Electronic Colloquium on Computational Complexity, Report No. 32 (2024)

Explicit Time and Space Efficient Encoders Exist Only With Random Access

Joshua Cook[*] Dana Moshkovitz[†]

February 21, 2024

Abstract

We give the first *explicit* constant rate, constant relative distance, linear codes with an encoder that runs in time $n^{1+o(1)}$ and space $polylog(n)$ provided random access to the message. Prior to this work, the only such codes were *non*-explicit, for instance repeat accumulate codes [DJM98] and the codes described in [Gá+13]. To construct our codes, we also give explicit, efficiently invertible, lossless condensers with constant entropy gap and polylogarithmic seed length.

In contrast to encoders with random access to the message, we show that encoders with sequential access to the message can not run in almost linear time and polylogarithmic space. Our notion of sequential access is much stronger than streaming access.

[*]jac22855@utexas.edu. Department of Computer Science, UT Austin. This material is based upon work supported by the National Science Foundation under grant number 2200956.

[†]danama@cs.utexas.edu. Department of Computer Science, UT Austin. This material is based upon work supported by the National Science Foundation under grant number 2200956.

ISSN 1433-8092

Contents

1 Introduction

In this paper, we study the time and space efficiency of encoders for error correcting codes. An error correcting code is a function that maps any two distinct messages to codewords that are very far apart in hamming distance. Error correcting codes [Ree00; Ham50] have numerous practical and theoretical applications. Because of these applications, both codes with efficient encoders and lower bounds for encoders are useful. Codes can also be used as hard functions in lower bounds.

The efficiency of encoding codes has been extensively studied. Some notions of encoding complexity are the size and depth a circuit requires to encode the code. Spielman gave explicit codes that could be encoded by linear sized circuits with logarithmic depth and fan in 2 [Spi95]. For unbounded fan in circuits, Gál, Hansen, Koucký, Pudlák, and Viola gave tight bounds on the size and depth required to encode codes [Gá+13]. For depth 2 parity circuits, Gál et al. showed that the minimum circuit size required to encode a constant relative distance code was $\Theta(n(\log(n)/\log(\log(n)))^2)$.

In this paper, the notion of encoding complexity we focus on is the time and space required to encode. We investigate the time and space with two different models of how the message is accessed. One is the RAM model where any bit of the message can be accessed in one time step. The other is a sequential model, where the message can only be accessed through heads that can only move one space at a time.

Efficient Encoders with Random Access Branching programs are the nonuniform version of RAM algorithms. Many problems, such as sorting and finding unique elements, have time space lower bounds for branching programs [BC82; Yes84; Bea89; Abr91; BK23].

There are time and space lower bounds for encoding codes too. Bazzi and Mitter proved that for any code with constant relative distance, any branching programs encoding them in linear time require linear space [BM05], but they don't give any lower bounds on space if the encoder time is $\Omega(n\log(n))$. Self dual codes require branching programs that run in time T and space S to have $ST = \Omega(n^2)$ [SV06]. Some of the best lower bounds for nondeterministic branching programs are for recognzing codewords. For any good enough code, L, we have that any nondeterministic branching program recognising L in linear time requires linear space [Juk09].

Many codes can be computed either in almost linear time, or polylogarithmic space, but not both simultaneously. For example, Reed-Solomon codes have encoders that run in almost linear time with almost linear space, and encoders that run in polynomial time and polylogarithmic space, but Reed-Solomon codes do not have known encoders that run in almost linear time and polylogarithmic space. In fact, every constant rate linear code has both a quadratic time, log space branching program and a linear time, linear space branching program. However, not all linear codes have branching programs that are polylogarithmic space and almost linear time.

Repeat-Accumulate (RA) codes [DJM98] and the depth 2 circuits of [Gá+13] are both *non*-explicit codes that have branching programs that run in quasilinear time and logarithmic space. The best explicit RA codes only have relative distance $O(\frac{\log(N)}{N})$ [GM08]. The authors of [Gá+13] could only partially derandomize their construction. Prior to this work, no *explicit* codes were known that can be encoded simultaneously in almost linear time and polylogarithmic space.

Efficient Encoders with Sequential Access. Branching programs are a model with *random* access to the input. But some hardware accesses data sequentially, and some algorithms output data sequentially. Our algorithms with sequential access to the message will have a restricted number of heads, h, to access the message with. These heads can only be moved one space per time step, or jumped to the location of any other head. Even though the algorithm only has sequential access to the input, it has random access to its smaller working space.

