k$. The fact that the predictor P_i had $\mathsf{adv}_{U(S)}(P_i) \geq \epsilon$ implies that P_i failed on at most $(1/2 - \epsilon)k$ many strings from S. Thus, the string s' has at most $(1/2 - \epsilon)k$ many 1s in it. This shows that s' has very low Hamming weight. Thus, although the string s' is not shorter than the string it replaced, it has significantly more structure.

We prove in Lemma 3.1 that such a string s' with low Hamming weight can be efficiently compressed. Moreover, it can be compressed enough to also store the index i and still be shorter than k bits. This makes intuitive sense: Hoeffding's inequality shows that *most* strings of length k have Hamming weight approximately $k/2$. Thus, if a string has Hamming weight far lower than $k/2$, one should be able to represent the string using significantly fewer than k many bits.

Therefore, whenever the catalytic tape contains a multiset $S \subseteq \{0,1\}^n$ which is *not* a solution to the $\mathsf{UNPRED}^{\mathcal{O}_P}$ instance, we can compress the catalytic tape. We do this repeatedly until either the catalytic tape contains a string which *is* a solution, or until we have freed up kn bits on the catalytic tape. In the former case, we have a solution! In the latter case, we now have enough free space on the catalytic tape to run an algorithm which brute forces over all multisets of size k until it finds a solution. In either case, we have the solution to the input instance *explicitly written on the catalytic tape* at some point. We can then decompress everything and thus restore the catalytic tape back to its initial state. Thus, we obtain a $\mathsf{RestorableSearchCL}^{\mathcal{O}_P}$ algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$.

This concludes our overview. We now prove our lemma which shows that strings with unexpectedly low Hamming weight can be compressed.

Lemma 3.1 (Low Hamming Weight Strings Can Be Compressed)**.** *There exists a pair of* $\mathsf{inplaceFL}$ *algorithms* $\mathsf{HamComp}$ *and* $\mathsf{HamDecomp}$ *as follows. Let* $n \in \mathbb{N}$*.*

1. $\mathsf{HamComp}$ *takes as input* $s \in \{0,1\}^n$ *with* $\sum_{i=1}^n s_i < n/2$*. Let* $\delta := 1 - 2/n \sum_{i=1}^n s_i$*.* $\mathsf{HamComp}$ *replaces* s *with a string* $(0^y, t)$ *where* $y := \lfloor \delta^2 n \log_2(e)/4 \rfloor$ *and* $t \in \{0,1\}^{n-y}$*.*

2. *For any string* $s \in \{0,1\}^n$ *with* $\sum_{i=1}^n s_i < n/2$*,*

$$\mathsf{HamDecomp}(\mathsf{HamComp}(s)) = s.$$

   That is, $\mathsf{HamDecomp}$ *takes as input the output of* $\mathsf{HamComp}(s)$*,* $(0^y, t)$ *above, and replaces it with the original string* s*.*

Proof. The proof relies on a re-indexing of length n binary strings which orders strings by their Hamming weight (in increasing order), and breaks ties lexicographically. Let $\mathsf{lex}_n(s)$ denote the lexicographic rank of s amongst all length n binary strings which have the same Hamming weight as s. Let $s := b \circ s'$ be a length n binary string with Hamming weight k where $s' \in \{0,1\}^{n-1}$ and $b \in \{0,1\}$. Note that

$$\mathsf{lex}_n(s) = \begin{cases} \mathsf{lex}_{n-1}(s'), & \text{if } b = 0, \\ \mathsf{lex}_{n-1}(s') + \binom{n-1}{k}, & \text{otherwise,} \end{cases} \tag{1}$$

and $\mathsf{lex}_0(\emptyset) := 0$. We define $\mathsf{reind}_n(s) := \mathsf{lex}_n(s) + \sum_{i=0}^{k-1} \binom{n}{i}$.

The algorithm for $\mathsf{HamComp}$ is as follows:

1. Initialize $k_0 = 0$ on the work tape.

2. Iterate i from 1 to n:

   (a) Let s^i be the length i suffix of s, and let k_i be the Hamming weight of s^i.

(b) Replace k_{i-1} on the work tape with k_i. To do this, we only need to add 1 to k_{i-1} if $s_{n-i+1} = 1$.

(c) Replace s^i with $\mathsf{lex}_i(s^i)$ on the input tape[10]. $\mathsf{lex}_{i-1}(s^{i-1})$ is already written on the input tape. Using Equation 1, computing $\mathsf{lex}_i(s^i)$ only involves adding $\binom{i-1}{k_i}$ to $\mathsf{lex}_{i-1}(s^{i-1})$, if $s_{n-i+1} = 1$. Computing the binomial coefficient can be done in logspace [HAM02, BCK+14], and the addition is easy to do in place. For this step, s_{n-i+1} can first be temporarily written on the work tape and replaced with 0 on the input tape.

3. Now, $\mathsf{lex}_n(s)$ is written on the input tape and $k \coloneqq k_n$ is written on the work tape. Replacing $\mathsf{lex}_n(s)$ with $\mathsf{reind}_n(s)$ just involves adding $\sum_{i=0}^{k-1} \binom{n}{i}$ to $\mathsf{lex}_n(s)$. Since each binomial coefficient can be computed in logspace and each addition can be done in place, this can be done in $\mathsf{inplaceFL}$.

By definition, $\mathsf{reind}_n(s) < \sum_{i=0}^{k} \binom{n}{i}$. For $k \leq (1-\delta)n/2$, by the Chernoff bound, this is $\leq e^{-\delta^2 n/4} \cdot 2^n$. This implies that

$$\mathsf{reind}_n(s) < 2^n \cdot 2^{-\delta^2 n \log_2(e)/4} \leq 2^{n-y}.$$

Thus, the binary encoding of $\mathsf{reind}_n(s)$ has a prefix of at least y many 0s.

We now proceed to the decompression algorithm. We define the following function:

$$\mathsf{lex}^{-1}_{n,k}(r) \coloneqq \begin{cases} 0 \circ \mathsf{lex}^{-1}_{n-1,k}(r), & \text{if } r < \binom{n-1}{k}, \\ 1 \circ \mathsf{lex}^{-1}_{n-1,k-1}(r - \binom{n-1}{k}), & \text{otherwise,} \end{cases} \tag{2}$$

and $\mathsf{lex}^{-1}_{0,0}(0) \coloneqq \emptyset$. Simply observe that for a string $s \in \{0,1\}^n$ of Hamming weight k, we have $\mathsf{lex}^{-1}_{n,k}(\mathsf{lex}_n(s)) = s$.

The algorithm for $\mathsf{HamDecomp}$ is as follows:

1. First, from $r = \mathsf{reind}_n(s)$ on the input tape, write down the Hamming weight k of s on the work tape. This is easy to compute: it is simply the minimum integer k such that $\sum_{i=0}^{k} \binom{n}{i} > r$. The sum of binomial coefficients can be computed in logspace, as can the comparison of n-bit integers.

2. Once we have k, subtract $\sum_{i=0}^{k-1} \binom{n}{i}$ from r on the input tape. Now, we simply want to compute $\mathsf{lex}^{-1}_{n,k}(r)$, where r is the string on the input tape, and k is written on the work tape.

3. Use Equation 2 to replace r with $\mathsf{lex}^{-1}_{n,k}(r)$ by iteratively replacing the prefix in rounds, similar to the algorithm for $\mathsf{HamComp}$, where we replaced the suffix.

This completes the proof. □

The following lemma uses the compression of low Hamming weight strings from Lemma 3.1 in order to compress a multiset $S \subseteq \{0,1\}^n$ which does not constitute a valid output to a $\mathsf{UNPRED}^{\mathcal{O}_P}$ instance.

Lemma 3.2 (Non-Solutions of UNPRED Can Be Compressed)**.** *Let $(n, m, \epsilon, \mathcal{P} = \{P_1, \ldots, P_m\}, \mathsf{ind}_1, \ldots, \mathsf{ind}_m)$ be a $\mathsf{UNPRED}^{\mathcal{O}_P}$ instance, and let $k \in \mathbb{N}$ such that $\lfloor \epsilon^2 k \log_2(e) \rfloor \geq \lceil \log_2 m \rceil + 1$. Let $\tau \in \{0,1\}^*$ be a string of length $\geq kn$ written on the catalytic tape, where the first kn bits of τ are interpreted as k binary strings of length n. Let S be the multiset formed by these k strings. There exists a pair of $\mathsf{CL}^{\mathcal{O}_P}$ subroutines Comp and Decomp such that, if $\mathsf{adv}_{U(S)}(\mathcal{P}) \geq \epsilon$, then:*

[10]Note that this is possible due to the fact that $\mathsf{lex}_i(s^i) < 2^i$, and thus fits inside i bits of space occupied by s^i.

1. Comp *replaces* S *on the catalytic tape with* $(0, t)$*, where* $t \in \{0,1\}^{nk-1}$.

2. Decomp *replaces* $(0, t)$ *on the catalytic tape with* S.

Proof. Interpret the first kn bits of τ as $s_1^1 s_1^2 \dots s_1^k \dots s_n^1 \dots s_n^k$. Let $S \coloneqq \{s^1, s^2, \dots, s^k\}$. That is, we assume that the strings are written in column-major order: the first k bits of τ are the first bit of each of the k strings, and so on. Of course, we can easily reorder strings between column-major and row-major order. We choose column-major ordering because it is convenient for us.

We will refer to the positions of the catalytic tape containing $s_i^1 \dots s_i^k$ as the ith *block* of the catalytic tape. The algorithm for Comp is as follows:

1. If $\mathsf{adv}_{U(S)}(\mathcal{P}) \geq \epsilon$, then there exists $i^* \in [m]$ such that $\mathsf{adv}_{U(S)}(P_{i^*}) \geq \epsilon$. Such an i^* can be computed in catalytic logspace by iterating over all $i \in [m]$ and explicitly computing $\mathsf{adv}_{U(S)}(P_i)$ for all i.

2. Let $j \coloneqq \mathsf{ind}_{i^*}$. Store i^* and j on the work tape. For all $i \in [k]$, replace s_j^i on the catalytic tape with s'_i defined as follows:

$$s'_i \coloneqq \begin{cases} 0, & \text{if } P_{i^*} \text{ succeeds on } s^i, \\ 1, & \text{otherwise.} \end{cases}$$

   Let $s' \coloneqq s'_1 \dots s'_k \in \{0,1\}^k$.

3. Due to the fact that $\mathsf{adv}_{U(S)}(P_{i^*}) \geq \epsilon$, the string s' has Hamming weight $\leq (1 - 2\epsilon) \cdot k/2$.

4. Apply $\mathsf{HamComp}$ from Lemma 3.1 on input s' to replace s' on the catalytic tape with $(0^{\lceil \log m \rceil + 1}, \bar{s})$, where $\bar{s} \in \{0,1\}^{k - \lceil \log m \rceil - 1}$.

5. Replace $(0^{\lceil \log m \rceil + 1}, \bar{s})$ on the catalytic tape with $(0, i^* - 1, \bar{s})$.

6. Swap the jth block of the catalytic tape, now containing $(0, i^* - 1, \bar{s})$, with the 1st block.

The algorithm for Decomp is as follows:

1. We have $(0, i^* - 1, \bar{s})$ written in the 1st block of the catalytic tape, and want to replace it with $s_j^1 \dots s_j^k$.

2. Let $j \coloneqq \mathsf{ind}_{i^*}$. Write down j and i^* on the work tape.

3. In order to undo Step 6 of Comp, swap the jth block on the catalytic tape with the 1st block, which contains $(0, i^* - 1, \bar{s})$.

4. Replace $(0, i^* - 1, \bar{s})$ in the jth block of the catalytic tape with $(0^{\lceil \log m \rceil + 1}, \bar{s})$. That is, just replace $i^* - 1$ with 0s.

5. Now, use $\mathsf{HamDecomp}$ from Lemma 3.1 to replace $(0^{\lceil \log m \rceil + 1}, \bar{s})$ with s'.

6. Note that $s'_i = 0$ if and only if P_{i^*} succeeds on s^i. Moreover, for each $i \in [k]$, we have s^i_{-j} written on the catalytic tape. Thus, for each $i \in [k]$, we can re-obtain $s_j^i \coloneqq s'_i \oplus P_{i^*}(s^i_{-j})$.

Note that the algorithms Comp and Decomp use the predictor oracles only in Steps 1 and 2 of Comp, and in Step 6 of Decomp. In all three cases, the input to the predictor is explicitly written down on the catalytic tape. Thus, these algorithms fit within our catalytic oracle model. This completes the proof. □

Finally, we can combine our ideas to obtain a $\mathsf{RestorableSearchCL}^{\mathcal{O}_P}$ algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$. The idea is straightforward: at each stage either the string on the catalytic tape constitutes a valid output, or, by Lemma 3.2, it can be further compressed. In the former case, we are done. In the latter case, we can go ahead and compress. Eventually, either the catalytic tape will contain a valid output, or we will obtain a sufficient amount of free space on the catalytic tape to perform a brute-force search.

Theorem 3.3.

$$\mathsf{UNPRED}^{\mathcal{O}_P} \in \mathsf{RestorableSearchCL}^{\mathcal{O}_P} \subseteq \mathsf{SearchCL}^{\mathcal{O}_P}.$$

Moreover, for an instance with parameters (n, m, ϵ), the algorithm outputs a multiset of size

$$\lceil \frac{\log(8m)}{\epsilon^2} \rceil = O(\log(2m)/\epsilon^2).$$

Proof. We will describe the $\mathsf{RestorableSearchCL}^{\mathcal{O}_P}$ algorithm that solves a $\mathsf{UNPRED}^{\mathcal{O}_P}$ instance. Let the input instance be $(n, m, \epsilon, \mathcal{P} = \{P_1, \ldots, P_m\}, \mathsf{ind}_1, \ldots, \mathsf{ind}_m)$. Let

$$k \coloneqq \lceil \frac{\log(8m)}{\epsilon^2} \rceil.$$

The multiset that we output will have size k. It's easy to see that $\lfloor \epsilon^2 k \log(e) \rfloor \geq \lceil \log_2 m \rceil + 1$, and that $k = O(\log(2m)/\epsilon^2)$. We divide the catalytic tape τ into two parts:

1. The first part has length kn. We will call this the 'output section'.
2. The second part has length kn. We will call this the 'reserve section'.

The algorithm works as follows:

1. Interpret the kn bits of the output section of τ as $S = s_1^1 s_1^2 \ldots s_1^k \ldots s_n^1 \ldots s_n^k$. Each s^i is a string in $\{0,1\}^n$, and S is thus a size k multiset of strings in $\{0,1\}^n$.
2. Maintain a counter x on the work tape, which is initialized with value 1.
3. Check whether S is a valid output for the input $\mathsf{UNPRED}^{\mathcal{O}_P}$ instance. To do this, for each predictor P_i and each $s \in S$, check whether P_i succeeds on s. Count the number of strings on which P_i succeeds, and simply check whether this number is at least $k(1/2 + \epsilon)$. All of this can be done in logspace, treating the catalytic tape as read-only.
4. If S constitutes a valid output for this instance, simply halt the compute procedure and skip to Step 6 for restoration.
5. Else, run Comp from Lemma 3.2 to compress S down to a 0 followed by $kn - 1$ bits. Swap this 0 with the xth bit in the reserve section. Increment x.

(a) If $x > kn$, then the reserve section is now a blank string of kn 0s. Use this reserve section to brute force over all possible size k multisets of length n binary strings, and check whether each of them is a valid solution for this instance (using the algorithm in step 3). We know that at least one of them will be, so as soon as we find one, swap the reserve section with the output section and halt the compute procedure. This swap is to ensure that the output, which is in the reserve section, occupies the first kn bits of the catalytic tape. As soon as the restoration procedure begins, swap the reserve section and the output section again, revert the reserve section to all 0s, and move to Step 6.

(b) Else, return to Step 3.

6. While $x > 1$, swap the $(x-1)^{th}$ bit in the reserve section with the first bit of the output section, run Decomp from Lemma 3.2 on the output section, and decrement x.

By the compression and decompression properties of Lemma 3.2, we are guaranteed to have the original catalytic tape upon termination of the restoration procedure. This algorithm uses the predictor oracle in Steps 3 and 5(a), and in the Lemma 3.2 subroutines. In each of these uses, the input to the predictor oracle is explicitly written on the catalytic tape. Thus, this algorithm fits within our catalytic oracle model, as defined in Section 2.5. This completes the proof. □

Replacing oracle queries in the algorithm of Theorem 3.3 with explicit circuit evaluations implies the following corollary.

Corollary 3.4.

$$\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{RestorableSearchCL} \subseteq \mathsf{SearchCL}.$$

Moreover, for an instance with parameters (n, m, ϵ), the algorithm outputs a multiset of size

$$\lceil \frac{\log(8m)}{\epsilon^2} \rceil = O(\log(2m)/\epsilon^2).$$

Proof. Buhrman, Cleve, Koucký, Loff, and Speelman [BCK+14] showed that

$$\mathsf{TC}^1 \subseteq \mathsf{CL}.$$

Thus, if the predictors are given as input in the form of TC^1 circuits, then one can directly use the oracle algorithm in Theorem 3.3, and substitute every oracle call to the predictors with an explicit circuit evaluation using the algorithm from [BCK+14]. This completes the proof. □

4 Catalytic Logspace Reduces to UNPRED

In this section we show that $\mathsf{UNPRED} \in \mathsf{FP}$ would imply $\mathsf{SearchCL} \subseteq \mathsf{FP}$, which would of course imply that $\mathsf{CL} \subseteq \mathsf{P}$.

It was shown by Buhrman, Cleve, Koucký, Loff, and Speelman [BCK+14] that $\mathsf{CL} \subseteq \mathsf{ZPP}$. Their proof follows from the fact that for any catalytic logspace machine M, there exists a logspace-computable polynomial $p_M : \mathbb{N} \to \mathbb{N}$ such that for any input $x \in \{0,1\}^*$,

$$\mathbb{E}_\tau[\mathsf{time}(M(x,\tau))] \leq p_M(|x|).$$

Thus, a randomized algorithm could sample a uniformly random catalytic tape τ and run $M(x,\tau)$. If M terminates within $2p_M(|x|)$ steps, then this is a polynomial-time simulation. Else, after $2p_M(|x|)$ many steps it could sample a fresh catalytic tape and start again. By Markov's inequality,

the probability over a uniformly random catalytic tape τ that $M(x, \tau)$ takes at most $2p_M(|x|)$ time is $\geq 1/2$. Therefore, excluding the time required to sample τ, the expected runtime of this algorithm is $O(p_M(|x|))$, which implies that it is a ZPP algorithm.

A natural approach towards derandomizing this algorithm is to solve the following problem deterministically in polynomial time:

Definition 4.1 (Fast Catalytic Tape Construction)**.** Let M be a catalytic logspace machine which, on an input of length n, uses at most $q(n)$ catalytic space. Let $p : \mathbb{N} \to \mathbb{N}$ be a function such that $p(n) = \text{poly}(n)$. FastCatTape(M, p) is a search problem defined as follows:

1. **Input:** $x \in \{0,1\}^*$.
2. **Output:** $\tau \in \{0,1\}^{q(|x|)}$ such that $\mathsf{time}(M(x, \tau)) \leq p(|x|)$.

We will show in this section (Theorem 4.20) that for every catalytic logspace machine M there exists a logspace-computable polynomially bounded function p_M such that FastCatTape$(M, p_M) \leq_{\mathsf{P}}$ UNPRED. Thus, UNPRED $\in$ FP would imply FastCatTape$(M, p_M) \in$ FP, which would imply that M can be simulated by a polynomial-time machine.

Our reduction from FastCatTape(M, p_M) to UNPRED goes through reductions to two problems that we have not formally introduced yet. The first is the LossyCoding problem (Definition 4.9). The second is the WeakExcess problem (Definition 4.3). These are both total derandomization problems.

In the LossyCoding problem you are given a pair of functions: one which compresses an n-bit string to $n-1$ bits, and one which decompresses an $n-1$-bit string to n bits. You are asked to produce as output an n-bit string which is *not* compressed and decompressed correctly by this pair of functions. Cook, Li, Mertz, and Pyne (Theorem 4.19) proved that FastCatTape(M, p_M) reduces to an instance of LossyCoding where both functions are computable in polynomial time. Moreover, both functions are actually computable in-place using polynomial time and logarithmic work space.

In Section 4.2, we reduce such instances of LossyCoding to the WeakExcess problem. In this problem you are given as input a highly structured yet exponentially large directed graph in the form of its compressed circuit representation. You are required to produce as output a vertex of this graph whose out-degree is at least as large as its in-degree. We defer further intuition about this problem until after we have formally defined the class of graphs and the problem, in Section 4.1.

Thus, we walk backwards through our reductions. First, we reduce WeakExcess to UNPRED in Section 4.1. Then, we reduce certain families of LossyCoding to WeakExcess in Section 4.2. Finally, we complete our reduction from FastCatTape to UNPRED in Section 4.3.

4.1 WeakExcess Reduces to UNPRED

Before we can define the WeakExcess problem, we need to define a well-structured family of graphs that we work with. We call these *deterministic hypercube digraphs*. These are directed multigraphs over the vertex set $\{0,1\}^n$. For every edge (u, v) in such a graph, u and v differ in at most one bit. Finally, each edge of the graph has a *label* in $\{0,1\}^\ell$. For each vertex v and label $l \in \{0,1\}^\ell$, there must be at most one edge leaving v which has the label l.

Although our statements and proofs are general, we focus on the parameter regime $\ell = O(\log n)$.

Definition 4.2 (Deterministic Hypercube Digraph)**.** Let $n, \ell \in \mathbb{N}$. A deterministic hypercube digraph of dimension n with ℓ-bit labels is defined to be a directed multigraph

$$G = (V, E, \mathsf{lab} : E \to \{0,1\}^\ell, \mathsf{ind} : \{0,1\}^\ell \to [n]),$$

where V is the vertex set, E a multiset of edges, lab is an edge-labelling function on E with labels in the set $\{0,1\}^\ell$, and ind is a function which maps labels in $\{0,1\}^\ell$ to indices in $[n]$. The constraints of this definition are as follows:

1. $V \coloneqq \{0,1\}^n$.
2. For each vertex $v \in V$ and label $l \in \{0,1\}^\ell$, there is at most one edge e leaving v with $\mathsf{lab}(e) = l$.
3. For all edges $e = (v, v') \in E$ we have $|\{i \mid v_i \neq v'_i\}| \leq 1$.

   Moreover, $\{i \mid v_i \neq v'_i\} \subseteq \{\mathsf{ind}(\mathsf{lab}(e))\}$. That is, for each edge label $l \in \{0,1\}^\ell$, the value $\mathsf{ind}(l)$ tells you which index of the vertex string could change when you traverse an edge labelled with l.

Now we define the WeakExcess problem. In this problem you are given a deterministic hypercube digraph G of dimension n with ℓ-bit labels and you need to compute a vertex $v \in \{0,1\}^n$ such that out-degree$_G(v) \geq$ in-degree$_G(v)$. The catch of course is that the graph has $\exp(n)$ size. Thus, you are given this graph with its representation compressed into a Boolean circuit. You may send the circuit queries of the form (v, l), where $v \in \{0,1\}^n$ and $l \in \{0,1\}^\ell$. In response, the circuit accepts if and only if there exists a non-self-loop edge in G leaving v with label l. We refer to this as an $\mathsf{EdgeExists}(v, l)$ query.

Definition 4.3 (WeakExcess)**.** WeakExcess is a total search problem defined as follows.

1. **Input:** The input consists of integers $n \in \mathbb{N}$ and $\ell \in \mathbb{N}$, given in unary. Then, a circuit $C : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}$. Finally, $\mathsf{ind} : \{0,1\}^\ell \to [n]$, explicitly given as a table.

   There is a hidden deterministic hypercube digraph of dimension n with ℓ-bit labels. Let this be $G = (V, E, \mathsf{lab}, \mathsf{ind})$. G can be accessed via the circuit C. $C(v, l) = 1$ if and only if there exists a non-self-loop edge $e \in E$ leaving v with label l. Else, $C(v, l) = 0$. Let this value be denoted by $\mathsf{EdgeExists}(v, l)$.
2. **Output:** The output consists of $v \in \{0,1\}^n$ such that

   $$\text{out-degree}_G(v) \geq \text{in-degree}_G(v).$$

WeakExcess resembles the problem Excess, introduced by Beame, Cook, Edmonds, Impagliazzo, and Pitassi [BCE+98], in which one is given access to a graph G, along with a vertex v^* where out-degree$_G(v^*) >$ in-degree$_G(v^*)$, and is asked to compute a vertex v such that in-degree$_G(v) >$ out-degree$_G(v)$. It is unclear to us whether there is any connection between these problems beyond the superficial.

WeakExcess is an auxiliary problem that we use for the purpose of simplifying our exposition. We do not claim that it is worth studying in its own right, though it may well be. Nevertheless, in order to provide intuition for this problem we first show that WeakExcess is a total derandomization problem. That is, a randomly sampled multiset $S \subseteq V$ of size $\Theta(2^\ell)$ contains a solution to WeakExcess with constant probability.

Observation 4.4. *Let $G = (V, E, \mathsf{lab}, \mathsf{ind})$ be a deterministic hypercube digraph of dimension n with ℓ-bit labels, let $k \in \mathbb{N}$ be a positive integer, and let $S \subseteq \{0,1\}^n$ be a random multiset of size k, where each string in S is drawn independently and uniformly at random from $\{0,1\}^n$ with replacement. Then*

$$\Pr[S \text{ contains a solution to } \mathsf{WeakExcess}] \geq 1 - (1 - \frac{1}{2^\ell + 1})^k \geq 1 - \frac{1}{e^{k/(2^\ell+1)}}.$$

Thus, by setting $k \geq 2(2^\ell + 1)$, $\mathsf{WeakExcess}$ becomes a derandomization problem.

Proof. This is due to the fact that for all $v \in V$, we have out-degree$_G(v) \leq 2^\ell$. Thus, for every 2^ℓ vertices that have in-degree$_G(v) >$ out-degree$_G(v)$, there must exist at least one vertex which has out-degree$_G(v) \geq$ in-degree$_G(v)$. Therefore,

$$\begin{aligned}
&\Pr_{v \sim U(\{0,1\}^n)}[\text{out-degree}_G(v) \geq \text{in-degree}_G(v)] \geq \frac{1}{2^\ell + 1} \\
\implies &\Pr_{S \subseteq \{0,1\}^n}[\nexists v \in S \text{ with out-degree}_G(v) \geq \text{in-degree}_G(v)] \leq (1 - \frac{1}{2^\ell + 1})^k \\
\implies &\Pr_{S \subseteq \{0,1\}^n}[\exists v \in S \text{ with out-degree}_G(v) \geq \text{in-degree}_G(v)] \geq 1 - (1 - \frac{1}{2^\ell + 1})^k \\
&\geq 1 - \frac{1}{e^{k/(2^\ell+1)}}.
\end{aligned}$$

This completes the proof. □

We will now start proving our reduction from $\mathsf{WeakExcess}$ to UNPRED.

First, we introduce a bit of notation that will make the rest of this section far easier to read. For a string $s \in \{0,1\}^{n-1}$, an index $i \in [n]$, and a bit $b \in \{0,1\}$, we define $s^{i,b}$ to be the string in $\{0,1\}^n$ obtained by adding the bit b in the ith position of s. That is, $s^{i,b} \coloneqq s_1 s_2 \dots s_{i-1} \circ b \circ s_i \dots s_{n-1}$.

In the following definition we will formally define the predictors that will be the result of our reduction. Before we do this, we briefly discuss the intuition underlying this definition.

Imagine the following family of middle-bit predictors: for each label $l \in \{0,1\}^\ell$, there is a predictor P_l over $\{0,1\}^n$ at index $\mathsf{ind}(l)$. Given a string $s \in \{0,1\}^{n-1}$, P_l has two options: if it outputs 0, then it is guessing that the completion of s is the string $s^{\mathsf{ind}(l),0}$, and otherwise it is guessing that the completion is $s^{\mathsf{ind}(l),1}$. Thus, P_l needs to decide between both of these strings. In order to do this, P_l looks at the set of edges between the vertices $s^{\mathsf{ind}(l),0}$ and $s^{\mathsf{ind}(l),1}$ which have the label l.

If there are strictly more edges going from $s^{\mathsf{ind}(l),0}$ to $s^{\mathsf{ind}(l),1}$, then P_l outputs 1. Else, if there are strictly more edges from $s^{\mathsf{ind}(l),1}$ to $s^{\mathsf{ind}(l),0}$, then P_l outputs 0. In the remaining case P_l simply guesses an output uniformly at random.

In this construction, intuitively, predictors tend to guess the heads of edges more often than they do the tails. Thus, for a vertex $v \in \{0,1\}^n$, if v is the head of an edge far more often than it is the tail (i.e., if v has out-degree$_G(v) <$ in-degree$_G(v)$), then these predictors are likely to perform well on it. As a corollary of this, if the predictors do not perform well on a vertex v, it implies that v must be a valid output for $\mathsf{WeakExcess}$. This intuition is formalized in Lemma 4.6.

Now we formally define these predictors. In order to make the predictors deterministic, we have to simulate random guessing by constructing *two* predictors for each label l.

Definition 4.5 (Predictors Corresponding to a $\mathsf{WeakExcess}$ Instance)**.** Let $G = (V, E, \mathsf{lab}, \mathsf{ind})$ be a deterministic hypercube digraph of dimension n with ℓ-bit labels.

We define a family of middle-bit predictors $\mathcal{P}(G) \coloneqq (P_{lb}(G))_{l\in\{0,1\}^\ell, b\in\{0,1\}}$. For $l \in \{0,1\}^\ell$ and $b \in \{0,1\}$, $P_{lb}(G)$ is a middle-bit predictor over $\{0,1\}^n$ at index $\mathsf{ind}(l)$. For $s \in \{0,1\}^{n-1}$, we define

$$P_{lb}(G)(s) \coloneqq \begin{cases} 1, & \text{if } \mathsf{EdgeExists}(s^{\mathsf{ind}(l),0}, l) \wedge \neg\mathsf{EdgeExists}(s^{\mathsf{ind}(l),1}, l), \\ 0, & \text{if } \mathsf{EdgeExists}(s^{\mathsf{ind}(l),1}, l) \wedge \neg\mathsf{EdgeExists}(s^{\mathsf{ind}(l),0}, l), \\ b, & \text{otherwise.} \end{cases}$$

Lemma 4.6. *Let $G = (V, E, \mathsf{lab}, \mathsf{ind})$ be a deterministic hypercube digraph of dimension n with ℓ-bit labels. For any string $s \in \{0,1\}^n$,*

$$\Pr_{l\in\{0,1\}^\ell, j\in\{0,1\}}[P_{lj}(G) \text{ succeeds on } s] = \frac{1}{2} + \frac{\textit{in-degree}_G(s) - \textit{out-degree}_G(s)}{2^{\ell+1}}. \tag{3}$$

Proof. This follows directly from the following facts:

1. For every label $l \in \{0,1\}^\ell$ such that there exists a non-self-loop edge e with $\mathsf{lab}(e) = l$ with vertex s as the **head**, but not a non-self-loop edge e' with $\mathsf{lab}(e') = l$ with vertex s as the **tail**, both P_{l0} and P_{l1} succeed on s.
2. For every label $l \in \{0,1\}^\ell$ such that there exists a non-self-loop edge e with $\mathsf{lab}(e) = l$ with vertex s as the **tail**, but not a non-self-loop edge e' with $\mathsf{lab}(e') = l$ with vertex s as the **head**, neither P_{l0} nor P_{l1} succeeds on s.
3. For every other label $l \in \{0,1\}^\ell$, only $P_{ls_{\mathsf{ind}(l)}}$ succeeds on s.

Let's say we have a labels of the first type, b of the second, and c of the third. Of course, $a + b + c = 2^\ell$. Each label of type a contributes 1 to $\text{in-degree}_G(s) - \text{out-degree}_G(s)$, each label of type b contributes -1, and each label of type c contributes 0. Therefore, the right-hand side of Equation 3 can be written as $1/2+(a-b)/2^{\ell+1}$. On the other hand, each label of type a contributes $1/2^\ell$ to the left-hand side of Equation 3, each label of type b contributes 0, and each label of type c contributes $1/2^{\ell+1}$. Therefore, the left-hand side can be written as $a/2^\ell + c/2^{\ell+1}$. By substituting c with $2^\ell - (a + b)$, the left-hand side becomes $a/2^\ell + 1/2 - a/2^{\ell+1} - b/2^{\ell+1} = 1/2 + (a - b)/2^{\ell+1}$, which is exactly the right-hand side. This completes the proof. □

Corollary 4.7. *Let $G = (V, E, \mathsf{lab}, \mathsf{ind})$ be a deterministic hypercube digraph of dimension n with ℓ-bit labels, and let $S \subseteq \{0,1\}^n$ be a non-empty multiset such that*

$$\mathsf{adv}_{U(S)}(\mathcal{P}(G)) < 1/2^{\ell+1}.$$

S must contain a solution to WeakExcess *on G.*

Proof. By Lemma 2.5, there exists a string $s \in S$ such that $\Pr_{l\in\{0,1\}^\ell, j\in\{0,1\}}[P_{lj}(G) \text{ succeeds on } s] < 1/2 + 1/2^{\ell+1}$. Combining this with Lemma 4.6 shows that $\text{in-degree}_G(s) - \text{out-degree}_G(s) < 1 \implies \text{in-degree}_G(s) \leq \text{out-degree}_G(s)$, since both the in-degree and out-degree are integers. This shows that S contains a solution to WeakExcess. This completes the proof. □

Theorem 4.8.

$$\mathsf{WeakExcess} \leq_{\mathsf{P}} \mathsf{UNPRED}.$$

Proof. Let $(n, \ell, C, \mathsf{ind})$ be a WeakExcess instance. Let G be the deterministic hypercube digraph corresponding to this instance. Consider the UNPRED instance described by the predictors $\mathcal{P}(G)$ and $\epsilon := 1/2^{\ell+1}$. By Corollary 4.7, any solution S to this UNPRED instance contains a solution to the WeakExcess instance.

It remains to verify the efficiency of this reduction. First, $1/\epsilon = 2^{\ell+1}$ is polynomial in the size of the WeakExcess instance, and so it can be output in unary format in polynomial time. Then, the UNPRED instance is over $\{0,1\}^n$ and contains $2^{\ell+1}$ predictors. The circuit C has $n+\ell$ input gates, which implies that $|C| \geq n$. Therefore, n can be produced as unary output, as can $2^{\ell+1}$. Next, we need to verify that the circuit computing each predictor can be computed in polynomial time. The predictors are indexed by $l \in \{0,1\}^\ell$ and $b \in \{0,1\}$. On an input s, a predictor makes two simple EdgeExists queries and then computes a function over the resulting two outputs of C, and b. Thus, each predictor is essentially 2 copies of C with the input bits hardwired to l, along with the $\mathsf{ind}(l)^{th}$ input bit in the two copies hardwired to 0 and 1 respectively. Clearly, these predictors can be computed in polynomial time. Finally, the index of a predictor corresponding to $l \in \{0,1\}^\ell$ is just $\mathsf{ind}(l)$, which is explicitly given as input and thus can be copied directly.

Now, let S be a polynomial-size solution to this UNPRED instance. For every $v \in S$, one can check whether v is a solution to the WeakExcess instance using the following equation:

$$\text{in-degree}_G(v) - \text{out-degree}_G(v) = \sum_{l \in \{0,1\}^\ell} (\mathsf{EdgeExists}(v^{\oplus \mathsf{ind}(l)}, l) - \mathsf{EdgeExists}(v, l)),$$

where $v^{\oplus \mathsf{ind}(l)}$ refers to the string v with the $\mathsf{ind}(l)^{th}$ bit flipped. Clearly, this can be computed in polynomial time for each v, since it only requires $2^{\ell+1}$ many queries to C. By computing this value for each $v \in S$, we can simply output the first v for which this value is non-positive. Such a v must exist, and is a solution to the WeakExcess instance. This completes the proof. □

This completes the reduction from WeakExcess to UNPRED. We will now describe the reduction from LossyCoding to WeakExcess.

4.2 LossyCoding Reduces to WeakExcess

The LossyCoding problem is a total derandomization problem introduced by Korten [Kor22].

Definition 4.9 (Lossy Coding)**.** Let $C : \{0,1\}^n \to \{0,1\}^{n-1}$ and $D : \{0,1\}^{n-1} \to \{0,1\}^n$. A string $x \in \{0,1\}^n$ is a solution to LossyCoding on (C, D) if $D(C(x)) \neq x$.

Similar to UNPRED and WeakExcess, it is easy to see that LossyCoding is a total derandomization problem.

Observation 4.10 ([Kor22])**.** *For any $C : \{0,1\}^n \to \{0,1\}^{n-1}$ and $D : \{0,1\}^{n-1} \to \{0,1\}^n$,*

$$\Pr_{x \in \{0,1\}^n}[x \text{ is a solution to } \mathsf{LossyCoding} \text{ on } (C, D)] \geq 1/2.$$

Therefore, LossyCoding *is a derandomization problem.*

LossyCoding has been studied in many regimes with various restrictions on the functions C and D. We will describe our own restrictions on C and D in a moment. First, for a LossyCoding instance (C, D), we define a function D^{ext} which is just D with an extra input bit that is completely ignored by the function. This only serves the purpose of turning D into a length-preserving function.

Definition 4.11 (Extension of D)**.** Let (C, D) be a LossyCoding instance with $C : \{0,1\}^n \to \{0,1\}^{n-1}$ and $D : \{0,1\}^{n-1} \to \{0,1\}^n$. We define the function $D^{\mathsf{ext}} : \{0,1\}^n \to \{0,1\}^n$ as follows. For $x \in \{0,1\}^n$,

$$D^{\mathsf{ext}}(x) \coloneqq D(x_{<n}).$$

Definition 4.12 (LossyCoding Family)**.** Let $(C_x, D_x)_{x\in\{0,1\}^*}$ be a family of LossyCoding instances. We refer to LossyCoding on $(C_x, D_x)_{x\in\{0,1\}^*}$ as the search problem where the input consists of $x \in \{0,1\}^*$ and the output must consist of a solution to LossyCoding on (C_x, D_x).

Let $(C_x, D_x)_{x\in\{0,1\}^*}$ be a family of LossyCoding instances. Our reduction from LossyCoding on this family to WeakExcess is efficient when two conditions hold: (1) C_x and D_x can be computed in polynomial time, and (2) D_x^{ext} can be computed by a polynomial-time inplaceFL machine with read-only access to advice x. Note that the measures of efficiency, i.e., 'polynomial' time and 'logarithmic' space, are in terms of the length of x. Thus, an implicit requirement is also that the instances (C_x, D_x) are over strings of polynomial size, and this size is computable in polynomial time.

For an in-place Turing machine M which has read-only access to advice x and read-write input y, we will refer to the machine's computation on (x, y) as $M_x(y)$ instead of $M(x, y)$. This is to avoid any confusion between the roles of the two strings.

Now, for a LossyCoding instance (C, D), we use $\mathsf{NonSol}(C, D)$ to refer to the set of strings which are *not* a solution to the instance.

Definition 4.13. Let $(C : \{0,1\}^n \to \{0,1\}^{n-1}, D : \{0,1\}^{n-1} \to \{0,1\}^n)$ be a LossyCoding instance. We define the set $\mathsf{NonSol}(C, D) \subseteq \{0,1\}^n$ as

$$\mathsf{NonSol}(C, D) \coloneqq \{x \in \{0,1\}^n \mid D(C(x)) = x\}.$$

We now define some additional notation for the remainder of this section. For an in-place Turing machine M, we denote the configuration of M by a tuple (y, σ), where y is the string on the read-write input tape, and σ is all remaining information about the configuration of M: the work tape string, the tape-head locations, and the state of the finite-control automaton. We refer to the second element of the tuple, σ, as the *state* of M. In cases where M has access to read-only advice, we include the tape head location in the state, but not the advice string itself. Thus, observe that the length of σ is $O(\log(\text{input length}) + \log(\text{advice length}) + \text{work space})$. We will refer to this as the *state size* of M.

For a configuration (y, σ), we use $M(y, \sigma)$ to denote the configuration reached by running M for 1 step starting at (y, σ). Following our function composition notation, $M^{(t)}(y, \sigma)$ denotes the configuration reached after running M for t steps. We denote by $\mathsf{input\text{-}head}(\sigma)$ the current location of the input tape head. We use σ_{start} to denote the initial state of M. Following standard notation, $\mathsf{time}(M(y))$ denotes the number of steps M runs on input y before halting.

Now, we informally describe our reduction to WeakExcess and the intuition behind it. We fix a lossy coding instance (C, D) and an in-place Turing machine M which computes D^{ext}. We construct a deterministic hypercube digraph $G \coloneqq G(C, D, M)$ as follows. The vertices of G correspond to strings on the input tape of M. The edges of G will carry labels which correspond to the remaining configuration of M, along with an additional bit which only serves to make the graph deterministic. Thus, for any edge of G, the tail of the edge along with its label defines a total configuration of M. We will define G to be the union of all directed computation paths of M on inputs of the form $C(y) \circ b$ where $y \in \mathsf{NonSol}(C, D)$ and $b \in \{0,1\}$. Refer to Figure 7 for a visual representation of

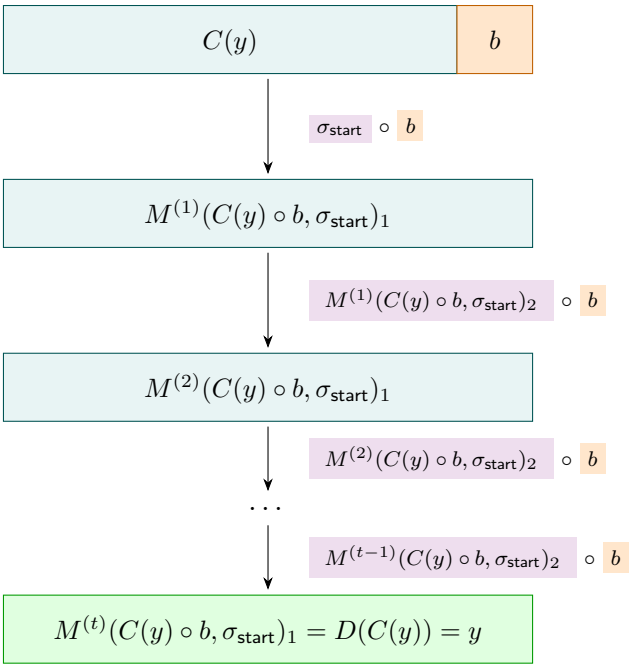

Figure 7: A computation path of M starting with input $C(y) \circ b$, represented in the graph G. The boxes correspond to vertices, the arrows to edges, and the text on the right side of each arrow is the label of the corresponding edge. Here $t := \mathsf{time}(M(C(y) \circ b))$. The label of each vertex corresponds to the string on the read-write input tape of M. The label of each edge corresponds to its state and the bit b.

such a path in G. We note that a computation path of M, when projected to G, forms a directed walk.