Sequential access arises naturally when composing bounded space algorithms. The standard way to run one algorithm, A, on the output of another algorithm, B, space efficiently is by running A until it wants an input bit, then running B until it outputs that bit. This is a very sequential way to access the output of B. The obvious way to make this faster without storing the whole output of B is to keep intermediate states of B and only simulate B starting from the most recent intermediate state. This is like storing multiple heads to the output of B. Copying one intermediate state over another is like jumping one head to another.

We emphasize that sequential access is much stronger than streaming access as the program can choose which head moves and can read the same message many times. It is also stronger than standard Turing Machine access as there are multiple heads and there are head to head jumps. Other researchers have also studied models of computation with head to head jumps [SV77; Kos79; PSS80].

1.1 Main Results

We give the first explicit code with constant relative distance, constant rate and an almost linear time, polylogartihmic space RAM algorithm encoder.

Theorem 1.1 (Explicit Almost Linear Time, Polylog Space Encodable Codes)**.** *For any $\epsilon > 0$, and N, there exists a linear code*

$$C : \{0,1\}^N \to \Sigma^M$$

that has relative distance $1 - \epsilon$, output length $M = O(N)$ and alphabet $\Sigma = \{0,1\}^{poly(1/\epsilon)}$. Further C is computable in time $N\, poly(2^{\log(\log(N))^3}/\epsilon) + 2^{poly(1/\epsilon)}$ and space $O(\log(N)^2 + \log(N)\, poly(1/\epsilon))$ with random access to the message.

For constant ϵ, we have constant alphabet size, $\Sigma = \{0,1\}^{O(1)}$, and further C is computable in time $N^{1+o(1)}$ and space $O(\log(N)^2)$.

While one might want an actually linear time, log space encoder, Bazzi and Miiter [BM05] established that any linear time encoder requires linear space. So if one requires polylogarithmic space encoders, they can't run in linear time.

The distance of a linear code is the Hamming weight of its smallest non-zero codeword. So the idea of our code is to take any non-zero message and concentrate it's weight. To do this, we use a kind of function called a lossless condenser, so we call our codes "condenser codes".

A lossless condenser is a function that takes an input source with entropy and a uniform seed and outputs a source that is smaller than the input source, but still has the same entropy as the input source plus the seed. The idea of condenser codes is to treat the input bits that are one as a source of entropy. If this source can be losslessly condensed to an output with constant entropy gap, then this can be used to give a linear function whose output has constant hamming weight. See Section 2.3 for details.

Expander graphs have been used in constructing codes before, for instance for distance amplification [Alo+92] or Spielmann codes [Spi95]. But both of these construct codes iteratively in a way that makes it difficult to encode both time and space efficiently. In contrast, condensers give something closer to a one shot construction that makes it easier to encode time and space efficiently. Expanders have also been used to define codes using local constraints, such as with LDPC codes [Gal62; MN96; SS96], or even the three c codes [Din+22], but these codes don't have known efficient encoders [DDS23].

Next we give a lower bound for codes where the encoder only has sequential access to the message.

Theorem 1.2 (Lower Bounds For Encoders With Sequential Access)**.** *Suppose C is a code with relative distance δ encoding N bits. Suppose A is an algorithm computing C running in time T space S and using h sequential heads to access the message. Further assume $S > h\log(N)$. Then*

$$hST = \Omega(\delta N^2).$$

Our lower bounds on the encoders with sequential access to the message are much stronger than those by Bazzi and Mitter [BM05] on encoders with random access to the message. We give lower bounds on the space of an encoder with sequential access to the message, as long as it runs in time $\ll N^2$. Since in the branching model of computation there are codes with time $O(N\log(N))$ and space $\log(N)$ encoders, this shows that random access to the message is important for time and space efficiently computing a code.

In the case where we are trying to encode the output of a low space algorithm in a constant relative distance code, we can set $h = O(S)$. This gives a lower bound of $S^2T = \Omega(N^2)$. Since this sequential model of computation arises naturally when composing low space algorithms, this gives a technical barrier to time and space efficiently encoding the output of low space algorithms in codes. For example, it rules out certain natural approaches to constructing complexity preserving PCPs [BS+13].

Finally we show that our lower bound for encoding codes using sequential access to the message, Theorem 1.2, is tight up to low order factors by giving an explicit code that nearly achieves the lower bound. This code is based on a tensor code using the code in Theorem 1.1.

Theorem 1.3 (Encoders With Sequential Access Meeting the Lower Bounds). *For any number of heads $h \geq 2$, time $T \geq N$, space $S = \Omega(h \log(N))$, relative distance $\delta > 0$ with $hST = \Omega(\delta N^2)$, there exists a code with constant rate and relative distance $\Omega(\delta)$ encoded by a time $TN^{o(1)}$, space $S\,polylog(N)$ algorithm using h sequential heads to access the message.*

Remark (Uniformity of Results For Sequential Access). *We note that our lower bounds on sequential access hold for arbitrary operations on the working memory. That is, Theorem 1.2 even holds for nonuniform algorithms. But our encoders matching the lower bounds are uniform and only needs random access to working memory. That is, Theorem 1.3, provides a uniform algorithm.*

To construct the explicit codes for encoders with random access to the input, we use condensers. Our condensers need to be efficiently invertable. That is, given an output of the condenser, we need to efficiently iterate through all inputs that map to that output efficiently. To simplify this, we ask that our condenser output 2 outputs, the condensed output and a buffer so that the resulting function is invertible as a function. Additionally, our condenser needs to be good in several other ways. It needs to be lossless, have constant entropy gap, and have polylogarithmic seed size. See Section 2.2.1 for an informal definition of lossless condensers or Section 5 for a formal definition.

Theorem 1.4 (Good Invertible Condensers Exist). *For every n, k and ϵ such that $\epsilon > 2^{3-n/\log^*(n)^{\log^*(n)}}$, there is a time $poly(n)$, space $O(n^2)$ invertible, lossless $(n,k) \to_\epsilon (m, k+d)$ condenser*

$$C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n+d-m}$$

with $d = O(\log(n/\epsilon)^3)$ and $m = k + d + O(\log(1/\epsilon))$.

The efficiency of inverting lossless condensers is not commonly studied. However, efficiently invertible extractors have been used in wiretap protocols [CDS12] and non-malleable extractors [CG17; CGL16; Li17]. Lossless condensers and extractors are closely related: they are both special cases of condensers.

Some prior lossless condensers are invertible, such as multiplicity code based condensers [KTS22], but they don't have constant entropy gap and small seed. Prior constant entropy loss extractors [TSUZ01; GUV07] can be used to build lossless condensers with small seed and small entropy gap, but are not known to be invertible.

1.2 Time and Space Efficient Decoding?!

Other common notions of a code's complexity are the time required to encode and the time required to decode. Spielman codes [Spi95] not only have linear time encoders, but also have linear time decoders. But both the encoder and decoder require linear space.

Repeat accumulate codes, the codes of [Gá+13], and ours all make non-adaptive, also known as input oblivious, queries. That is, on any message, they always query the same message bits in the same order at the same time. While one can time and space efficiently deterministically *encode* codes with non-adaptive queries, one can not efficiently *decode* with deterministic, non-adaptive decoders, even with non-uniform branching programs. Gronemeier [Gro06] proved that any decoders which are deterministic and non-adaptive which run in time $O(N^{1+\alpha})$ must use space $\Omega(N^{1-\alpha})$. Thus one can not hope for decoders to be as efficient as encoders without being adaptive, or randomized.

There are randomized decoders that run in $N^{o(1)}$ space and almost linear time with random access to the message. Specifically, good locally decodable codes have this property [KSY14; Kop+16]. However, these codes have no known time and space efficient encoders.

1.3 On Sequential Access to The Input

The model of computation with sequential access to the input is less studied than branching programs. So we will formally define sequential access and briefly discuss some of its properties.

Definition 1.5 (Sequential Oracle). *Let x be some specific string of length n and h be an integer. Then a sequential oracle to x with h heads is a machine that has as state h integers within the range $[n]$. It has an input tape that can take up to two integers, $u_1, u_2 \in [h]$, indicating up to two of the h heads, and an operation which can be one of the following:*

1. *Move forward. This increments the u_1th head by one, if it is less than n.*

2. *Move backward. This decrements the u_1th head by one, if it is more than* 1.

3. *Read. Let i be the value of the u_1th head. Then this returns x_i.*

4. *Jump to. This sets the u_1th head to be the same value of the u_2th head.*

Any one of these operations can be done in one time step.

We call the oracle a non-reversible, sequential oracle if it can not use the 'move backward' operation.

We call the oracle a non-jumping, sequential oracle if it can not use the 'jump to' operation.

We say that an algorithm has sequential access to an input if the only way that input can be accessed is through a sequential oracle.

We assume that all sequential oracles start with all heads at position 1.