The intuition behind this construction is as follows: a solution to LossyCoding on (C, D) is any string $y \notin \mathsf{NonSol}(C, D)$. For all strings $y \in \mathsf{NonSol}(C, D)$, since M is deterministic, y will be the source of at most one computation path in G: the one whose sink is $D^{\mathsf{ext}}(y)$. On the other hand, y will be the sink of at least two computation paths: the ones whose sources are $C(y) \circ 0$ and $C(y) \circ 1$ respectively. Every other occurrence of y in a computation path of G will be as an intermediate vertex; such occurrences do not affect $\text{out-degree}_G(y) - \text{in-degree}_G(y)$. Thus, for any $y \in \mathsf{NonSol}(C, D)$, we will have $\text{in-degree}_G(y) > \text{out-degree}_G(y)$. The contrapositive of this statement is that for any vertex y of G such that $\text{out-degree}_G(y) \geq \text{in-degree}_G(y)$, it must be the case that $y \notin \mathsf{NonSol}(C, D)$. Thus, a solution for WeakExcess on G is a solution for LossyCoding on (C, D).

Definition 4.14. Let (C, D) where $C : \{0,1\}^n \to \{0,1\}^{n-1}$ and $D : \{0,1\}^{n-1} \to \{0,1\}^n$ be a LossyCoding instance, and M be an in-place Turing machine which computes D^{ext}. Let ℓ be the state size of M. We define a deterministic hypercube digraph of dimension n with $\ell + 1$-bit labels, $G(C, D, M) := (V, E, \mathsf{lab}, \mathsf{ind})$, as follows:

1. $V := \{0,1\}^n$.

2. $E \coloneqq \{e_{x,b,t} \mid x \in \mathsf{NonSol}(C,D), b \in \{0,1\}, 0 \le t < \mathsf{time}(M(C(x) \circ b))\}$, where
$$e_{x,b,t} \coloneqq (M^{(t)}(C(x) \circ b, \sigma_{\mathsf{start}})_1, M^{(t+1)}(C(x) \circ b, \sigma_{\mathsf{start}})_1).$$
Note that this is a multigraph, and thus we may have (x,b,t) and (x',b',t') with $e_{x,b,t} = e_{x',b',t'}$. We retain both distinct copies in the graph.

3. For every $x \in \mathsf{NonSol}(C,D)$, $b \in \{0,1\}$, and $0 \le t < \mathsf{time}(M(C(x) \circ b))$, we define
$$\mathsf{lab}(e_{x,b,t}) \coloneqq M^{(t)}(C(x) \circ b, \sigma_{\mathsf{start}})_2 \circ b.$$

4. For any $l \in \{0,1\}^{\ell+1}$ we define
$$\mathsf{ind}(l) \coloneqq \mathsf{input\text{-}head}(l_{\le \ell}).$$

Since $G(C,D,M)$ has $\ell+1$-bit labels, the input size of the corresponding WeakExcess instance is at least $2^{\ell+1}$ bits. In order for this to be at most polynomial, we need ℓ to be at most logarithmic. ℓ, being the state size of M, is at least linear in the size of its work tape. This is precisely why we need M to be an inplaceFL machine.

Observation 4.15. *Let C, D, M, n, ℓ, and $G(C,D,M)$ be as in Definition 4.14. $G(C,D,M) = (V,E,\mathsf{lab},\mathsf{ind})$ is indeed a deterministic hypercube digraph of dimension n with $\ell+1$-bit labels.*

Proof. First, for any vertex v of G and label $l \in \{0,1\}^{\ell+1}$, there is at most one edge leaving v with label l. This follows from the fact that the function D is injective when restricted to inputs in the image of C over $\mathsf{NonSol}(C,D)$. That is, for all $y \neq y' \in \mathsf{NonSol}(C,D)$, $D(C(y)) = y \neq y' = D(C(y'))$. If there were two edges $e_{x,b,t}$ and $e_{x',b',t'}$ leaving the same vertex and carrying the same label, their corresponding computation paths must end in the same output, y. This is only acceptable in two cases: (1) When these edges belong to different computation paths starting at $C(y)\circ 0$ and $C(y)\circ 1$ respectively; however, this would imply that $b \neq b'$, and thus the labels of these edges would differ; or (2) When these edges belong to the same computation path. However, this would mean that the computation of M is cyclic, which violates the assumption that M computes D^{ext}. Thus, the graph G is indeed 'deterministic' (every vertex has at most one edge leaving it with any given label).

Second, the index of the bit of a vertex v that can change when you traverse an edge leaving v with label l can be determined only from the label. This is easy to see: at any point in the computation of M, the only input tape bit that can change is the one at the location of the input tape head. This location is exactly $\mathsf{ind}(l) = \mathsf{input\text{-}head}(l_{\le \ell})$.

This completes the proof. □

Lemma 4.16. *Let (C,D) where $C : \{0,1\}^n \to \{0,1\}^{n-1}$ and $D : \{0,1\}^{n-1} \to \{0,1\}^n$ be a* LossyCoding *instance, and M be an in-place Turing machine which computes D^{ext}. Let ℓ be the state size of M. Let $G(C,D,M) = (V,E,\mathsf{lab},\mathsf{ind})$. Then any vertex $v \in \{0,1\}^n$ such that*
$$\textit{out-degree}_G(v) \ge \textit{in-degree}_G(v)$$
satisfies $D(C(v)) \neq v$. That is, any solution to WeakExcess *on G is a solution to* LossyCoding *on (C,D).*

Proof. Observe that for any $y \in \mathsf{NonSol}(C,D)$ and $b \in \{0,1\}$, the family of edges
$$(e_{y,b,t})_{t=0}^{\mathsf{time}(M(C(y)\circ b))-1}$$

forms a directed walk (possibly with self-loops) from the vertex $C(y) \circ b$ to the vertex y. Thus, for all intermediate vertices v' on this walk, the contribution of this walk to

$$\text{out-degree}_G(v') - \text{in-degree}_G(v')$$

is 0. The first vertex of the walk, $C(y) \circ b$, receives a contribution of $+1$, and the last vertex of the walk, y, receives a contribution of -1.

Therefore, for any vertex v which satisfies $\text{out-degree}_G(v) \geq \text{in-degree}_G(v)$, we have

$$|\{(y \in \mathsf{NonSol}(C, D), b \in \{0,1\}) \mid C(y) \circ b = v\}| \geq |\{(y \in \mathsf{NonSol}(C, D), b \in \{0,1\}) \mid y = v\}|.$$

Note that the left-hand side of this inequality is at most 1, since any $y \in \mathsf{NonSol}(C, D)$ and $b \in \{0,1\}$ with $C(y) \circ b = v$ would imply that $y = D(v_{<n})$ and $b = v_n$. The right-hand side of this inequality is either 0 or 2. The inequality thus implies that the right-hand side of the inequality is 0, which implies that $v \notin \mathsf{NonSol}(C, D)$. This implies that $D(C(v)) \neq v$.

This completes the proof. □

Observation 4.17. *Let (C, D) where $C : \{0,1\}^n \to \{0,1\}^{n-1}$ and $D : \{0,1\}^{n-1} \to \{0,1\}^n$ be a* LossyCoding *instance, and M be an in-place Turing machine which computes D^{ext}. Let ℓ be the state size of M. Let $v \in \{0,1\}^n$ be a vertex of $G(C, D, M)$ and $l \in \{0,1\}^{\ell+1}$ be a label. The answer to an* EdgeExists(v, l) *query on $G(C, D, M)$ is* true *if and only if:*

1. *There exists $y \in \mathsf{NonSol}(C, D)$ and $0 \leq t < \mathsf{time}(M(C(y) \circ l_{\ell+1}))$ such that*
$$M^{(t)}(C(y) \circ l_{\ell+1}, \sigma_{\mathsf{start}}) = (v, l_{\leq \ell}),$$
and
2. *$M(v, l_{\leq \ell})_1 \neq v$. This is to ensure that the edge is not a self-loop.*

This follows directly from definitions. Now, the following theorem observes that when C is polynomial-time computable, and M is a polynomial-time inplaceFL machine computing D^{ext}, these EdgeExists queries can be answered by a polynomial-size circuit.

Theorem 4.18. *Let $n : \{0,1\}^* \to \mathbb{N}$ be a function and $(C_x, D_x)_{x \in \{0,1\}^*}$ be a family of* LossyCoding *instances where $C_x : \{0,1\}^{n(x)} \to \{0,1\}^{n(x)-1}$ and $D_x : \{0,1\}^{n(x)-1} \to \{0,1\}^{n(x)}$ such that*

1. *The function n is computable in polynomial time and is polynomially bounded. That is, $n(x) \leq \text{poly}(|x|)$.*
2. *There exists a polynomial-time Turing machine M which, given as input $x \in \{0,1\}^*$ and $y \in \{0,1\}^{n(x)}$, computes $C_x(y)$.*
3. *There exists a polynomial-time* inplaceFL *Turing machine M' which, given read-only access to advice x and read-write input $y \in \{0,1\}^{n(x)}$, computes $D_x^{\mathsf{ext}}(y)$.*

Then

$$\mathsf{LossyCoding} \text{ on } (C_x, D_x)_x \leq_{\mathsf{P}} \mathsf{WeakExcess}.$$

Proof. The reduction proceeds as follows. By the first assumption, one can compute $n(x)$ in polynomial time. Now, since M' is a polynomial-time-bounded Turing machine, and $n(x)$ is a polynomially bounded function, there must exist a polynomial-time computable and polynomially bounded function T such that $T(|x|) \geq \mathsf{time}(M'_x(y))$ for all $y \in \{0,1\}^{n(x)}$.

Now, the reduction needs to take as input $x \in \{0,1\}^*$ and produce as output the WeakExcess instance corresponding to $G(C_x, D_x, M'_x)$. By Lemma 4.16, any solution to this WeakExcess instance is a solution to the LossyCoding instance (C_x, D_x).

Observe that, because M' is an inplaceFL machine and $n(x) \leq \text{poly}(|x|)$, the state size of M', say ℓ, is $O(\log(|x|))$. Thus, every part of the WeakExcess instance corresponding to $G(C_x, D_x, M'_x)$, except the circuit computing EdgeExists queries, can certainly be computed in polynomial time. In particular, the dimension of the graph $n(x)$ and the bit-length of the labels $\ell + 1$ can both be produced in unary. Furthermore, the index table $\mathsf{ind} : \{0,1\}^{\ell+1} \to [n(x)]$ can also be produced. This table has $\exp(\ell) \cdot \log(n(x))$ size. Since $\ell = O(\log(|x|))$ and $n(x) \leq \text{poly}(|x|)$, this is also $\leq \text{poly}(|x|)$. Therefore, all parts of the WeakExcess instance of $G(C_x, D_x, M'_x)$ except the circuit answering EdgeExists queries can indeed be produced in $\text{poly}(|x|)$ time.

Finally, for the circuit answering EdgeExists queries on $G(C_x, D_x, M'_x)$, we refer to Observation 4.17. This circuit must check two conditions, and then simply take the logical and of their values. We will show that both conditions can be checked in polynomial time, and thus that a polynomial-size circuit which answers EdgeExists queries can be produced in polynomial time.

The second condition is easier: it simply requires simulating M'_x for one step, and checking whether this step changed the string on the input tape. Clearly the circuit which decides this can be computed in polynomial time.

The first condition is more complicated. Fix an input $v \in \{0,1\}^{n(x)}$ and $l \in \{0,1\}^{\ell+1}$ to EdgeExists. The first condition requires checking whether there exists a string $y \in \mathsf{NonSol}(C_x, D_x)$ such that M'_x, when executed on input $C_x(y) \circ l_{\ell+1}$, reaches the configuration $(v, l_{\leq \ell})$. Of course, running a brute-force search over all strings $y \in \mathsf{NonSol}(C_x, D_x)$ would be too expensive; it would be easier to solve LossyCoding directly. Instead, we do the following. We simulate the machine M'_x from the configuration $(v, l_{\leq \ell})$ until it terminates. If it does not terminate within $T(|x|)$ steps, we can already reject, so let's assume that it terminated. Let $z \in \{0,1\}^{n(x)}$ be the string on the read-write input tape upon termination. If there indeed exists a string y which makes the condition true, it must be the case that $y = z$. Thus, we now need to check only two things: whether $z \in \mathsf{NonSol}(C_x, D_x)$, and whether the machine M'_x, when executed on the input $C_x(z) \circ l_{\ell+1}$, reaches the configuration $(v, l_{\leq \ell})$ at any point. Since M' and M are both polynomial-time machines, both of these facts can be checked in polynomial time. Thus, the first condition can also be checked in polynomial time. Since this check is uniform, it means that a circuit which decides the first condition can also be produced in polynomial time.

Combining the circuits corresponding to the two conditions provides a single circuit which answers EdgeExists queries. This completes the input to the WeakExcess problem, and thus completes the proof. □

4.3 SearchCL Reduces to UNPRED

The following theorem is proven by Cook, Li, Mertz, and Pyne [CLMP25]. That paper was written before the one by Cook, Ghentiyala, Mertz, Pyne, and Sheffield [CGM+25] which introduced inplaceFL and related classes. Thus, the theorem statement uses different language. However, our restatement can be inferred directly by unpacking and repackaging definitions. This is a reduction from FastCatTape to LossyCoding, on the type of LossyCoding instances that Theorem 4.18 efficiently reduces to WeakExcess.

Theorem 4.19 (Theorem 3.1, [CLMP25])**.** *Let M be a catalytic logspace machine which, on an input of length n, uses at most $p(n)$ catalytic space. There exists a logspace-computable polynomially bounded function $p_M : \mathbb{N} \to \mathbb{N}$ and a family of* LossyCoding *instances $(C_x, D_x)_{x\in\{0,1\}^*}$ such that:*

1. *$C_x : \{0,1\}^{n(x)} \to \{0,1\}^{n(x)-1}$ and $D_x : \{0,1\}^{n(x)-1} \to \{0,1\}^{n(x)}$, where the function n is polynomially bounded and logspace computable.*
2. *There exists a polynomial-time Turing machine M_1 which takes as input $x \in \{0,1\}^*$ and $y \in \{0,1\}^{n(x)}$, and produces as output $C_x(y)$.*
3. *There exists a polynomial-time* inplaceFL *Turing machine M_2 which, given read-only access to advice $x \in \{0,1\}^*$ and read-write input $y \in \{0,1\}^{n(x)}$, computes $D_x^{\mathsf{ext}}(y)$.*
4. *For any $x \in \{0,1\}^*$, and for any solution y to* LossyCoding *on (C_x, D_x), $y_{\leq p(|x|)}$ is a solution to* FastCatTape(M, p_M) *on input x.*

This completes all of our intermediate reductions. We can now chain them together into the following concluding theorems.

Theorem 4.20. *For every catalytic logspace machine M, there exists a logspace-computable polynomially bounded function p_M such that*

$$\mathsf{FastCatTape}(M, p_M) \leq_{\mathsf{P}} \mathsf{UNPRED}.$$

Proof. The function p_M is the one from the statement of Theorem 4.19. Theorem 4.19 reduces FastCatTape(M, p_M) to LossyCoding, where the family of LossyCoding output instances satisfies the constraints of Theorem 4.18. Theorem 4.18 reduces such LossyCoding instances to WeakExcess. Finally, Theorem 4.8 reduces WeakExcess to UNPRED.

Therefore, through this chain, we reduce FastCatTape(M, p_M) to UNPRED. □

Theorem 4.21.

$$\mathsf{UNPRED} \in \mathsf{FP} \implies \mathsf{SearchCL} \subseteq \mathsf{FP} \implies \mathsf{CL} \subseteq \mathsf{P}.$$

Proof. This follows directly as a corollary from Theorem 4.20. □

5 Lower Bounds for UNPRED$^{\mathcal{O}_P}$ and NextBitUNPRED$^{\mathcal{O}_P}$

In this section we will show two results. The first is that there does not exist a deterministic polynomial-time algorithm for UNPRED$^{\mathcal{O}_P}$. This is in contrast to NextBitUNPRED$^{\mathcal{O}_P}$, for which deterministic polynomial-time algorithms are known [GW00, LPT24]. The second result shows that the sequential algorithms of [GW00, LPT24] are, in some sense, optimal. We prove this by showing that there does not exist a deterministic parallel algorithm for NextBitUNPRED$^{\mathcal{O}_P}$ which has truly sublinear depth and polynomial work. Specifically, any truly sublinear depth algorithm must have superpolynomial work. Both of these results are unconditional.

For both problems we do this by explicitly constructing an adaptive adversary. Specifically, we construct predictor oracles $\mathcal{O}_P$ which force any (parallel, in the case of NextBitUNPRED$^{\mathcal{O}_P}$) algorithm either to make a superpolynomial number of queries or to produce an output set of superpolynomial size. Section 5.1 contains the lower bound for UNPRED$^{\mathcal{O}_P}$ and Section 5.2 contains the lower bound for NextBitUNPRED$^{\mathcal{O}_P}$.

Both sections proceed in the same way: we begin by modelling the problem that this adversary needs to solve. In both cases, this is an *online* problem. We imagine that the queries made by an $\mathsf{UNPRED}^{\mathcal{O}_P}$ (or $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$) algorithm are the input to an online algorithm, which needs to make irrevocable decisions in response to these queries. Specifically, on each query, it needs to decide the value of a predictor on the queried string. In both cases, this online problem will be a graph balancing problem. We first provide a sufficiently good online algorithm for these problems, in Sections 5.1.1 and 5.2.1, and then show how these online algorithms can be used by the adversaries to obtain the final lower bounds in Sections 5.1.2 and 5.2.2.

For both problems, what we specifically show is a superpolynomial lower bound on the total number of queries made by the algorithm plus its output size. Naturally, the running time (or total work) of any algorithm is lower bounded by the sum of these values. We note that it is not possible to remove this dependence on output size: the naive algorithm which makes no queries and produces as output the entire space $\{0,1\}^n$ is an algorithm which makes 0 queries but solves $\mathsf{UNPRED}^{\mathcal{O}_P}$. We capture such naive algorithms through the dependence on output size.

5.1 Lower Bound for $\mathsf{UNPRED}^{\mathcal{O}_P}$

In this section we will prove our lower bound on $\mathsf{UNPRED}^{\mathcal{O}_P}$. We restrict ourselves to instances over $\{0,1\}^n$ where, for each $i \in [n]$, there is exactly one predictor P_i at index i. Let $s \in \{0,1\}^{n-1}$ be a query to the predictor P_i. P_i will answer either 0 or 1. If the predictor answers 0, then it fails on the string $s^{i,1}$ (which is used to denote the string s extended to an n-bit string by placing a 1 in its ith coordinate). Else, if the predictor answers 1, then it fails on the string $s^{i,0}$. Thus, we can view the response to this query as the adversary irrevocably deciding which of the two strings the predictor P_i will fail on. By Lemma 2.5, any valid output $S \subseteq \{0,1\}^n$ for $\mathsf{UNPRED}^{\mathcal{O}_P}$ contains a string $s \in S$ such that at least $(1/2-\epsilon)n$ predictors fail on s. We will design the adversary to answer queries in a way which delays this from happening as long as possible, in order to force the algorithm to make many queries.

On each query $s \in \{0,1\}^{n-1}$ to P_i, the adversary must reveal that P_i fails on either $s^{i,0}$ or $s^{i,1}$. This can be viewed as giving an orientation to the hypercube edge between $s^{i,0}$ and $s^{i,1}$. If the predictor answers 0, then it fails on $s^{i,1}$, and we direct the edge from $s^{i,0}$ to $s^{i,1}$. Else, we direct the edge from $s^{i,1}$ to $s^{i,0}$. Rephrasing what we said in the previous paragraph, any valid output $S \subseteq \{0,1\}^n$ to $\mathsf{UNPRED}^{\mathcal{O}_P}$ must contain a string s which has in-degree at least $(1/2-\epsilon)n$ in this graph (i.e., at least $(1/2-\epsilon)n$ predictors must fail on it). Thus, the online problem for the adversary is as follows: given a stream of hypercube edges, assign an orientation to each edge. The goal is to minimize the maximum in-degree of any vertex in the subgraph formed by the queried edges. Specifically, we would like to obtain an online algorithm with the following guarantee: if any vertex in the subgraph formed by queried edges has in-degree $\geq x$, then it must be the case that at least $f(x)$ many edges have been queried. We want f to be as large as possible. This will imply that any algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$ needs to make at least $f((1/2-\epsilon)n)$ many queries in order to produce a valid output S. We will show that there exists an online algorithm for this problem with $f(x) = \exp(\Omega(\sqrt{x}))$. This will suffice for our lower bound.

Note that this problem only guarantees that the subgraph of *queried* edges does not contain a valid output. What if the algorithm outputs a multiset of strings $S \subseteq \{0,1\}^n$ that it never queried? In order to handle this, we add an additional post-processing step to the algorithm which simply sends the query string s_{-i} to P_i for every $s \in S$ and $i \in [n]$. This adds an additional $|S| \cdot n$ queries to the algorithm. This is exactly how the output size appears in the lower bound.

5.1.1 Online Hypercube Graph Balancing Problem

For $n \in \mathbb{N}$, let Q_n denote the undirected Boolean hypercube on vertex set $\{0,1\}^n$, where two vertices are adjacent if and only if they differ in exactly one coordinate.

Definition 5.1 (Online Hypercube Graph Balancing Problem)**.** The online hypercube graph balancing problem is the online problem defined as follows:

1. **Input:** $n \in \mathbb{N}$, followed by a sequence of distinct edges $(e_1, e_2, \dots)$ of Q_n **revealed one at a time**. Each e_i is a query.
2. **Output:** For every queried edge e_i, the algorithm must orient e_i before seeing e_{i+1}. That is, it must make the edge e_i directed by choosing a direction for it. This choice is irrevocable, and in particular cannot be changed after seeing e_{i+1}.

The goal of the algorithm is to minimize the maximum in-degree of any vertex. Formally, fix a non-empty input edge sequence $(e_1, \dots, e_t)$. Let G be the directed graph over vertex set $\{0,1\}^n$ and edge set $\{e_1, \dots, e_t\}$, with directions as chosen by the algorithm. Let $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be a non-decreasing function. The algorithm has in-degree bound f if

$$t \geq f(\max_{v \in V(G)} \text{in-degree}_G(v))$$

holds for every such sequence $(e_1, \dots, e_t)$.

Lemma 5.2. *There exists a deterministic algorithm for the online hypercube graph balancing problem with in-degree bound f, where $f(h) \geq \exp(\Omega(\sqrt{h}))$.*

We defer the proof of Lemma 5.2 to Section A of the appendix. The work of Hart and Nisan [HN18] can be adapted to obtain a deterministic algorithm with in-degree bound $\exp(\Omega(h))$, which is stronger than the $\exp(\Omega(\sqrt{h}))$ bound we present here. On the other hand, our algorithm is very simple and suffices for our purpose.

5.1.2 A Lower Bound for $\mathsf{UNPRED}^{\mathcal{O}_P}$ via Hypercube Balancing

Theorem 5.3. *Let $\mathcal{A}$ be a deterministic algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$. Let $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be a non-decreasing function such that there exists a deterministic algorithm for the online hypercube graph balancing problem with in-degree bound f. Then for every $n \in \mathbb{N}$ and $0 < \epsilon < 1/2$, there exists an instance of $\mathsf{UNPRED}^{\mathcal{O}_P}$, $I = (n, m = n, \epsilon, \mathsf{ind}_1, \dots, \mathsf{ind}_n)$, and a predictor oracle $\mathcal{O}_P$ such that*

$$Q + |S| \geq \frac{f((1/2 - \epsilon)n)}{n},$$

where Q denotes the number of queries $\mathcal{A}$ makes to the oracle $\mathcal{O}_P$ on input instance I, and S denotes its output multiset. Moreover, this instance satisfies $\mathsf{ind}_i = i$ for all $i \in [n]$.

Proof. We modify $\mathcal{A}$ so that, after it outputs S, it performs a post-processing round where it queries $\mathcal{O}_P(i, s_{-i})$ for every $s \in S$ and $i \in [n]$. This leaves the output S unchanged and adds only $n|S|$ new oracle queries.

Since $\mathcal{A}$ is deterministic and fixed, the oracle $\mathcal{O}_P$ can internally simulate $\mathcal{A}$ before answering each query. Thus, we assume while answering each query that $\mathcal{O}_P$ has complete access to all past queries made by $\mathcal{A}$.

For $i \in [n]$ and a string $s \in \{0,1\}^{n-1}$, we define $s^{i,0} \in \{0,1\}^n$ to be the string s with a 0 added in its ith position, and similarly $s^{i,1}$ to be the string s with a 1 added in its ith position. We view a query $(i \in [n], s \in \{0,1\}^{n-1})$ to $\mathcal{O}_P$ as an edge $\{s^{i,0}, s^{i,1}\}$ of Q_n. In response to a query (i, s), $\mathcal{O}_P$ needs to answer either 0 or 1. This can be viewed as an orientation of the edge $\{s^{i,0}, s^{i,1}\}$: if $\mathcal{O}_P$ answers 0, then the edge is oriented towards $s^{i,1}$, and if $\mathcal{O}_P$ answers 1, then the edge is oriented towards $s^{i,0}$.

Under this viewpoint, we define the oracle $\mathcal{O}_P$ to answer queries exactly as the online hypercube graph balancing algorithm which achieves in-degree bound f. Since we assumed that $\mathcal{O}_P$ has complete access to all past queries of $\mathcal{A}$, clearly $\mathcal{O}_P$ can simulate the answers of the online algorithm. If $\mathcal{A}$ repeats a query, then $\mathcal{O}_P$ returns its previous answer instead of simulating the online algorithm.

For $i \in [n]$ and $s \in \{0,1\}^{n-1}$ such that (i, s) is never queried by $\mathcal{A}$, arbitrarily define $\mathcal{O}_P(i, s) \coloneqq 0$. Let $P_1, \ldots, P_n$ be the predictors defined by this oracle $\mathcal{O}_P$. By the correctness of $\mathcal{A}$, we have

$$\mathsf{adv}_{U(S)}(\{P_1, \ldots, P_n\}) < \epsilon.$$

By Lemma 2.5, this implies that there exists $s^* \in S$ with

$$\Pr_{i \in [n]}[P_i \text{ succeeds on } s^*] < \frac{1}{2} + \epsilon. \tag{4}$$

Let $(e_1, \ldots, e_t)$ be the sequence of distinct edges of Q_n queried by $\mathcal{A}$. Consider the directed graph G over vertex set $\{0,1\}^n$ and edge set $\{e_1, \ldots, e_t\}$ (i.e., the subgraph of Q_n formed by the oriented edges) with directions induced by the answers of $\mathcal{O}_P$. Observe that for any query $(i \in [n], s \in \{0,1\}^{n-1})$ made by $\mathcal{A}$, the edge $\{s^{i,0}, s^{i,1}\}$ in G is directed towards $s^{i,0}$ if and only if P_i fails on $s^{i,0}$.

The post-processing round of $\mathcal{A}$ ensures that it made every query of the form $(i \in [n], s^*_{-i})$. Therefore, the vertex s^* in G has total degree (in-degree + out-degree) exactly n. By Equation 4, this implies that s^* has in-degree $\geq (\frac{1}{2} - \epsilon)n$ in G. Since $\mathcal{O}_P$ is an online hypercube graph balancing algorithm with in-degree bound f, this implies that the number of edges t in G must satisfy

$$t \geq f((1/2 - \epsilon)n).$$

In our case, every edge in $\{e_1, \ldots, e_t\}$ corresponds to a query made by $\mathcal{A}$ to $\mathcal{O}_P$. Due to our post-processing round, we have $t \leq Q + n|S|$. Substituting t with $Q + n|S|$ in the above equation and then dividing both sides by n proves the theorem. □

Corollary 5.4. *Let $\mathcal{A}$ be a deterministic algorithm for $\mathsf{UNPRED}^{\mathcal{O}_P}$. Then, on instances with n predictors over $\{0,1\}^n$ and parameter $\epsilon \coloneqq 1/10$, the worst-case running time of $\mathcal{A}$ satisfies*

$$\mathsf{time}(\mathcal{A}) \geq \exp(\Omega(\sqrt{n})).$$

Proof. By plugging the deterministic algorithm for the online hypercube graph balancing problem from Lemma 5.2 into Theorem 5.3, we obtain a lower bound on the query complexity plus the output size of $\mathcal{A}$. Of course, $\mathcal{A}$ must use one unit of time for each query it makes and for each string it produces as output. Thus, the stated lower bound on $\mathsf{time}(\mathcal{A})$ follows. □

5.2 Lower Bound for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$

In this section we will prove our lower bound on parallel algorithms for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$. We assume that the reader has read Section 5.1. We further build on the intuition from that section.

In particular, we will again focus on instances over $\{0,1\}^n$ where there is exactly one predictor P_i per index $i \in [n]$.

In the case of $\mathsf{UNPRED}^{\mathcal{O}_P}$, a query to a predictor P_i could only force the predictor to reveal one of two strings on which it fails. In the case of $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ on the other hand, a single query can force the predictor to reveal $\exp(n)$ many strings on which it fails. For example, consider the empty query $\emptyset$ made to P_1. If P_1 answers 0, then it fails on every string whose first bit is 1. If P_1 answers 1, then it fails on every string whose first bit is 0. Thus, it immediately reveals 2^{n-1} many strings on which it fails. The polynomial-time algorithms of [GW00, LPT24] exploit this property: they force the predictor P_1 to reveal which bit it fails on, set the first bit of the string to be equal to that, and then continue with P_2 and so on[11]. However, these algorithms are inherently sequential: the queries that they make to the predictor P_i depend on the output of the predictor P_{i-1}. We exploit this sequential nature of these algorithms in our lower bound.

Let T_n be a complete binary tree of depth n. Naturally, each vertex of T_n at depth i corresponds to a string $s \in \{0,1\}^i$. Specifically, the root corresponds to the string $\emptyset$, and for a non-leaf vertex corresponding to string s, the two children of this vertex correspond to the strings $s \circ 0$ and $s \circ 1$.

Let $s^1, \ldots, s^Q$ be the strings queried by a parallel algorithm for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ *in a single round*. That is, the algorithm sends all of these strings to the oracle $\mathcal{O}_P$ simultaneously. We view each query string as referring to its corresponding internal vertex of T_n. The adversary needs to place a *token* on exactly one child of every queried vertex. Placing a token on a vertex corresponding to a string $s \in \{0,1\}^i$ means that P_i fails on every string whose prefix is s. In other words, P_i fails on every string corresponding to a leaf in the subtree of this vertex. Similar to the case of $\mathsf{UNPRED}^{\mathcal{O}_P}$, the goal of the adversary is to minimize, after any set of queries, the maximum number of predictors that fail on any given string. This is equivalent to minimizing the maximum number of tokens on any root-to-leaf path.

Specifically, we would like to design an adversary with the following guarantee: if a set of queries $s^1, \ldots, s^Q$ results in the adversary placing x tokens on some root-to-leaf path, then it must be the case that $Q \geq f(x)$. We would like f to be as large as possible. This will imply that any algorithm for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ which runs for R rounds needs to make at least $f((1/2-\epsilon)n/R)$ many queries in order to produce a valid output S. We will show that there exists an algorithm for the adversary with $f(x) = \exp(x)$ (approximately). Thus, any parallel algorithm for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ with $R \leq n^{1-\delta}$ rounds must make at least $\exp(\Omega(n^\delta))$ many queries. This will suffice for our lower bound. In Section 5.2.1, we phrase this differently as an online problem where, in each step, the online algorithm receives an entire batch of vertices (corresponding to a set of queries made by the $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ algorithm). Thus, the requirement on the function f is slightly weaker and phrased differently. Nevertheless, the intuition we have described here is exactly the one that underlies our construction.

We also note that it is possible to describe the lower bound in Section 5.1 in terms of a similar token problem on the hypercube. We chose to instead present it as a hypercube orientation problem only to provide additional ways in which one can view the problem.

5.2.1 Online Prefix-Tree Balancing Problem

For $n \in \mathbb{N}$, let T_n denote the binary prefix tree over $\{0,1\}^n$. That is, the vertex set of T_n is $\bigcup_{i=0}^{n} \{0,1\}^i$. The empty string $\emptyset$ is the root of T_n. A vertex $s \in V(T_n)$ with $|s| < n$ has two

[11] These algorithms need to deal with multiple predictors at the same index, and so they are more technical. Nevertheless, they very much follow this simple idea.

children: the vertex $s \circ 0$ and the vertex $s \circ 1$.

Definition 5.5 (Online Prefix-Tree Balancing Problem)**.** The online prefix-tree balancing problem is the online problem defined as follows:

1. **Input:** $n \in \mathbb{N}$, followed by a sequence of pairwise disjoint batches $(B_1, B_2, \dots)$, where each batch B_i is a subset of the non-leaf vertices of T_n. These batches are **revealed one at a time**.

2. **Output:** For every queried batch B_i, the algorithm must irrevocably place a token on exactly one child of each vertex $v \in B_i$ before seeing B_{i+1}.

The goal of the algorithm is to minimize the maximum number of tokens on any root-to-leaf path. Formally, fix a batch sequence $(B_1, \dots, B_t)$. For a leaf vertex $v \in \{0,1\}^n$, let $\mathsf{load}(v)$ denote the number of tokens placed by the algorithm on the path from the root $\emptyset$ to the leaf v. Let B denote $\max_{i \in [t]} |B_i|$. For a non-decreasing function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, the algorithm has token bound f if

$$t\lfloor \log(B+1) \rfloor \geq f(\max_{v \in \{0,1\}^n} \mathsf{load}(v))$$

holds for every such sequence $(B_1, \dots, B_t)$.

Lemma 5.6. *There exists a deterministic algorithm for the online prefix-tree balancing problem with token bound $f(h) = h$.*

We defer the proof of this statement to Section B of the appendix.

5.2.2 A Parallel Lower Bound for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ via Prefix-Tree Balancing

In this section, we show a lower bound on the complexity of deterministic parallel algorithms for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$. We work with the following general structure of a parallel algorithm: the algorithm has R rounds. In each round, it makes $\leq Q$ queries to the oracle $\mathcal{O}_P$. A query made in the ith round can depend upon the answer of $\mathcal{O}_P$ to queries made in previous rounds, but not on the answer to other queries made in the ith round.

On a $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ instance over $\{0,1\}^n$ with predictors $P_1, \dots, P_m$, the algorithms of [GW00, LPT24] run in n rounds, and make polynomially many queries per round. In the following theorem, we show that these algorithms are essentially tight. That is, any algorithm which is guaranteed to run in a truly sublinear number of rounds must have superpolynomial work.

Theorem 5.7. *Let $\mathcal{A}$ be a deterministic parallel algorithm for $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$. Let $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be a non-decreasing function such that there exists a deterministic algorithm for the online prefix-tree balancing problem with token bound f. Then, for every $n \in \mathbb{N}$ and $0 < \epsilon \leq 1/2$, there exists an instance of $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$, $I = (n, m = n, \epsilon, \mathsf{ind}_1, \dots, \mathsf{ind}_m)$, and a predictor oracle $\mathcal{O}_P$ such that*

$$(R+1)\lfloor \log(\max(Q, |S|n) + 1) \rfloor \geq f((\frac{1}{2} - \epsilon)n),$$

where R denotes the number of rounds of $\mathcal{A}$ on input I with oracle $\mathcal{O}_P$, Q denotes the maximum number of queries $\mathcal{A}$ makes in any round, and S denotes its output multiset. Moreover, the instance I satisfies $\mathsf{ind}_i = i$ for all $i \in [n]$.

Proof. Similar to what we did in the proof of Theorem 5.3, we add a post-processing round to $\mathcal{A}$ where it queries $(i, s_{<i})$ for every $i \in [n]$ and $s \in S$. This adds only one round with $n|S|$ many queries. Again, since $\mathcal{A}$ is a fixed deterministic algorithm, we can assume that $\mathcal{O}_P$ has access to all past queries of $\mathcal{A}$.

A query $(i \in [n], s \in \{0,1\}^{i-1})$ to $\mathcal{O}_P$ corresponds to the non-leaf vertex s of T_n. Thus, the queries $(i_1, s_1), \ldots, (i_k, s_k)$ that $\mathcal{A}$ makes in one round correspond to a subset of non-leaf vertices of T_n. In order to answer the queries in each round, the oracle $\mathcal{O}_P$ simply treats the non-leaf vertices corresponding to the queries as a batch, and simulates a deterministic online prefix-tree balancing algorithm with token bound f. For a specific query (i_j, s_j), if the prefix-tree balancing algorithm chose to place the token of s_j at $s_j \circ 0$, then the oracle $\mathcal{O}_P$ answers this query with 1 (and vice versa). Of course, if $\mathcal{A}$ repeats a query then $\mathcal{O}_P$ responds with the same answer it gave earlier instead of using the online algorithm. For each tuple (i, s) which is never queried by $\mathcal{A}$, we arbitrarily define $\mathcal{O}_P(i, s) := 0$.

Now, let $P_1, \ldots, P_n$ be the predictors corresponding to this oracle $\mathcal{O}_P$. By the correctness of $\mathcal{A}$, we have

$$\mathsf{adv}_{U(S)}(\{P_1, \ldots, P_n\}) < \epsilon.$$

By Lemma 2.5, this implies that there exists $s^* \in S$ with

$$\Pr_{i \in [n]}[P_i \text{ succeeds on } s^*] < \frac{1}{2} + \epsilon.$$

Let (i, s) be a query made by $\mathcal{A}$, where s is a prefix of s^*. Then, by definition, the online algorithm placed a token on the vertex $s \circ s^*_i$ in T_n if and only if P_i fails on s^*. Due to the post-processing round, for every $i \in [n]$, $\mathcal{A}$ must have queried $(i, s^*_{<i})$. Therefore, upon termination of $\mathcal{A}$, the root-to-leaf path of vertex s^* in T_n must contain at least $(1/2 - \epsilon)n$ many tokens. Then, since $\mathcal{O}_P$ is an algorithm with token bound f, we have

$$(R+1)\lfloor \log(\max(Q, |S|n) + 1) \rfloor \geq f((\frac{1}{2} - \epsilon)n).$$

This completes the proof. □

Corollary 5.8. *Let $0 < \delta \leq 1$. Let $\mathcal{A}$ be a deterministic parallel algorithm for* $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ *which, on instances of* $\mathsf{NextBitUNPRED}^{\mathcal{O}_P}$ *with n predictors over $\{0,1\}^n$ and parameter $\epsilon := 1/10$, satisfies $\mathsf{depth}(\mathcal{A}) \leq O(n^{1-\delta})$. Then the worst-case work of $\mathcal{A}$ on such instances satisfies*

$$\mathsf{work}(\mathcal{A}) \geq \exp(\Omega(n^{\delta})).$$

Proof. Simply note that the work of $\mathcal{A}$ is lower bounded by both the maximum number of queries that it makes in any round, and its output size. Thus, using the notation of Theorem 5.7, $\mathsf{work}(\mathcal{A}) \geq \max(Q, |S|n)$. Plugging the algorithm from Lemma 5.6 into Theorem 5.7 gives

$$(R+1)\lfloor \log(\mathsf{work}(\mathcal{A}) + 1) \rfloor \geq (\frac{1}{2} - \epsilon)n.$$

By assumption we have

$$R + 1 \leq O(n^{1-\delta}).$$

Thus,

$$\mathsf{work}(\mathcal{A}) \geq \exp(\Omega(n^{\delta})).$$

□

6 StableCL versus SearchCL

In this section, we consider the problem of whether SearchCL reduces to CL. For example, does $\mathsf{CL} \subseteq \mathsf{L}$ imply that $\mathsf{SearchCL} \subseteq \mathsf{FL}$? To see why this is not immediately true, recall Definitions 2.7 and 2.9. The standard reduction from search to decision is as follows: let's say we have a Turing machine M which solves a search problem $P \subseteq \{0,1\}^* \times \{0,1\}^*$. Typically, one can define a Turing machine M' which accepts on input $(x \in \{0,1\}^*, i \in \mathbb{N})$ if and only if the ith bit of $M(x)$ is equal to 1. This sort of reduction is generally successful if M and M' are simple time-bounded or space-bounded Turing machines. However, this approach has a clear flaw when M and M' are more complicated machines such as randomized or catalytic. If M is a catalytic machine, then it takes as input an additional catalytic string τ, and the simulating machine M' must also take this string as input. Now, the natural generalization of the approach would be to define $M'(x, i, \tau)$ to accept if and only if the ith bit of $M(x, \tau)$ is equal to 1. However, note that Definition 2.9 allows M to have different outputs for different catalytic tapes. That is, there may exist τ and τ' such that $M(x, \tau) \neq M(x, \tau')$. At the same time, Definition 2.7 requires that a catalytic machine M' solving a decision problem must have the same output for all catalytic tapes. Thus, $M'(x, i, \tau)$ must be equal to $M'(x, i, \tau')$ for all τ, τ'. It is thus clear that this reduction cannot work naively.

This brings us to the following problem.

Problem 6.1. Is $\mathsf{FL}^{\mathsf{CL}} = \mathsf{SearchCL}$?

For readers familiar with complexity theory, this is similar to the problem of SearchBPP vs $\mathsf{FP}^{\mathsf{BPP}}$. In that case, it is known that $\mathsf{FP}^{\mathsf{BPP}}$ is equivalent to PseudodeterministicBPP [GG11]. This is a subclass of SearchBPP where the randomized Turing machine is required to output the *same* solution for any given input with high probability. Thus, it eliminates the additional instability in search problems caused by the randomness.

This approach naturally extends to our case. We define a subclass of SearchCL known as StableCL, where the catalytic Turing machine is required, for a fixed input x, to produce the same output y for all initial catalytic tape configurations. Thus, it eliminates the additional instability in search problems caused by the initial catalytic tape.

Definition 6.2 (StableCL)**.** A catalytic Turing machine M which uses $c(n)$ catalytic space and computes a search problem $P \subseteq \{0,1\}^* \times \{0,1\}^*$ is stable if for every $x \in \{0,1\}^n$ and $\tau, \tau' \in \{0,1\}^{c(n)}$, $M(x, \tau) = M(x, \tau')$. StableCL is the class of search problems computable by stable catalytic Turing machines with $O(\log n)$ work space and $\mathrm{poly}(n)$ catalytic space.

In this section, we will show that the question of whether StableCL = SearchCL is equivalent to the question of whether SearchCL is equal to $\mathsf{FL}^{\mathsf{CL}}$. Moreover, since $\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{SearchCL}$ (due to Corollary 3.4), the question of whether $\mathsf{UNPRED}^{\mathsf{TC}^1} \in \mathsf{StableCL}$ is a special case of this problem. We show that any algorithm which follows the same idea as the one in Corollary 3.4 cannot work. In particular, Corollary 3.4 additionally shows that $\mathsf{UNPRED}^{\mathsf{TC}^1}$ is in RestorableSearchCL. If we additionally define a stable version of RestorableSearchCL, then we can show that it is unconditionally impossible for any search problem with output size $\omega(\log n)$ to be computed in stable RestorableSearchCL.

We begin by showing that $\mathsf{StableCL} = \mathsf{FL}^{\mathsf{CL}}$. Since an FL machine simply does not possess linear read-write space, we use the classical oracle tape model here.

Lemma 6.3. $\mathsf{StableCL} \subseteq \mathsf{FL}^{\mathsf{CL}}$.

Proof. Let P be a search problem in $\mathsf{StableCL}$ and M be the stable catalytic logspace Turing machine which computes it. For any $x \in \{0,1\}^*$, we define $M(x) \coloneqq M(x, 0 \dots 0)$. Since M is stable, note that $M(x) = M(x, \tau)$ for any initial catalytic tape configuration τ. We define the languages P_{len} and P_{ind} as follows:

1. $(x \in \{0,1\}^*, i \in \mathbb{N}) \in P_{\mathsf{len}}$ if and only if $|M(x)| \geq i$.
2. $(x \in \{0,1\}^*, i \in \mathbb{N}) \in P_{\mathsf{ind}}$ if and only if $(x, i) \in P_{\mathsf{len}}$ and the ith bit of $M(x)$ is equal to 1.

It remains to be proven that P_{len} and P_{ind} can be decided by catalytic logspace machines. First, as discussed in Section 4, a catalytic logspace machine runs in polynomial time on average. Thus, the length of $M(x)$ must be at most polynomial. Therefore, in order to decide P_{len}, a catalytic logspace machine can simply simulate M, keep track of its output length on the work tape, and accept if and only if the output length is at least i. Similarly, to decide P_{ind}, a catalytic logspace machine can simulate M, remember the ith bit of its output, and then simply accept if and only if this bit is 1. If M terminates before reaching the ith bit then the machine should, of course, reject. Thus, P_{len} and P_{ind} are in CL.

Now, simply note that an FL machine with oracle access to P_{ind} and P_{len} can easily compute P as follows: on input x, it iterates a counter i from 1 in ascending order. It first uses P_{len} to check whether i is at most the length of $M(x)$. If this is false, then it terminates. Else, it uses P_{ind} to compute the ith bit of $M(x)$, and writes this on the output tape. It then increments i. Clearly this machine solves the search problem P.

This completes the proof. □

Lemma 6.4. $\mathsf{FL}^{\mathsf{CL}} \subseteq \mathsf{StableCL}$.

Proof. Simply note that any $\mathsf{FL}^{\mathsf{CL}}$ machine can be simulated by a single catalytic logspace machine, which uses its own catalytic tape as the catalytic tape for all CL machines used by the $\mathsf{FL}^{\mathsf{CL}}$ machine as oracles[12]. All CL machines used as oracles are stable, and the FL machine itself does not use a catalytic tape at all; thus, this simulating machine is also stable. This completes the proof. □

Corollary 6.5. $\mathsf{FL}^{\mathsf{CL}} = \mathsf{StableCL}$.

Thus, the following problem is equivalent to Problem 6.1.

Problem 6.6. Is $\mathsf{StableCL} = \mathsf{SearchCL}$?

We note that Agarwala, Alekseev, and Vinciguerra [AAV26] briefly discuss the need for $\mathsf{StableCL}$ independently of Problem 6.1. Their $\mathsf{SearchCL}$ algorithm for linear matroid intersection uses the $\mathsf{SearchCL}$ algorithm for bipartite matching by Agarwala and Mertz [AM25] as a subroutine. In fact, they use this subroutine multiple times across their algorithm, even as their algorithm mutates the string on the catalytic tape. Thus, they have to deal with the fact that the bipartite matching algorithm may return different matchings across the course of their algorithm, even when queried on the same graph. This is evidence of the fact that the notion of stability (and pseudodeterminism) is particularly important for subroutines in the low-space setting, where an algorithm cannot afford to store the entire output of a search subroutine on its work tape, and thus relies on stability of the subroutine across multiple calls.

[12] Replacing an oracle with another space-bounded machine is a standard technique in space-bounded algorithms.

Now, essentially all of the breakthrough SearchCL algorithms of recent years (bipartite matching, linear matroid intersection, general matching, graph sparsification) are *not* stable. Furthermore, it is extremely unclear how they could be turned into stable algorithms. There are tricks, such as those in [AV26], that can be exploited to make certain SearchCL algorithms stable, such as those for s-t minimum cut and global minimum cut (which arise as corollaries of the bipartite matching algorithm). However, these are highly problem-specific and it is unclear whether any general strategy exists. We note that in the related SearchBPP vs $\mathsf{FP}^{\mathsf{BPP}}$ case, it is known under standard circuit-hardness assumptions [IW97] that efficiently verifiable randomized search problems admit pseudodeterministic polynomial-time algorithms. We can draw no such conclusion here. Thus, it is not only unclear how a proof of StableCL = SearchCL would go, but it is also very unclear whether this fact is true at all.

In this section, we show a major barrier towards proving StableCL = SearchCL. Note that the algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ is not only a SearchCL algorithm, but also a RestorableSearchCL algorithm. This means that the algorithm can explicitly write down the correct output on a prefix of the catalytic tape itself. Now, imagine a variant of RestorableSearchCL which additionally requires the algorithm to be stable. That is, we require the correct output to be written on a prefix of the catalytic tape and to be the same across all instantiations of the catalytic tape. We show that such an algorithm is unconditionally impossible for any search problem whose output has length $\omega(\log n)$. Thus, any restorable catalytic logspace algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ *cannot* be stable. This implies that any StableCL algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ must be substantially different from the one in Corollary 3.4. This also shows why the SearchCL algorithms of recent years seem difficult to make stable - because, as we show in this work, they all can be reinterpreted as reductions to $\mathsf{UNPRED}^{\mathsf{TC}^1}$.

We begin by defining StableRestorableSearchCL.

Definition 6.7 (StableRestorableSearchCL)**.** A restorable catalytic Turing machine M is defined to be stable if for all $x \in \{0,1\}^*$ and all initial catalytic tape configurations τ, τ', we have $M(x,\tau) = M(x,\tau')$. A search problem P is in StableRestorableSearchCL if it is computed by a stable restorable catalytic logspace machine.

The following lemma shows that, up to additive logarithmic terms, any stable restorable catalytic machine which computes a search problem P must use at least as much work space as the size of the output.

Lemma 6.8. *Let $P \subseteq \{0,1\}^* \times \{0,1\}^*$ be a search problem. Fix $x \in \{0,1\}^*$. Let M be a stable restorable catalytic Turing machine which has a finite-control automaton consisting of z states and, on an input $x \in \{0,1\}^n$, uses s work space, c catalytic space, and produces as output y such that $(x,y) \in P$. Then*

$$s + \lceil \log n \rceil + \lceil \log s \rceil + \lceil \log c \rceil + \lceil \log z \rceil \geq |y|.$$

Proof. Note that the entire configuration of M on an input x with any initial catalytic tape string at any point can be described using $s + c + \lceil \log n \rceil + \lceil \log s \rceil + \lceil \log c \rceil + \lceil \log z \rceil$ many bits. The bits are required to store the locations of the tape heads, the strings on the work tape and catalytic tape, and the current state in the finite control of M.

Consider the map $\Pi_M^x : \{0,1\}^c \to \{0,1\}^{s+c+\lceil \log n \rceil + \lceil \log s \rceil + \lceil \log c \rceil + \lceil \log z \rceil}$ which maps an initial catalytic tape $\tau \in \{0,1\}^c$ to the complete configuration of M when the $\mathsf{Compute}_M$ procedure terminates. Observe that, from $\Pi_M^x(\tau)$, the procedure $\mathsf{Restore}_M$ recovers τ. Thus, Π_M^x must be

injective. Now, if M is stable, then the first $|y|$ bits of the catalytic tape when $\mathsf{Compute}_M$ terminates are the same for all initial catalytic tapes - they are the string y. Thus, the image of Π_M^x has size at most $2^{s+c-|y|+\lceil\log n\rceil+\lceil\log s\rceil+\lceil\log c\rceil+\lceil\log z\rceil}$. On the other hand, the domain has size 2^c. Injectivity implies the size of the image is at least the size of the domain. Thus,

$$s + \lceil\log n\rceil + \lceil\log s\rceil + \lceil\log c\rceil + \lceil\log z\rceil \geq |y|.$$

This completes the proof. □

Corollary 6.9. *A restorable catalytic logspace algorithm for* $\mathsf{UNPRED}^{\mathsf{TC}^1}$ *cannot be stable.*

Proof. This follows directly from Lemma 6.8. It is very easy to see that there are families of $\mathsf{UNPRED}^{\mathsf{TC}^1}$ instances where any valid output has size $\Omega(n)$. Thus, Lemma 6.8 would imply that any stable restorable catalytic algorithm for $\mathsf{UNPRED}^{\mathsf{TC}^1}$ using $\mathrm{poly}(n)$ catalytic space would require $\Omega(n)$ work space. Such an algorithm would not be a catalytic logspace algorithm. □

7 Next-Bit Predictors, Pessimistic Estimators, and Graph Sparsification

In this section we will show that, up to small changes in parameters, the problem of fooling next-bit predictors ($\mathsf{NextBitUNPRED}$) is equivalent to the problem of computing an approximate solution to a *pessimistic estimator* instance (Definition 7.1). In particular, this will imply that the graph sparsification algorithm by Marcussen, Pyne, and Rubinfeld [MPR26] can be restated as a TC^1 reduction to $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1}$.

The underlying method of pessimistic estimators was introduced by Raghavan [Rag88]. We begin by defining pessimistic estimator instances, and what an approximate solution to such an instance means.

Definition 7.1 (Pessimistic Estimator)**.** Let $n \in \mathbb{N}$, and define $S_n \coloneqq \bigcup_{i=0}^{n}\{0,1\}^i$. A pessimistic estimator instance over $\{0,1\}^n$ consists of a function $F : S_n \to \mathbb{R}_{>0}$ with the property that for any string $s \in S_{n-1}$,

$$\frac{F(s\circ 0) + F(s\circ 1)}{2} \leq F(s). \tag{5}$$

For $\alpha \in \mathbb{R}_{>0}$, a string $s \in \{0,1\}^n$ is an α-approximate solution to the pessimistic estimator instance F if

$$F(s) \leq \alpha \cdot F(\emptyset).$$

An approach that has been fairly successful in the field of derandomization is this: in order to obtain a string $s \in S \subseteq \{0,1\}^n$, define a pessimistic estimator F such that any (possibly even approximate) solution to F belongs to S.

Observe that, given oracle access to the function F, computing an exact solution (i.e., a 1-approximation) to a pessimistic estimator instance is easy in polynomial time: begin with the string $s = \emptyset$. By Equation 5, you are guaranteed that there exists $b \in \{0,1\}$ such that $F(s \circ b) \leq F(s)$. Set $s \leftarrow s \circ b$, and iterate this process until s has length n. This process is highly sequential. In fact, it is possible to show (though we will not do it here) that the problem of computing an exact solution to a pessimistic estimator instance F is P-complete under logspace reductions, even when F is represented explicitly by an NC^1 circuit given as input.

Thus, while computing an exact solution to a pessimistic estimator instance is easy in polynomial time, the fact that it is P-complete means we have to settle for approximations in weaker computational models. Observe that the expected value of $F(s)$ over a uniformly random string $s \in \{0,1\}^n$ is at most $F(\emptyset)$. Thus, for any $\alpha > 1$, a uniformly random string $s \in \{0,1\}^n$ is an α-approximate solution to a pessimistic estimator instance F with probability $\geq 1 - 1/\alpha$. This follows from the Markov bound. This implies that, for $\alpha \geq 1 + \Omega(1/\text{poly}(n))$, a set of $\text{poly}(n)$ uniformly random strings from $\{0,1\}^n$ contains an α-approximate solution to F with high probability. Such a solution can be verified using just two queries to F. Thus, the problem of producing an α-approximate solution to a pessimistic estimator instance is a total derandomization problem.

Marcussen, Pyne, and Rubinfeld [MPR26] use pessimistic estimators to solve the graph spectral sparsification problem (GraphSparsification, formally defined in Definitions 7.2 and 7.3). Specifically, they showed that GraphSparsification has a L-uniform TC^1 reduction to the problem of computing an n-approximation to a pessimistic estimator instance F, where F is explicitly given as a TC^1 circuit. They concluded that GraphSparsification belongs to SearchCL by giving a compress-or-random algorithm which computes an approximate solution to the pessimistic estimator instance.

In Section 7.1 we will show that the problem of computing an approximate solution to a pessimistic estimator instance reduces to NextBitUNPRED. In Section 7.2 we will state the reduction in [MPR26] from GraphSparsification to the problem of computing an approximate solution to a pessimistic estimator instance. We will stitch this with our reduction from Section 7.1 to show that GraphSparsification $\leq_{\text{L-uniform } \mathsf{TC}^1}$ $\text{NextBitUNPRED}^{\mathsf{TC}^1}$. Finally, in Section 7.3, we will show that NextBitUNPRED itself reduces to the problem of computing an approximate solution to a pessimistic estimator instance. Thus, Sections 7.1 and 7.3 together show an equivalence between NextBitUNPRED and the problem of computing an approximate solution to a pessimistic estimator instance, up to parameters.

Before we proceed further, we address a technicality. We will deal with problems where the function F is explicitly computed by a circuit. Note that the input to F is not of fixed length. Thus, when we refer to a circuit C computing F, we mean that C has $n + \lceil \log(n+1) \rceil$ input gates, where the first $\lceil \log(n+1) \rceil$ bits of any input are interpreted as an integer $i \in \{0, 1, \ldots, n\}$. For an input $(i, s \in \{0,1\}^n)$ to C, we have $C(i,s) = F(s_{\leq i})$. That is, C evaluates the function F on the first i bits of the string s. Moreover, in all of our computational theorems, each output of C, and thus of F, will be a positive rational encoded as a pair of numerator-denominator integers, instead of arbitrary positive reals. We will follow this convention for the entire section.

Now, the following two definitions together define the problem of GraphSparsification. Note that, in the end, we will largely just cite the reduction of [MPR26]. Thus, it is not strictly necessary to understand these definitions. Nevertheless, we include them for the interested reader.

Definition 7.2 (Spectral Approximation)**.** Let G be an undirected graph with Laplacian L_G, and let $\epsilon > 0$. Let H be a weighted undirected graph such that H has the same vertex set as G and the edge weights of H are nonnegative. H is defined to be an ϵ-spectral approximation of G if

$$(1 - \epsilon) L_G \preceq L_H \preceq (1 + \epsilon) L_G,$$

where $\preceq$ denotes the Loewner order on real symmetric matrices: $A \preceq B$ if and only if $B - A$ is positive semidefinite.

Definition 7.3 (GraphSparsification)**.** The graph spectral sparsification problem is a search problem defined as follows:

1. **Input** : An unweighted undirected simple graph G on n vertices and m edges, and a parameter $\delta \in \mathbb{R}_{>0}$.

2. **Output** : A weighted graph H with $O(n\delta^{-2}\log n)$ edges such that H is a δ-spectral approximation of G.

7.1 Computing Approximate Solutions to Pessimistic Estimator Instances Reduces to NextBitUNPRED

We begin by defining, for any pessimistic estimator instance F, the family of next-bit predictors that will be the result of our reduction. This is done in Definition 7.4.

The intuition behind this definition is as follows. We are given a pessimistic estimator instance F, and we want to define a set of next-bit predictors $\mathcal{P}(F)$ such that, for any multiset S which does *not* contain an approximate solution to F, the predictors $\mathcal{P}(F)$ perform well on S. In order to do this, we define one predictor P_i for each index $i \in [n]$. Given as input $s_{<i}$ for a string $s \in \{0,1\}^n$, P_i must predict s_i. P_i works as follows: for each $b \in \{0,1\}$, it produces as output the bit b with probability $F(s_{<i} \circ b)/(F(s_{<i} \circ 0) + F(s_{<i} \circ 1))$.

To see why this works, let's consider a string $s \in \{0,1\}^n$ such that s is far from a good solution to the pessimistic estimator instance F. Consider all prefixes of s: $s_{\leq 0} = \emptyset, s_{\leq 1}, \ldots, s_{\leq n-1}, s_{\leq n} = s$. Since $F(s)$ is much larger than $F(s_{\leq 0})$, there must exist $i \in [n]$ such that $F(s_{\leq i})$ is much larger than $F(s_{\leq i-1})$. Now consider the predictor P_i as defined above, acting on the string s. The definition of P_i naturally makes its prediction lean towards the bit b which maximizes $F(s_{\leq i-1} \circ b)$. Since $F(s_{\leq i}) > F(s_{\leq i-1})$, this bit b is exactly s_i. Thus, P_i succeeds on s with good probability.

This is exactly the intuition that we will formalize in Lemma 7.5. We note that this is quite similar to Yao's hybrid argument.

Note that the predictors described above are randomized, while we require deterministic predictors. Thus, we will actually define a family $\mathcal{P}(F,T)$ where T is an integer parameter. T should be thought of as a discretization parameter, which is used to define deterministic variants of the randomized predictors $\mathcal{P}(F)$ described above. Larger values of T will result in better predictors at the cost of a larger family.

Definition 7.4 (Next-Bit Predictors Corresponding to a Pessimistic Estimator)**.** Let $n \in \mathbb{N}$, let $F : S_n \to \mathbb{R}_{>0}$ be a pessimistic estimator instance over $\{0,1\}^n$ as in Definition 7.1, and let $T \in \mathbb{N}$ be a parameter.

We define the family $\mathcal{P}(F,T) \coloneqq \{P_{ij} \mid i \in [n], j \in [T]\}$ of next-bit predictors, where P_{ij} is a next-bit predictor at index i defined as follows: For $s \in \{0,1\}^{i-1}$,

$$P_{ij}(s) \coloneqq \mathbb{1}\left[j \leq \lfloor \frac{1}{2} + \frac{T \cdot F(s \circ 1)}{F(s \circ 0) + F(s \circ 1)} \rfloor\right].$$

Now, we show that the predictors $\mathcal{P}(F,T)$ indeed constitute a valid reduction.

Lemma 7.5. *Let $n \in \mathbb{N}$ and let $F : S_n \to \mathbb{R}_{>0}$ be a pessimistic estimator instance over $\{0,1\}^n$ as in Definition 7.1. Let $\alpha, \epsilon \in \mathbb{R}_{>0}$, and $T \in \mathbb{N}$ be parameters such that*

$$\epsilon \leq \frac{\ln \alpha}{2n} - \frac{1}{2T}. \tag{6}$$

Let $S \subseteq \{0,1\}^n$ be a multiset such that $\mathsf{adv}_{U(S)}(\mathcal{P}(F,T)) < \epsilon$. There exists $s \in S$ such that s is an α-approximate solution to the pessimistic estimator instance F.

Proof. Fix $i \in [n]$ and a prefix $s \in \{0,1\}^{i-1}$, and write

$$p \coloneqq \frac{F(s \circ 1)}{F(s \circ 0) + F(s \circ 1)}, \qquad r \coloneqq \left\lfloor Tp + \frac{1}{2} \right\rfloor.$$

By definition, $P_{ij}(s) = 1$ if and only if $j \leq r$. Since j is an integer, we get

$$P_{ij}(s) = 1 \iff F(s \circ 1) \geq \frac{(2j-1)}{2T} \cdot (F(s \circ 0) + F(s \circ 1)).$$

Note that exactly r of the predictors $\{P_{ij}\}_{j=1}^{T}$ predict $P_{ij}(s) = 1$. Moreover, rounding gives

$$Tp - \frac{1}{2} < r \leq Tp + \frac{1}{2} \quad \implies \quad \left|\frac{r}{T} - p\right| \leq \frac{1}{2T}.$$

Next, let $s \in \{0,1\}^i$, with p and r defined as above for the prefix $s_{<i}$. If $s_i = 1$, the success probability of a uniformly random predictor from $\{P_{ij}\}_{j=1}^{T}$ on s is $\frac{r}{T} \geq p - \frac{1}{2T}$. If $s_i = 0$, it is $1 - \frac{r}{T} \geq 1 - p - \frac{1}{2T}$. Therefore,

$$\begin{aligned}
\Pr_{j \in [T]}[P_{ij}(s_{<i}) = s_i] &\geq \frac{F(s)}{F(s_{<i} \circ 0) + F(s_{<i} \circ 1)} - \frac{1}{2T} \\
&\geq \frac{F(s)}{2F(s_{<i})} - \frac{1}{2T}. && \text{(by Equation 5)} \qquad (7)
\end{aligned}$$

Now, let $s \in \{0,1\}^n$ be a string which is **not** an α-approximate solution to F. That is,

$$F(s) > \alpha \cdot F(\emptyset).$$

We have

$$\begin{aligned}
\Pr_{i \in [n], j \in [T]}[P_{ij}(s_{<i}) = s_i] &= \frac{1}{n} \sum_{i=1}^{n} \Pr_{j \in [T]}[P_{ij}(s_{<i}) = s_i] \\
&\geq \left(\frac{1}{n} \sum_{i=1}^{n} \frac{F(s_{\leq i})}{2F(s_{<i})} \right) - \frac{1}{2T} && \text{(By Equation 7)} \\
&\geq \left(\frac{F(s)}{F(\emptyset)2^n} \right)^{1/n} - \frac{1}{2T} && \text{(By AM-GM Inequality)} \\
&\geq \left(\frac{\alpha}{2^n} \right)^{1/n} - \frac{1}{2T} \\
&= \frac{\alpha^{1/n}}{2} - \frac{1}{2T} \\
&\geq \frac{1}{2} + \frac{\ln(\alpha)}{2n} - \frac{1}{2T} && \text{(By } e^x \geq 1 + x\text{)} \\
&\geq \frac{1}{2} + \epsilon. && \text{(By Equation 6)}
\end{aligned}$$

This implies that for any multiset $S \subseteq \{0,1\}^n$ such that $\forall s \in S$, s is **not** an α-approximate solution to F, we have

$$\Pr_{s \in S, i \in [n], j \in [T]}[P_{ij} \text{ succeeds on } s] \geq \frac{1}{2} + \epsilon.$$

This implies that there exists $P_{ij} \in \mathcal{P}(F, T)$ such that

$$\Pr_{s \in S}[P_{ij} \text{ succeeds on } s] \geq \frac{1}{2} + \epsilon$$

$$\implies \mathsf{adv}_{U(S)}(\mathcal{P}(F,T)) \geq \epsilon.$$

Thus, for any multiset S with $\mathsf{adv}_{U(S)}(\mathcal{P}(F,T)) < \epsilon$, there must exist $s \in S$ such that s is an α-approximate solution to F. This completes the proof. □

Finally, we show that this reduction can be performed on explicit circuits without blowing up their size or depth.

Theorem 7.6. *There exists a logspace algorithm which takes as input $n \in \mathbb{N}$, a circuit F which computes a pessimistic estimator instance over $\{0,1\}^n$, and a parameter $T \in \mathbb{N}$ encoded in unary. It produces as output the set of predictors $\mathcal{P}(F,T)$ from Definition 7.4 in the form of circuits, such that:*

1. *Each predictor $P_{ij} \in \mathcal{P}(F,T)$ has*
$$\mathsf{size}(P_{ij}) \leq \mathrm{poly}(\mathsf{size}(F) + \log T) \qquad \text{and} \qquad \mathsf{depth}(P_{ij}) \leq O(\mathsf{depth}(F)).$$
2. *For any parameters $0 < \epsilon < 1$ and $\alpha \geq 1$ such that*
$$\epsilon \leq \frac{\ln \alpha}{2n} - \frac{1}{2T},$$
and any multiset $S \subseteq \{0,1\}^n$ such that $\mathsf{adv}_{U(S)}(\mathcal{P}(F,T)) < \epsilon$, there exists $s \in S$ such that s is an α-approximate solution to the pessimistic estimator instance F.

Proof. $\mathcal{P}(F,T)$ is of course exactly the set of predictors defined in Definition 7.4. The size and depth bounds of the first part are easy to see: the predictor P_{ij} on an input s simply computes the values $F(s \circ 0)$ and $F(s \circ 1)$ in parallel, and then produces as output the result of a simple comparison involving the aforementioned values, T, and j. The second part follows directly from Lemma 7.5. □

7.2 Graph Sparsification Reduces to NextBitUNPRED$^{\mathsf{TC}^1}$

In this section we use the reduction by Marcussen, Pyne, and Rubinfeld [MPR26], which shows that GraphSparsification reduces to the problem of computing an approximate solution to a TC^1 pessimistic estimator instance, and our reduction from Section 7.1, to show that GraphSparsification reduces to NextBitUNPRED$^{\mathsf{TC}^1}$.

The following theorem was proven by Marcussen, Pyne, and Rubinfeld [MPR26]. It is mainly Theorem 4.1 of their paper, but we combine it with additional details from other parts to obtain a clean statement that will suffice for our reduction.

Theorem 7.7 ([MPR26])**.** *There exists a logspace algorithm which takes as input a simple unweighted undirected graph G on n vertices and m edges and a parameter $0 < \epsilon < 1/2$, and produces as output TC^1 circuits $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ such that:*

1. *$\mathcal{A}$ produces as output a nonnegative integer $A(G,\epsilon) = \mathrm{poly}(n,m)$.*
2. *$\mathcal{B}$ takes as input a binary string s of length at most $A(G,\epsilon)$. It produces as output a value denoted by $F_G^\epsilon(s) \in \mathbb{R}_{>0}$ such that the function*
$$F_G^\epsilon : S_{A(G,\epsilon)} \to \mathbb{R}_{>0}$$
is a pessimistic estimator instance over $\{0,1\}^{A(G,\epsilon)}$.

3. *$\mathcal{C}$ takes as input a string $s \in \{0,1\}^{A(G,\epsilon)}$. It produces as output a simple edge-weighted undirected graph $H(G,s,\epsilon)$ such that:*

   (a) *(1) $V(H(G,s,\epsilon)) = V(G)$, (2) $E(H(G,s,\epsilon)) \subseteq E(G)$, and (3) all edge weights of $H(G,s,\epsilon)$ lie in $[n]$.*

   (b) *For every integer $k \in \mathbb{Z}_{>0}$, if s is an n^k-approximate solution to the pessimistic estimator instance F^{ϵ}_G, the subgraph $H(G,s,\epsilon)$ has $O((1+2^k\epsilon)n\epsilon^{-2}\log n)$ edges and is a $2^k\epsilon$-spectral approximation of G.*

Therefore, GraphSparsification on an input (G,δ) with $0 < \delta < 1$ reduces in L-uniform TC^1 to the problem of computing an n-approximate solution to a pessimistic estimator instance. The pessimistic estimator instance is over $\{0,1\}^{\mathrm{poly}(n,m)}$ and the pessimistic estimator function is computable by a TC^1 circuit. This follows by invoking Theorem 7.7 with parameters $k \coloneqq 1$ and $\epsilon \coloneqq \delta/2$.

Theorem 7.8.

$$\mathsf{GraphSparsification} \leq_{\mathsf{L\text{-}uniform}\ \mathsf{TC}^1} \mathsf{NextBitUNPRED}^{\mathsf{TC}^1}.$$

Proof. Let (G,δ) be a GraphSparsification instance, where $G = (V,E)$, $|V| = n$, $|E| = m$, and $\delta \in \mathbb{R}_{>0}$. Theorem 7.7 reduces GraphSparsification to the problem of computing an n-approximate solution to a pessimistic estimator instance F over $\{0,1\}^{A(G,\delta/2)}$, where $A(G,\delta/2) = \mathrm{poly}(n,m)$ and F is a circuit with $\mathsf{depth}(F) = O(\log(n+m))$ and $\mathsf{size}(F) = \mathrm{poly}(n)$.

We first handle some edge cases: if $\delta \geq 1$, we can return the empty graph on vertex set $V(G)$. Else, if $n \leq 2$, we can return G. Finally, if $A(G,\delta/2) = 0$, we can simply return $\mathcal{C}(\emptyset)$. From here on, we assume that $\delta < 1$, $n \geq 3$, and $A(G,\delta/2) \geq 1$.

Set $T \coloneqq 2A(G,\delta/2) = \mathrm{poly}(n,m)$ and $\epsilon \coloneqq 0.25/A(G,\delta/2)$. By Theorem 7.6, there exists a logspace algorithm which takes as input F and T, and produces as output circuits computing a set of next-bit predictors $\mathcal{P}(F,T)$, such that the circuits have $\mathsf{size} = \mathrm{poly}(n,m)$ and $\mathsf{depth} = O(\log(n+m))$. Theorem 7.6 guarantees that for any $S \subseteq \{0,1\}^{A(G,\delta/2)}$ which is a solution to the $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1}$ instance given by $\mathcal{P}(F,T)$ and ϵ, there exists $s \in S$ which is an n-approximate solution to the pessimistic estimator instance F.

Thus, given a polynomial-size solution S to this $\mathsf{NextBitUNPRED}^{\mathsf{TC}^1}$ instance, an L-uniform TC^1 algorithm could simply check in parallel for each $s \in S$ whether $F(s) \leq n \cdot F(\emptyset)$. By Theorem 7.7, such an s suffices to solve GraphSparsification in L-uniform TC^1.

This completes the proof. □

7.3 NextBitUNPRED Reduces to Computing Approximate Solutions to Pessimistic Estimator Instances

In this section, we show that NextBitUNPRED itself also reduces to the problem of computing approximate solutions to pessimistic estimator instances. The construction here is very similar to those in Wigderson and Xiao [WX08], and was directly inspired by Li [Li21].

We begin by formally defining the pessimistic estimator instance that will be the result of our reduction. We are given a NextBitUNPRED instance over $\{0,1\}^n$. Let's say the predictors are $P_1,\ldots,P_m$, where P_i predicts index $\mathsf{ind}_i \in [n]$. For such an instance, the pessimistic estimator F will need to be over much larger strings of length kn, for a parameter k. We will then interpret a string $S \in \{0,1\}^{kn}$ as the concatenation of k strings $s^1,\ldots,s^k \in \{0,1\}^n$. Similar to Section 3, we

will interpret S to contain $s^1, \ldots, s^k$ in column-major order. That is, the first k bits of S are the first bit of each string, and so on.

Now, to define the pessimistic estimator itself. The estimator F is the sum of m estimators, one for each predictor. Let us describe the estimator F_i corresponding to just one predictor P_i. Let S' be a prefix of $S \in \{0,1\}^{kn}$. We will define $F_i(S')$. In order to do this, the estimator needs two values in particular: x and j. x is the number of strings $s \in S$ such that the bits of $s_{\leq \mathsf{ind}_i}$ have already been completed in S', and they result in P_i succeeding. The second value, j, is simply the number of strings $s \in S$ whose first ind_i bits have already been filled into S'. This is similar to x, but also includes strings on which P_i is now guaranteed to fail. For a fixed parameter ϵ, we define

$$F_i(S') \coloneqq (1+\epsilon)^x \left(\frac{2+\epsilon}{2}\right)^{k-j}.$$

We emphasize that x and j depend on S'. It is a simple exercise to verify that this is indeed a pessimistic estimator. Claim 7.10 shows that the initial value $F_i(\emptyset)$ of this estimator is $\leq \exp(k\epsilon/2)$. Theorem 7.11 will show that if the multiset corresponding to a string $S \in \{0,1\}^{kn}$ is not a solution to NextBitUNPRED with parameter $\epsilon/2$, then $F_i(S) \geq \exp(k\epsilon(1+\epsilon)/(2+\epsilon))$ for some $i \in [m]$. Thus, non-solutions S result in pessimistic estimator values $F_i(S)$ that are larger than $F_i(\emptyset)$. We will use this fact to then complete the reduction in Corollary 7.12.

Note that this reduction depends on parameters k and ϵ. Thus, the result of our reduction will be called $F(\mathcal{P}, k, \epsilon)$, and $F(P, k, \epsilon)$ for a fixed predictor $P \in \mathcal{P}$.

Definition 7.9 (Pessimistic Estimators Corresponding to Next-Bit Predictors)**.** Let $\mathcal{P} = \bigcup_{i=1}^{n} \mathcal{P}_i$ be a non-empty set of next-bit predictors over $\{0,1\}^n$ where $\mathcal{P}_i$ is the set of predictors at index i.

For a parameter $k \in \mathbb{N}$, a string $s \in S_{kn}$, and $i \in [n]$, let

$$j_{\max}(k, s, i) \coloneqq \min\{k, \max\{0, |s| - k(i-1)\}\}.$$

Furthermore, for $1 \leq j \leq j_{\max}(k, s, i)$, we define

$$\mathsf{str}(k, s, i, j) \coloneqq s_j \circ s_{k+j} \circ s_{2k+j} \circ \cdots \circ s_{(i-1)k+j}.$$

For a predictor $P \in \mathcal{P}_i$ and a parameter $\epsilon \in \mathbb{R}_{>0}$, we define a pessimistic estimator $F(P, k, \epsilon)$ over $\{0,1\}^{kn}$ as follows: For $s \in S_{kn}$, let

$$X(P, k, s) \coloneqq \sum_{j=1}^{j_{\max}(k,s,i)} \mathbb{1}[P(\mathsf{str}(k, s, i, j)_{<i}) = \mathsf{str}(k, s, i, j)_i],$$

and

$$F(P, k, \epsilon)(s) \coloneqq (1+\epsilon)^{X(P,k,s)} \left(\frac{2+\epsilon}{2}\right)^{k - j_{\max}(k,s,i)}.$$

Finally, we define a single pessimistic estimator instance $F(\mathcal{P}, k, \epsilon)$ corresponding to the entire set of predictors $\mathcal{P}$, as follows:

$$F(\mathcal{P}, k, \epsilon)(s) \coloneqq \sum_{P \in \mathcal{P}} F(P, k, \epsilon)(s).$$

It is a simple exercise to verify that each $F(P, k, \epsilon)$ satisfies Equation 5, and is therefore indeed a pessimistic estimator. It is equally simple to verify that the sum of pessimistic estimators (i.e., $F(\mathcal{P}, k, \epsilon)$) is itself a pessimistic estimator. In the following claim we show an upper bound on $F(P, k, \epsilon)(\emptyset)$.

Claim 7.10. *Let F be as in Definition 7.9. Let $\mathcal{P}$ be a set of next-bit predictors over $\{0,1\}^n$. Let $k \in \mathbb{N}$ and $\epsilon \in \mathbb{R}_{>0}$. Then, for all $P \in \mathcal{P}$,*

$$F(P,k,\epsilon)(\emptyset) \leq \exp(\frac{k\epsilon}{2}).$$

Proof. Note that $j_{\max}(k,\emptyset,i) = 0$ for all $k \in \mathbb{N}$ and $i \in [n]$. Therefore, $X(P,k,\emptyset) = 0$. This implies that

$$\begin{aligned} F(P,k,\epsilon)(\emptyset) &= \left(\frac{2+\epsilon}{2}\right)^k \\ &= \exp(k\ln(1+\frac{\epsilon}{2})) \\ &\leq \exp(\frac{k\epsilon}{2}). && \text{(Because } 1+x \leq e^x\text{)} \end{aligned}$$

This completes the proof. □

In the following theorem we show that any string $s \in \{0,1\}^{kn}$ such that $F(\mathcal{P},k,\epsilon)(s)$ is sufficiently small must represent a solution to the NextBitUNPRED instance $\mathcal{P}$ with parameter $\epsilon/2$.

Theorem 7.11. *Let F and* str *be as in Definition 7.9. Let $\mathcal{P}$ be a set of next-bit predictors over $\{0,1\}^n$. Let $k \in \mathbb{N}$, $\epsilon \in \mathbb{R}_{>0}$, and $s \in \{0,1\}^{kn}$ be such that $F(\mathcal{P},k,\epsilon)(s) < \exp(k\epsilon(1+\epsilon)/(2+\epsilon))$. Then, the multiset $S \coloneqq \{\mathsf{str}(k,s,n,j) \mid j \in [k]\}$ satisfies*

$$\mathsf{adv}_{U(S)}(\mathcal{P}) < \frac{\epsilon}{2}.$$

Proof. Assume to the contrary that

$$\mathsf{adv}_{U(S)}(\mathcal{P}) \geq \frac{\epsilon}{2}.$$

This implies that there exists $P \in \mathcal{P}$ such that

$$\mathsf{adv}_{U(S)}(P) \geq \frac{\epsilon}{2}.$$

Let's say that $P \in \mathcal{P}_i$. Simply observe that $j_{\max}(k,s,i) = k$. Therefore,

$$\begin{aligned} X(P,k,s) &= \sum_{j=1}^{k} \mathbb{1}[P \text{ succeeds on } \mathsf{str}(k,s,n,j)] \\ &\geq \frac{k}{2}(1+\epsilon). \end{aligned}$$

Now,

$$\begin{aligned} F(P,k,\epsilon)(s) &= (1+\epsilon)^{X(P,k,s)} \left(\frac{2+\epsilon}{2}\right)^{k-j_{\max}(k,s,i)} \\ &= (1+\epsilon)^{X(P,k,s)} \\ &\geq (1+\epsilon)^{k(1+\epsilon)/2} && \text{(Because } X(P,k,s) \geq k(1+\epsilon)/2\text{)} \\ &= \exp(\frac{k(1+\epsilon)}{2}\ln(1+\epsilon)) \\ &\geq \exp(\frac{k\epsilon(1+\epsilon)}{2+\epsilon}). && \text{(Because } \ln(1+\epsilon) \geq \frac{2\epsilon}{2+\epsilon}\text{)} \end{aligned}$$

Therefore, this implies that $F(\mathcal{P}, k, \epsilon)(s) \geq F(P, k, \epsilon)(s) \geq \exp(k\epsilon(1+\epsilon)/(2+\epsilon))$, which is a contradiction. This completes the proof. □

In the following corollary, for an appropriate choice of the parameter k, we show that the lower bound for non-solutions from Theorem 7.11 is more than twice the upper bound on $F(\mathcal{P}, k, 2\epsilon)(\emptyset)$ obtained from Claim 7.10. Thus, any 2-approximate solution to the pessimistic estimator instance $F(\mathcal{P}, k, 2\epsilon)$ corresponds to a solution to the NextBitUNPRED instance with parameter ϵ.

Corollary 7.12. *Let F and str be as in Definition 7.9. Let $\mathcal{P}$ be a set of m next-bit predictors over $\{0,1\}^n$. Let $0 < \epsilon < 1/2$, and $k \in \mathbb{N}$ such that $k \geq \ln(2.01m)(1+\epsilon)/\epsilon^2$. Let $s \in \{0,1\}^{kn}$ be a 2-approximate solution to the pessimistic estimator instance $F(\mathcal{P}, k, 2\epsilon)$. Then, the multiset $S \coloneqq \{\mathsf{str}(k, s, n, j) \mid j \in [k]\}$ satisfies*

$$\mathsf{adv}_{U(S)}(\mathcal{P}) < \epsilon.$$

Proof. For all $P \in \mathcal{P}$, due to Claim 7.10, we have

$$F(P, k, 2\epsilon)(\emptyset) \leq \exp(k\epsilon).$$

Thus,

$$F(\mathcal{P}, k, 2\epsilon)(s) \leq 2m \exp(k\epsilon).$$

Note that

$$\begin{aligned}
\frac{2m\exp(k\epsilon)}{\exp(2k\epsilon(1+2\epsilon)/(2+2\epsilon))} &= 2m\exp(k\epsilon - \frac{k\epsilon(1+2\epsilon)}{1+\epsilon}) \\
&= 2m\exp(\frac{-k\epsilon^2}{1+\epsilon}) \\
&\leq 2/2.01 && \text{(By plugging in the lower bound on } k.) \\
&< 1.
\end{aligned}$$

Thus, we have

$$F(\mathcal{P}, k, 2\epsilon)(s) < \exp(2k\epsilon(1+2\epsilon)/(2+2\epsilon)).$$

Then, by Theorem 7.11, the corollary follows. □

In the following observation we show that the circuit corresponding to the pessimistic estimator F can be constructed efficiently, the circuit itself incurs only a polynomial blow-up in size compared to the predictors, and only a constant factor blow-up in depth.

Observation 7.13. *There exists a logspace algorithm which takes as input $n \in \mathbb{N}$, a set of m next-bit predictors $\mathcal{P} = \bigcup_{i=1}^{n} \mathcal{P}_i$ over $\{0,1\}^n$ in the form of their circuits, a unary-encoded integer $k \in \mathbb{N}$, and a unary-encoded parameter $0 < \epsilon \leq 1$. The algorithm produces as output a circuit C_F computing the function $F(\mathcal{P}, k, \epsilon)$ from Definition 7.9 such that*

$$\mathsf{size}(C_F) = \mathrm{poly}(n, m, k, \epsilon^{-1}, \sum_{P \in \mathcal{P}} \mathsf{size}(P))$$

and

$$\mathsf{depth}(C_F) \leq O(\max_{P \in \mathcal{P}} \mathsf{depth}(P)).$$

Proof. Let $i \in [n]$ and $P \in \mathcal{P}_i$. Note that the circuit which computes $F(P, k, \epsilon)$ on a string $s \in S_{kn}$ is straightforward: it first computes $X(P, k, s)$ by querying P on $\mathsf{str}(k, s, i, j)_{<i}$ for each

$j \in [j_{\max}(k, s, i)]$ in parallel. Once it has computed $X(P, k, s)$, then computing $F(P, k, \epsilon)(s)$ only requires simple arithmetic which can be performed by a polynomial-size constant-depth circuit [HAM02]. The final step to computing $F(\mathcal{P}, k, \epsilon)$ is just summing these values over all $P \in \mathcal{P}$, which can again be done by a polynomial-size constant-depth circuit.

This completes the proof. □

Therefore, by Observation 7.13 and Corollary 7.12, we have a reduction from NextBitUNPRED to the problem of computing an approximate solution to a pessimistic estimator instance, where the pessimistic estimator is computed by a circuit of size at most polynomial in n, the sum of sizes of the predictor circuits and ϵ^{-1}, and depth at most a constant factor larger than the maximum depth of the predictor circuits. Combined with the reduction from Section 7.1, this shows that the problem of computing an approximate solution to a pessimistic estimator instance is, up to a small change in parameters, equivalent to NextBitUNPRED.

8 BlockLossyCoding Reduces to UNPRED

In this section we will show that the catalytic logspace algorithms for bipartite matching [AM25], linear matroid intersection [AAV26], and general matching [CDK+26] reduce in TC^1 to $\mathsf{UNPRED}^{\mathsf{TC}^1}$. We will prove this fact by identifying a common structural property in all three of these algorithms: these algorithms can be interpreted as reductions to a special form of the LossyCoding problem (as defined in Definition 4.9), that we call BlockLossyCoding.

An arbitrary LossyCoding instance has no promised structure in its compression and decompression. This lack of structure is exactly what makes it a difficult problem. The compress-or-random algorithms that we mentioned above, on the other hand, use compression in a very specific way. In all of these cases, the input to the compressor is interpreted as the concatenation of n strings, each of length ℓ. On any input string, the compressor selects a specific block, and then *only compresses the selected block*. The other blocks are left unchanged. Note that the selected block must be compressed enough to also store its index, which means that we must have $\ell > \log n$. We call such an instance of LossyCoding an (n, ℓ) BlockLossyCoding instance.

We will show that, when ℓ is logarithmic in the input size, (n, ℓ) BlockLossyCoding admits a simple reduction to UNPRED. When the compressor and decompressor are both TC^1 circuits, this reduction will result in TC^1 predictors. The catalytic search-to-decision reduction for the isolation lemma by Arvind, Chakraborty, and Datta [ACD26] follows the same form but, for the sake of succinctness, we will only prove the $\mathsf{UNPRED}^{\mathsf{TC}^1}$ reduction for the three algorithms stated above.

We begin by formally defining BlockLossyCoding.

Definition 8.1 (BlockLossyCoding)**.** An (n, ℓ) BlockLossyCoding instance is a LossyCoding instance

1. $C : (\{0,1\}^\ell)^n \to \{0,1\}^{\lceil \log(n+1) \rceil} \times \{0,1\}^k \times (\{0,1\}^\ell)^{n-1}$ and
2. $D : \{0,1\}^{\lceil \log(n+1) \rceil} \times \{0,1\}^k \times (\{0,1\}^\ell)^{n-1} \to (\{0,1\}^\ell)^n$,

where $k \coloneqq \ell - 1 - \lceil \log(n+1) \rceil \geq 0$. The instance must satisfy the following two conditions:

1. For all $x = (x_1, \ldots, x_n) \in (\{0,1\}^\ell)^n$,
$$(C(x))_1 \in [n].$$
We define $\mathsf{ind}_C(x) \coloneqq (C(x))_1 \in [n]$ and $\mathsf{advice}_C(x) \coloneqq (C(x))_2 \in \{0,1\}^k$.

2. For all $x = (x_1, \ldots, x_n) \in (\{0,1\}^\ell)^n$, let $C(x) = (\mathsf{ind}_C(x), \mathsf{advice}_C(x), (y_1, \ldots, y_{n-1}))$. Then
$$(y_1, \ldots, y_{n-1}) = (x_1, \ldots, x_{\mathsf{ind}_C(x)-1}, x_{\mathsf{ind}_C(x)+1}, \ldots, x_n).$$

We briefly explain the connection between this definition and our informal description. The idea is that the $\mathsf{ind}_C(x)^{th}$ block of the input string x is compressed down to the strings $\mathsf{ind}_C(x)$ and $\mathsf{advice}_C(x)$, whose combined length is strictly less than ℓ. A possible misconception is that one could try to solve BlockLossyCoding by interpreting it as LossyCoding on a fixed block. A block consists of only ℓ bits. A brute-force search over all length ℓ strings could then solve the problem efficiently. The reason this does not work is that $\mathsf{ind}_C(x)$ itself depends on x. Thus, it is not possible to restrict the problem to only one block, because the choice of the block itself depends on the input string.

Now, we prove our reduction from BlockLossyCoding to UNPRED. We begin by defining the predictors that will appear as a result of this reduction. Intuitively, one should imagine the predictors to be the result of a two-step reduction.

First, imagine that the predictors, instead of acting over a string of length $n\ell$ with the alphabet $\{0,1\}$, are acting over a string of length n with the alphabet $\{0,1\}^\ell$. That is, instead of guessing the value of a single bit, each predictor guesses the value of a whole block. Then, we could define a set of predictors $\{P_{it} \mid i \in [n], t \in \{0,1\}^k\}$ where P_{it} guesses the ℓ-bit string in the ith block. That is, given a string $s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n$ where each $s_i \in \{0,1\}^\ell$, P_{it} tries to predict s_i. To do this, it uses the decompressor D, plugging in the string $t \in \{0,1\}^k$ as the advice, and $i \in [n]$ as the index. Thus,
$$P_{it}(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n) \coloneqq D(i, t, s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)_i.$$
By definition of BlockLossyCoding, for any string $s \in \mathsf{NonSol}(C, D)$, there must be $i \in [n]$ and $t \in \{0,1\}^k$ such that P_{it} succeeds on s (where success means guessing the whole block correctly). In particular, this occurs when $i = \mathsf{ind}_C(s)$ and $t = \mathsf{advice}_C(s)$. Therefore, the strong pigeonhole principle implies that for a multiset $S \subseteq \mathsf{NonSol}(C, D) \subseteq (\{0,1\}^\ell)^n$, there must exist $i \in [n]$ and $t \in \{0,1\}^k$ such that P_{it} succeeds on $s \sim U(S)$ with probability at least $1/(n2^k)$. Plugging in the definition of k, this success probability is $\geq 2/2^\ell$. For bit predictors, this is not a very high success probability. However, note that we are not predicting just a single bit correctly, but rather an entire block. The probability of random guessing being correct on an entire block of ℓ bits is $1/2^\ell$. Thus, these predictors do indeed beat random guessing by a margin of $1/2^\ell$.

The second step is simply to transfer this $1/2^\ell$ margin of success over random guessing into bit predictors. To do this, the idea is as follows: instead of predicting the entire block at once, we define predictors P_{itj}, where i and t are as above, and $j \in [\ell]$. The predictor P_{itj} will try to predict only the jth bit of the ith block. Of course, if P_{itj} just simulated P_{it} and output its jth bit, then it would not necessarily achieve a better success probability. Instead, it works as follows: P_{itj} simulates P_{it}, but then also checks whether the output of P_{it} has succeeded on the first $j-1$ bits of the block. If any of those bits were predicted wrongly by P_{it}, then P_{itj} just produces as output a uniformly random bit. Else, it produces as output the jth bit of P_{it}. In order to simulate this random guessing, we will actually define two predictors for each i, t, and j: P_{itj0} and P_{itj1}.

The following definition formally defines these predictors, and then Lemma 8.3 will prove that they indeed correspond to a valid reduction.

Definition 8.2 (Predictors Corresponding to a BlockLossyCoding Instance)**.** Let (C, D) be an (n, ℓ)

BlockLossyCoding instance, and let $k \coloneqq \ell - 1 - \lceil \log(n+1) \rceil$. We define

$$\mathcal{P}(C, D) \coloneqq \{P_{itjb} \mid i \in [n], t \in \{0,1\}^k, j \in [\ell], b \in \{0,1\}\}.$$

P_{itjb} is a middle-bit predictor over $\{0,1\}^{n\ell}$ at index $(i-1)\ell + j$. For $s = s_1 \circ \cdots \circ s_n$ with each $s_i \in \{0,1\}^\ell$, let

$$d_{it}(s) \coloneqq D(i, t, (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n))_i.$$

Then

$$P_{itjb}(s_{-((i-1)\ell+j)}) \coloneqq \begin{cases} d_{it}(s)_j, & \text{if } (s_i)_{<j} = d_{it}(s)_{<j}, \\ b, & \text{otherwise.} \end{cases}$$

Lemma 8.3. *Let (C, D) be an (n, ℓ)* BlockLossyCoding *instance, and let $\mathcal{P}(C, D)$ be the predictors from Definition 8.2. For any non-empty multiset $S \subseteq (\{0,1\}^\ell)^n$ such that*

$$\mathsf{adv}_{U(S)}(\mathcal{P}(C, D)) < \frac{1}{2^{\ell+1}},$$

S contains a solution to the BlockLossyCoding *instance (C, D).*

Proof. Let $k \coloneqq \ell - 1 - \lceil \log(n+1) \rceil$ and $\epsilon \coloneqq 1/2^{\ell+1}$. Let $S \subseteq (\{0,1\}^\ell)^n$ such that

$$\mathsf{adv}_{U(S)}(\mathcal{P}(C, D)) < \epsilon.$$

Assume for the sake of contradiction that $S \subseteq \mathsf{NonSol}(C, D)$. Then, for every $s \in S$, the pair $i \coloneqq \mathsf{ind}_C(s)$ and $t \coloneqq \mathsf{advice}_C(s)$ satisfies

$$d_{it}(s) = D(C(s))_i = s_i.$$

By the strong pigeonhole principle, there must exist a fixed pair $i \in [n]$ and $t \in \{0,1\}^k$ such that

$$\Pr_{s \sim U(S)}[d_{it}(s) = s_i] \geq \frac{1}{n2^k} > \frac{2}{2^\ell}.$$

For this pair, we define

$$\chi_j \coloneqq \Pr_{s \sim U(S)}[(s_i)_{\leq j} = d_{it}(s)_{\leq j}]$$

for $j \in \{1, \ldots, \ell\}$, and $\chi_0 \coloneqq 1$. The strong pigeonhole principle argument showed that

$$\chi_\ell > \frac{2}{2^\ell}.$$

Now, fix any $j \in [\ell]$. Observe that the predictors P_{itj0} and P_{itj1} on a string $s \in (\{0,1\}^\ell)^n$ have two cases: if $d_{it}(s)_{<j} = (s_i)_{<j}$, then both predictors enter the first case of their definition and guess $d_{it}(s)_j$. They *both* succeed exactly when $(s_i)_{\leq j} = d_{it}(s)_{\leq j}$. This happens with probability χ_j. On the other hand, if $(s_i)_{<j} \neq d_{it}(s)_{<j}$, then the predictors enter the second case of their definition and guess 0 and 1 respectively, and exactly one of them succeeds. This happens with probability $(1 - \chi_{j-1})$. Therefore, for any $j \in [\ell]$,

$$\frac{\mathsf{succ}_{U(S)}(P_{itj0}) + \mathsf{succ}_{U(S)}(P_{itj1})}{2} = \chi_j + \frac{1 - \chi_{j-1}}{2} < \frac{1}{2} + \epsilon.$$

The last inequality follows due to the assumption that $\mathsf{adv}_{U(S)}(\mathcal{P}(C, D)) < \epsilon$. Therefore,

$$\chi_j \leq \frac{\chi_{j-1}}{2} + \epsilon.$$

Since $\chi_0 = 1$, this implies that

$$\chi_\ell \leq 2^{-\ell} + 2\epsilon = \frac{2}{2^\ell}.$$

This contradicts the lower bound on χ_ℓ. Thus, S contains a solution to (C, D). □

Lemma 8.4. *There exists an algorithm which takes as input $n, \ell \in \mathbb{N}$, along with an (n, ℓ)* BlockLossyCoding *instance (C, D) in circuit form. The algorithm uses $O(\ell + \log n + \log(\mathsf{size}(C)) + \log(\mathsf{size}(D)))$ work space. The algorithm produces as output, in circuit form, a set of $\leq \ell 2^\ell$ middle-bit predictors, $\mathcal{P}(C, D)$, over $\{0,1\}^{n\ell}$, and $\epsilon \coloneqq 1/2^{\ell+1}$, such that:*

1. *For each predictor $P \in \mathcal{P}(C, D)$, $\mathsf{size}(P) \leq \mathsf{size}(D) + O(\ell)$, and $\mathsf{depth}(P) \leq \mathsf{depth}(D) + O(\log \ell)$.*
2. *For any non-empty multiset $S \subseteq \{0,1\}^{n\ell}$ such that*

$$\mathsf{adv}_{U(S)}(\mathcal{P}(C, D)) < \epsilon,$$

S contains a solution to the BlockLossyCoding *instance (C, D).*

Proof. Let $k \coloneqq \ell - 1 - \lceil \log(n+1) \rceil$. The set $\mathcal{P}(C, D)$ from Definition 8.2 contains at most $\ell n 2^{k+1} \leq \ell 2^\ell$ predictors.

Each predictor P_{itjb} contains the circuit D with its first $\ell - 1$ input bits fixed (to $i \circ t$), and the output restricted to only ℓ of its ℓn output bits, together with $O(\ell)$ gates of depth $O(\log \ell)$ to check the prefix and select the output. These circuits can be produced in the stated work space by enumerating i, t, j, b and copying D. The unary encoding of $1/\epsilon = 2^{\ell+1}$ can also be produced using $O(\ell)$ work space. Thus, the first part of the lemma follows. The second part follows from Lemma 8.3.

This completes the proof. □

Finally, Agarwala and Mertz [AM25] implicitly prove the following reduction from BipMATCH to BlockLossyCoding.

Lemma 8.5 (Adapted from Theorem 6.3 of [AM25])**.** *There exists a logspace algorithm which takes as input an undirected bipartite graph $G = (V, E)$ with n vertices and m edges, and produces three circuits, each of which has $\mathsf{size} = \mathrm{poly}(n+m)$ and $\mathsf{depth} = O(\log(n+m))$:*

1. *C_G and D_G. This is an $(m, \ell = O(\log(n+m)))$ block lossy coding instance.*
2. *Sol_G. This circuit takes as input any $s \in (\{0,1\}^\ell)^m$ which is a solution to the* LossyCoding *instance (C_G, D_G), and produces as output a maximum matching of G.*

Corollary 8.6.

$$\mathsf{BipMATCH} \leq_{\mathsf{L\text{-}uniform}\ \mathsf{TC}^1} \mathsf{UNPRED}^{\mathsf{TC}^1}.$$

Proof. The reduction works as follows: it takes as input a bipartite graph G with n vertices and m edges. First, it simulates the logspace algorithm from Lemma 8.5 to produce the $(m, O(\log(n+m)))$ block lossy instance C_G, D_G, where both C_G and D_G have $\mathsf{size} = O(\mathrm{poly}(n+m))$ and $\mathsf{depth} = O(\log(n+m))$. Then, it uses the reduction from Lemma 8.4 to obtain, using $O(\log(n+m))$ work space, a set of $\leq \mathrm{poly}(n, m)$ predictors $\mathcal{P}(C_G, D_G)$ and $\epsilon = 1/2^{\ell+1} \geq 1/\mathrm{poly}(n+m)$. Each predictor has $\mathsf{size} = O(\mathrm{poly}(n+m))$ and $\mathsf{depth} = O(\log(n+m))$. Together, these predictors define an $\mathsf{UNPRED}^{\mathsf{TC}^1}$ instance.

Let S be a polynomial-size solution to the $\mathsf{UNPRED}^{\mathsf{TC}^1}$ instance. Then the reduction could iterate over all $s \in S$ in parallel and test whether $D_G(C_G(s)) = s$, i.e., whether $s \in \mathsf{NonSol}(C_G, D_G)$.

For the first s it finds such that $s \notin \mathsf{NonSol}(C_G, D_G)$ (by Lemma 8.4, such an s must exist), it could feed this s to the circuit Sol_G, obtained from Lemma 8.5, to obtain a maximum matching of G.

This completes the reduction. □

The algorithm of Agarwala, Alekseev, and Vinciguerra [AAV26] can be reinterpreted as the exact same statement as Lemma 8.5, but for linear matroid intersection. Similarly, the algorithm of Chakraborty, Datta, Kusre, Mukhopadhyay, and Sinhababu [CDK$^+$26] can be reinterpreted as the same statement as Lemma 8.5 but for general matching. Unfortunately, they do not state this as explicitly as Agarwala and Mertz [AM25]. It would obfuscate our contribution to recreate their algorithms here. Thus, we simply state the results we get by reinterpreting their algorithms as reductions to block lossy coding. The *exact* same proof as Corollary 8.6 then implies the following statements.

Corollary 8.7 ([AAV26])**.**

$$\mathsf{LinearMatroidIntersection} \leq_{\mathsf{L\text{-}uniform}\ \mathsf{TC}^1} \mathsf{UNPRED}^{\mathsf{TC}^1}.$$

Corollary 8.8 ([CDK$^+$26])**.**

$$\mathsf{MATCH} \leq_{\mathsf{L\text{-}uniform}\ \mathsf{TC}^1} \mathsf{UNPRED}^{\mathsf{TC}^1}.$$

9 Artificial Intelligence Disclosure

This work was the result of approximately 16 months of research, much of which was spent before LLM models possessed the strength that they now do.

Except for Section 4, LLMs played no role in this paper beyond proofreading, cleaning text, generating tables, and generating TikZ code after being given hand-drawn figures. Section 4 of this paper, in particular the reduction from the restricted version of LossyCoding to UNPRED, was obtained in collaboration with ChatGPT 5.6 Sol Extra High. However, this was extremely far from a one-shot prompt.

To elaborate: at this point, the authors had already obtained every other result in this paper. In particular, we were able to prove that every problem *currently known* to be in CL reduces to $\mathsf{UNPRED}^{\mathsf{TC}^1}$. However, there was exactly one exception: a restricted version of LossyCoding, which was shown to be in SearchCL by Cook et al. [CGM$^+$25]. This was a far more restricted version of LossyCoding than the one we study in this work: one where both the compressor and decompressor must be computable by $O(\log n)$ depth, $n + O(\log n)$ width, and fan-in 2 circuits.

We first gave ChatGPT access to all of our proofs (in a Notion workspace), and asked it to prove that FastCatTape reduces to UNPRED. ChatGPT claimed to have proved this. Upon analyzing ChatGPT's proof, we realized that its proof was too strong: we believed that it could be extended to show that all of ZPP reduces to UNPRED. The authors, along with Edward Pyne, spent significant time understanding whether this was possible. This led us to uncover a minor mathematical error in a previous paper, which ChatGPT had used as a hammer in its incorrect proof. We have already contacted the authors responsible for that paper, and thus we will not name it publicly.

We concluded that this problem was too hard for ChatGPT. Thus, we tempered our expectations and asked it to prove that the extremely restricted version of LossyCoding described above reduces to $\mathsf{UNPRED}^{\mathsf{TC}^1}$. Once again, we gave it access to all of the other proofs in this work. On this prompt, ChatGPT succeeded.

ChatGPT produced a proof that was extremely clever, yet opaque. We initially expected it to be erroneous, as its last attempt was. Thus, we spent a significant amount of time understanding and verifying this proof. In this process, we realized that if one abandoned the TC^1 requirement, then the proof that ChatGPT came up with relies primarily on the in-place computation of the decompressor. We realized that the decompressor of Cook et al. [CLMP25] also has this structure. From here, the authors were able to generalize ChatGPT's idea to obtain the full result of Section 4.

We would like to remark that we put significant effort into the exposition of these results. In particular, ChatGPT did not structure its proof as a reduction to $\mathsf{WeakExcess}$, or even formally define a graph[13]. The reduction to $\mathsf{WeakExcess}$ that we present in this work came as a result of several iterations of this paper that aimed to make the exposition as readable as possible.

The chat session described above consisted of a total of 36 prompts from us, and ChatGPT worked for a total of 35 hours, 33 minutes, and 15 seconds. We gave a link to the entire chat session, along with this AI disclosure section, to another ChatGPT 6.1 Sol Ultra session. It confirmed that this disclosure is a fair and accurate representation of events.

Acknowledgements

Both authors thank Ian Mertz, James Cook, and in particular Edward (Ted) Pyne for extensive discussions. The first author thanks Danupon Nanongkai for extensive discussions and supervision. Both authors thank Julian Casablancas for his underrated album *First Impressions of Earth*, without which this work would not have been possible.

[13] It did occasionally use graph terminology, such as walks and flow, which was comprehensible around 10% of the time.

References

[AAV26] Aryan Agarwala, Yaroslav Alekseev, and Antoine Vinciguerra. Linear Matroid Intersection Is in Catalytic Logspace. In *Innovations in Theoretical Computer Science Conference (ITCS)*, volume 362 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. doi:10.4230/LIPIcs.ITCS.2026.3.

[ACD26] V. Arvind, Srijan Chakraborty, and Samir Datta. Derandomizing isolation in catalytic logspace. *Computing Research Repository (CoRR)*, abs/2512.09374, 2026. URL: https://arxiv.org/abs/2512.09374.

[AM25] Aryan Agarwala and Ian Mertz. Bipartite Matching is in Catalytic Logspace . In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 360–372. IEEE Computer Society, 2025. doi:10.1109/FOCS63196.2025.00022.

[AS16] N. Alon and J.H. Spencer. *The Probabilistic Method*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 2016. URL: https://books.google.de/books?id=Iu5RCgAAQBAJ.

[AV26] Aryan Agarwala and Nithin Varma. Pseudodeterministic Algorithms for Minimum Cut Problems. In *Innovations in Theoretical Computer Science Conference (ITCS)*, volume 362 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. doi:10.4230/LIPIcs.ITCS.2026.4.

[BCE+98] Paul Beame, Stephen Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. The relative complexity of NP search problems. *Journal of Computer and System Sciences (J.CSS)*, 57(1):3–19, 1998. doi:10.1006/jcss.1998.1575.

[BCK+14] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *ACM Symposium on Theory of Computing (STOC)*, pages 857–866, 2014. doi:10.1145/2591796.2591874.

[BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing (SICOMP)*, 13(4):850–864, 1984.

[CDK+26] Srijan Chakraborty, Samir Datta, Aryan Kusre, Partha Mukhopadhyay, and Amit Sinhababu. Maximum matching and related problems in catalytic logspace. *Computing Research Repository (CoRR)*, abs/2604.24275, 2026. URL: https://arxiv.org/abs/2604.24275.

[CGG+26] Abhranil Chatterjee, Sumanta Ghosh, Rohit Gurjar, Roshan Raj, and Thomas Thierauf. Bipartite matching is in NC. *Electronic Colloquium on Computational Complexity (ECCC)*, page 100, 2026.

[CGM+25] James Cook, Surendra Ghentiyala, Ian Mertz, Edward Pyne, and Nathan S. Sheffield. The structure of in-place space-bounded computation. *Computing Research Repository (CoRR)*, abs/2510.12005, 2025. URL: https://arxiv.org/abs/2510.12005.

[CLMP25] James Cook, Jiatu Li, Ian Mertz, and Edward Pyne. The structure of catalytic space: Capturing randomness and time via compression. In *ACM Symposium on Theory of Computing (STOC)*, 2025.

[CM24] James Cook and Ian Mertz. Tree evaluation is in space O(log n · log log n). In *ACM Symposium on Theory of Computing (STOC)*, pages 1268–1278. ACM, 2024. doi:10.1145/3618260.3649664.

[CT21] Lijie Chen and Roei Tell. Simple and fast derandomization from very hard functions: eliminating randomness at almost no cost. In *ACM Symposium on Theory of Computing (STOC)*, pages 283–291. ACM, 2021. doi:10.1145/3406325.3451059.

[DMOZ22] Dean Doron, Dana Moshkovitz, Justin Oh, and David Zuckerman. Nearly optimal pseudorandomness from hardness. *Journal of the ACM (J.ACM)*, 69(6):43:1–43:55, 2022. doi:10.1145/3555307.

[DMOZ26] Dean Doron, Dana Moshkovitz, Justin Oh, and David Zuckerman. Pseudorandomness beating the hybrid argument for insensitive algorithms. *Electronic Colloquium on Computational Complexity (ECCC)*, TR26(TR26-082), 2026. URL: https://eccc.weizmann.ac.il/report/2026/082/.

[DPT24] Dean Doron, Edward Pyne, and Roei Tell. Opening up the distinguisher: A hardness to randomness approach for BPL = L that uses properties of BPL. In *ACM Symposium on Theory of Computing (STOC)*, pages 2039–2049, 2024.

[DPTW25] Dean Doron, Edward Pyne, Roei Tell, and Ryan Williams. When connectivity is hard, random walks are easy with non-determinism. In *ACM Symposium on Theory of Computing (STOC)*, 2025.

[ES73] Paul Erdős and John L. Selfridge. On a combinatorial game. *Journal of Combinatorial Theory, Series A*, 14(3):298–301, 1973.

[FSUV12] Bill Fefferman, Ronen Shaltiel, Christopher Umans, and Emanuele Viola. On beating the hybrid argument. In *Innovations in Theoretical Computer Science Conference (ITCS)*, pages 468–483. ACM, 2012. doi:10.1145/2090236.2090273.

[GG11] Eran Gat and Shafi Goldwasser. Probabilistic search algorithms with unique answers and their cryptographic applications. Technical Report TR11-136, Electronic Colloquium on Computational Complexity, 2011. URL: https://eccc.weizmann.ac.il/report/2011/136/.

[GW00] Oded Goldreich and Avi Wigderson. On pseudorandomness with respect to deterministic observers. In *ICALP Satellite Workshops*, pages 77–84, 2000.

[HAM02] William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences (J.CSS)*, 65(4):695–716, 2002. doi:10.1016/S0022-0000(02)00025-9.

[HN18] Sergiu Hart and Noam Nisan. The query complexity of correlated equilibria. *Games and Economic Behavior*, 108:401–410, 2018.

[IW97] Russell Impagliazzo and Avi Wigderson. $\mathsf{P} = \mathsf{BPP}$ if E requires exponential circuits: derandomizing the XOR lemma. In *ACM Symposium on Theory of Computing (STOC)*, pages 220–229, 1997.

[KMPS25] Michal Koucký, Ian Mertz, Edward Pyne, and Sasha Sami. Collapsing catalytic classes. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 455–463, 2025. doi:10.1109/FOCS63196.2025.00025.

[Kor22] Oliver Korten. Derandomization from time-space tradeoffs. In *Computational Complexity Conference (CCC)*, pages 37:1–37:26. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022.

[Kou16] Michal Koucký. Catalytic computation. *Bulletin of the EATCS (B.EATCS)*, 118, 2016.

[Li21] Jason Li. Deterministic mincut in almost-linear time. In *ACM Symposium on Theory of Computing (STOC)*, pages 384–395. ACM, 2021. doi:10.1145/3406325.3451114.

[LPT24] Jiatu Li, Edward Pyne, and Roei Tell. Distinguishing, predicting, and certifying: On the long reach of partial notions of pseudorandomness. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1500–1512, 2024. doi:10.1109/FOCS61266.2024.00095.

[Mer23] Ian Mertz. Reusing space: Techniques and open problems. *Bulletin of the EATCS (B.EATCS)*, 141:57–106, 2023.

[MPR26] Cassandra Marcussen, Edward Pyne, and Ronitt Rubinfeld. Graph spectral sparsification is in catalytic logspace. *Computing Research Repository (CoRR)*, abs/2608.21594, 2026. URL: https://arxiv.org/abs/2608.21594.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. doi:10.1017/CBO9780511814075.

[NW94] Noam Nisan and Avi Wigderson. Hardness vs. randomness. *Journal of Computer and System Sciences (J.CSS)*, 49(2):149–167, 1994.

[Ope26] OpenAI. Exact Derandomization of Logarithmic Space: L=RL=BPL, September 2026. URL: https://github.com/openai/math/blob/main/preprints/Exact-Derandomization-of-Logarithmic-Space-L-equals-RL-equals-BPL-September-23-2026/paper.pdf.

[PT26a] Edward Pyne and Roei Tell. Composing low-space algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2026.

[PT26b] Edward Pyne and Roei Tell. Using hardness vs randomness to design low-space algorithms. *Bulletin of the EATCS (B.EATCS)*, 148, 2026.

[Pyn24] Edward Pyne. Derandomizing logspace with a small shared hard drive. In *IEEE Conference on Computational Complexity (CCC)*, volume 300 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:20, 2024.

[Rag88] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences (J.CSS)*, 37(2):130–143, 1988. doi:10.1016/0022-0000(88)90003-7.

[Spe87] Joel Spencer. *Ten Lectures on the Probabilistic Method*. SIAM, 1987.

[SV22] Ronen Shaltiel and Emanuele Viola. On Hardness Assumptions Needed for "Extreme High-End" PRGs and Fast Derandomization. In *Innovations in Theoretical Computer Science Conference (ITCS)*, volume 215 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 116:1–116:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITCS.2022.116.

[Wil25] Ryan Williams. Simulating time in square-root space. In *ACM Symposium on Theory of Computing (STOC)*, 2025.

[WX08] Avi Wigderson and David Xiao. Derandomizing the Ahlswede-Winter matrix-valued Chernoff bound using pessimistic estimators, and applications. *Theory of Computing*, 4(3):53–76, 2008. doi:10.4086/toc.2008.v004a003.

[Yao82] Andrew C. Yao. Theory and application of trapdoor functions. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

A Online Hypercube Graph Balancing Problem

In this section, we prove the following lemma:

Lemma 5.2. *There exists a deterministic algorithm for the online hypercube graph balancing problem with in-degree bound f, where $f(h) \geq \exp(\Omega(\sqrt{h}))$.*

Consider the following simple greedy algorithm: At each step, upon receiving a query edge $\{u, v\}$, the algorithm orients $\{u, v\}$ toward the endpoint which has lower in-degree in the directed graph induced by the edges queried so far, with ties broken arbitrarily.

For the rest of this section, fix an edge query sequence $(e_1, \ldots, e_t)$. Let G be the directed graph over vertex set $\{0,1\}^n$ and edge set $\{e_1, \ldots, e_t\}$, with directions as chosen by the greedy algorithm.

The following observation shows that any vertex in G with high in-degree has several in-neighbours, each of which has reasonably high in-degree.

Observation A.1. *Let G be as fixed above. For every vertex $v \in V(G)$ and integer $j \in \mathbb{Z}_{\geq 0}$ such that $0 \leq j < \text{in-degree}_G(v)$, v has at least $\text{in-degree}_G(v) - j$ distinct in-neighbours in G, each of which has in-degree $\geq j$ in G.*

Proof. Consider the last $\text{in-degree}_G(v) - j$ edges that were oriented towards v by the greedy algorithm. In the step when each of these edges was queried, the in-degree of v must have already been at least j. The greedy algorithm always orients an edge towards the vertex with lower in-degree. Thus, the fact that the algorithm chose to orient these edges towards v implies that the in-degree of the other endpoint of each of these edges must have been at least j. These are the in-neighbours of v with in-degree at least j. This completes the proof. □

Lemma A.2. *Let G and t be as fixed above. For every vertex $v \in V(G)$,*

$$t \geq \frac{1}{2e^2} \exp\left(\sqrt{\frac{2 \cdot \text{in-degree}_G(v)}{3}}\right).$$

Proof. Let $h \coloneqq \text{in-degree}_G(v)$. When $h \leq 1$, the lemma statement follows trivially from the fact that $t \geq 1$. For the rest of this proof assume that $h \geq 2$. We define

$$b \coloneqq \left\lfloor \sqrt{\frac{2h}{3}} \right\rfloor \qquad \text{and} \qquad R_d \coloneqq h - db - \binom{d}{2} \quad (\text{for } 0 \leq d \leq b).$$

Since $h \geq 2$, we have $b \geq 1$ and

$$R_b = h - \frac{3b^2 - b}{2} \geq \frac{b}{2} > 0.$$

For a vertex $u \in V(G)$, we define its *level* $\in \{0, \ldots, n\}$ to be its Hamming distance from v. Furthermore, we call a vertex u at level $d \in \{0, \ldots, b\}$ *good* if $\text{in-degree}_G(u) \geq R_d$. Observe that the vertex v is good at level zero.

Let $0 \leq d < b$, and $u \in V(G)$ be a good vertex at level d. Due to Observation A.1 with $j \coloneqq R_{d+1}$, u must have at least $R_d - R_{d+1} = b + d$ in-neighbours u' with $\text{in-degree}_G(u') \geq R_{d+1}$. Since the Hamming distance of u from v is exactly d, at most d of these vertices can be at level $d - 1$. The remaining $\geq b$ of these in-neighbours must be at level $d + 1$.

Consequently, there are at least b^b paths in G, all of which end at the vertex v, have length exactly b, decrease their Hamming distance from v at each step, and visit only good vertices. Any

vertex of G can be the source of at most $b!$ such paths. We conclude that there are at least $b^b/b!$ distinct good vertices at level b. Therefore,

$$t = \sum_{v \in V(G)} \text{in-degree}_G(v) \geq R_b \frac{b^b}{b!}.$$

The following upper bound on $\ln(b!)$

$$\ln(b!) \leq \int_1^b \ln x \, dx + \ln b = b \ln b - b + 1 + \ln b$$

implies that

$$t \geq R_b \frac{e^{b \ln b}}{e^{b \ln b - b + 1 + \ln b}} = R_b \frac{e^{b-1}}{b}.$$

Together with $R_b \geq b/2$, this implies that

$$t \geq \frac{b}{2} \cdot \frac{e^{b-1}}{b} = \frac{1}{2} e^{b-1} \geq \frac{1}{2e^2} \exp\left(\sqrt{\frac{2h}{3}}\right).$$

This completes the proof. □

Lemma A.2 shows that the greedy algorithm has in-degree bound

$$f(h) \coloneqq \frac{1}{2e^2} \exp\left(\sqrt{\frac{2h}{3}}\right).$$

Clearly, $f(h) = \exp(\Omega(\sqrt{h}))$. This proves Lemma 5.2.

B Online Prefix-Tree Balancing Problem

In this section, we prove the following lemma:

Lemma 5.6. *There exists a deterministic algorithm for the online prefix-tree balancing problem with token bound $f(h) = h$.*

Consider the following simple greedy algorithm: At step i, let $B_i = \{v_1, \ldots, v_k\}$ be the queried batch with $|v_1| \geq |v_2| \geq \cdots \geq |v_k|$. The algorithm iterates j from 1 to k. The algorithm places the token of v_j on the child whose subtree contains fewer tokens placed **during the current batch**, with ties broken arbitrarily.

For the rest of this section, fix a query sequence $(B_1, \ldots, B_t)$.

Observation B.1. *Let T_n, $(B_1, \ldots, B_t)$, and the greedy algorithm be as above. Let $v \in \{0,1\}^n$ be a leaf vertex of T_n, and B_i be a batch in which the greedy algorithm places h tokens on the path from the root $\emptyset$ to the leaf v. Then $|B_i| \geq 2^h - 1$.*

Proof. We prove this via induction on h. First, for $h \leq 1$, the statement follows from the fact that $|B_i| \geq h$. We will now prove this for $h > 1$. Assume the inductive hypothesis that this statement is true for all $h' < h$.

Let $B_i = \{v_1, \ldots, v_k\}$ with $|v_1| \geq |v_2| \geq \cdots \geq |v_k|$. Let $v_j \in B_i$ be the hth vertex whose token the greedy algorithm places on the path from the root to v. Let's say that this token is placed on vertex $v_j \circ b$. Let B_i' denote the vertices in $\{v_1, \ldots, v_{j-1}\}$ which are contained in the subtree of

$v_j \circ b$. Observe that vertices outside B'_i do not affect the placement of the tokens of vertices in B'_i. Exactly $h-1$ tokens from B'_i are placed by the algorithm on the path from the root to v. Thus, by induction, we have $|B'_i| \geq 2^{h-1} - 1$.

Now, the greedy algorithm always places the token on the subtree with *fewer* tokens already placed on it. The fact that the algorithm chose to place the token on $v_j \circ b$ instead of $v_j \circ (\neg b)$ implies that the subtree of $v_j \circ (\neg b)$ already had at least $2^{h-1} - 1$ tokens placed on it. All of these tokens must come from queries in $\{v_1, \ldots, v_{j-1}\} \setminus B'_i$. Thus, we have

$$|\{v_1, \ldots, v_{j-1}\}| \geq |\{v_1, \ldots, v_{j-1}\} \setminus B'_i| + |B'_i| \geq 2^h - 2.$$

Therefore,

$$|B_i| \geq |\{v_1, \ldots, v_j\}| = |\{v_1, \ldots, v_{j-1}\}| + 1 \geq 2^h - 1.$$

This completes the proof. □

Proof of Lemma 5.6. Simply observe that for any $v \in \{0,1\}^n$, there must exist a batch B_i in which the algorithm places $h \geq \mathsf{load}(v)/t$ many tokens on the path from the root to the leaf v. Observation B.1 implies that

$$|B_i| \geq 2^h - 1.$$

This implies that

$$t\lfloor \log(|B_i| + 1) \rfloor \geq ht \geq \mathsf{load}(v).$$

Since for every vertex v, there exists a batch B_i for which this equation holds, taking the max over all B_i and over all vertices v proves the lemma. □

C A Uniform Version of UNPRED

Let L be a fixed language. We define a version of UNPRED known as UNPRED^L as follows.

Definition C.1 (UNPRED^L)**.** UNPRED^L is the search problem defined as follows:

1. **Input:** $n, m \in \mathbb{N}$, $0 < \epsilon \leq 1$, strings $x_1, \ldots, x_m$, and $\mathsf{ind}_1, \ldots, \mathsf{ind}_m \in [n]$.

   For all $i \in [m]$, $P_i : \{0,1\}^{n-1} \to \{0,1\}$ is a middle-bit predictor over $\{0,1\}^n$ at index $\mathsf{ind}_i \in [n]$. The value of a predictor P_i on a string $s \in \{0,1\}^{n-1}$ is defined as $P_i(s) \coloneqq L(x_i, s)$.

2. **Output:** A non-empty multiset $S \subseteq \{0,1\}^n$ such that

   $$\mathsf{adv}_{U(S)}(\{P_1, \ldots, P_m\}) < \epsilon.$$

In this definition, we think of the strings $x_1, \ldots, x_m$ as advice. If M is a Turing machine which decides the language L, then in order to obtain the value of a predictor P_i for $i \in [m]$ on a string s, one would evaluate M on s with advice x_i. If we fix L to be the Boolean circuit evaluation problem, and interpret the advice strings $x_1, \ldots, x_m$ as circuit descriptions, then we recover the original definition of UNPRED.

Now, we define a model of computation which is similar to inplaceFL, but allows for read-only access to an advice string.

Definition C.2 (Restorable Partial In-Place Computation)**.** A restorable partial in-place Turing machine M consists of two procedures: $\mathsf{Compute}_M$ and $\mathsf{Restore}_M$, which share a read-only input

tape, a read-write input tape, and a read-write work tape. M computes $R \subseteq \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^*$ if it satisfies the following two properties when executed on a read-only input x and read-write input y: 1) $\mathsf{Compute}_M(x,y)$ halts with $z \in \{0,1\}^*$ such that $(x,y,z) \in R$ written on the first $|z|$ bits of the read-write input tape; and 2) $\mathsf{Restore}_M$ starts from the read-write input tape and work tape configuration that $\mathsf{Compute}_M(x,y)$ halted with, and halts with y written on the read-write input tape.

We do not prove the following facts here, but we have done so in a previous draft, and we would be happy to share that draft with anyone interested.

Theorem C.3. *For any language L such that L can be computed by a restorable partial in-place Turing machine with logarithmic work space, we have*

$$\mathsf{UNPRED}^L \in \mathsf{SearchCL}.$$

Let M be such a machine. This follows directly from Theorem 3.3, by replacing each oracle query with an explicit evaluation of M. We use the part of the input tape of the catalytic machine which has x_i written on it as the read-only input tape of M, and use the catalytic tape which has the query string s written on it as the read-write input tape of M. We use fresh bits on the work tape of the catalytic machine as the work tape for M. For each query, we run $\mathsf{Compute}_M(x_i, s)$, copy the output bit of M to the work tape, and then run $\mathsf{Restore}_M$ to restore the catalytic tape.

Finally, the same argument as in Section 4, with more tedious proofs, shows the following theorem.

Theorem C.4. *There exists a language L that can be computed by a polynomial-time restorable partial in-place Turing machine with logarithmic work space such that*

$$\mathsf{SearchCL} \leq_{\mathsf{P}} \mathsf{UNPRED}^L.$$

These theorems together close the gap between $\mathsf{UNPRED}^{\mathsf{TC}^1}$ and UNPRED which exists in our simplified presentation.

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il