There are two potentially contentious operations of this sequential head model: the 'move backward' operation, and the 'jump to' operation. For simulating algorithms to access their output, the move backward operation may not be possible if the algorithm is not invertible. For hardware, often moving backward may be fine, but jumping heads is not. Our lower bounds, Theorem 1.2, holds even if heads can both jump and move backward. Our upper bounds, Theorem 1.3, needs heads that can jump, but not ones that move backward.[1]

The "jump to" operation is very powerful. Even with only head to head jumps, non-reversible heads can simulate reversible heads with only logarithmic overhead. Thus the resources needed when given reversible, sequential access is within a log factor of what is needed for non-reversible, sequential access to the input.

Lemma 1.6 (Reversibility Can Be Efficiently Simulated With Jumping)**.** *A single sequential head to a length N input can be simulated with $O(\log(N))$ non-reversible sequential heads to the same input with an expected time of $O(\log(N))$ for each head movement, and $O(\log(N))$ space.*

More generally, k sequential heads to a length N input can be simulated with $O(k\log(N))$ non-reversible sequential heads to that same input with an expected time of $O(\log(N))$ for each head movement, and $O(k\log(N))$ space.

Our sequential access to the input with reversible heads seems very similar to a Turing machine, but the addition of multiple heads makes it much more powerful. For instance, the classic example of a hard problem for a 1 tape Turing machine, the palindrome, takes quadratic time on a 1 tape Turing machine [Hen65]. Just two heads make palindromes easy to solve in linear time. Even if one is only given $O(\log(N))$ non-reversible heads and $O(\log(N))$ space, palindrome can be solved in $O(N\log(N))$ time with non-reversible, sequential access to the input.

There has been research on multi-head Turing Machines. Savitch and Vitányi [SV77] studied multi-head Turing Machines with heads that can jump to the location of other heads, like our heads do. This model was compared to and contrasted with multi-tape turing machines in several works [SV77; Kos79; PSS80]. Prior to this work, the only time space lower bounds for sequential access are those implied by the lower bounds for branching programs.

For completeness, one might ask if non-reversible sequential access is stronger than non-jumping sequential access and if non-jumping sequential access is stronger than non-jumping, non-reversible access. We show that this is indeed the case in the time and space bounded setting. Informally, we prove that

$$\begin{aligned}&\text{Random Access}\\ >&\text{Sequential Access}\\ \simeq&\text{Non-Reversible Sequential Access}\\ >&\text{Non-Jumping Sequential Access}\\ >&\text{Non-Jumping, Non-reversible Sequential Access}\end{aligned}$$

[1]Our codes can also be encoded in the same time and space bound with non-jumping sequential heads *if* one allows a preprocessing step to move all the heads into an initial position before starting.

That random access is more powerful than sequential access is a direct consequence of our explicit codes in the RAM model, Theorem 1.1, and our code lower bounds for sequential access to the input, Theorem 1.2. The other two inequalities come from our code lower bounds for sequential access, Theorem 1.2, and our explicit codes for sequential access, Theorem 1.3. See Section 9.3 for details.

2 Technique

For our results, we use the convention that capital letters are an exponential factor larger than their lower case counterparts. For example, $N = 2^n$, $K = 2^k$ and $M = 2^m$. In particular, our codes with efficient encoders with random access to the input are constructed using condensers that act on the indexes of the bits in the code. So our codes are functions on N bits, and our condensers are functions on n bits.

We start with proving our lower and upper bounds for encoders with sequential access to the input. Then we show how to build explicit codes with efficient encoders using random access to the input.

2.1 Sequential Access To The Message

If one only has a bounded number of heads to access the message and they can only move one space (or jump to other heads) in one time step, can we still time and space efficiently encode any code? We show that no codes with good distance can be time and space efficiently computed with only sequential access to the message.

We show that given space S, time T, relative distance δ and max number of heads h such that $hST = o(\delta N^2)$ that any algorithm running in time T and space S limited to at most h sequential heads can not compute a code with relative distance δ. Further when $hST = \Omega(\delta N^2)$, we show that an algorithm with time close to T, space close to S, and h heads computes a code with distance close to δ.

We start by showing the lower bound for non-adaptive sequential access to the input as a warm up. Then we give the lower bound for even adaptive sequential access to the input. Finally we show how to use time and space efficient codes that use random access to their message in order to construct codes with encoders with sequential access to the input that match our lower bounds.

2.1.1 Non-Adaptive Lower Bound

The idea is to group the message bits into intervals larger than S. Then for any interval, we think of a two person game, where one player has access to an interval when a head is in it, and gets to communicate about the contents of that interval to a second player whenever all heads leave that interval. The main ideas are that:

1. When a head leaves an interval, it can only communicate S bits. Thus the second player gets very little information about that interval every time it is visited.
2. If every head is far from an interval, it takes a lot of time to move a head back to that interval. So for most intervals, the second player only gets information about that interval few times.
3. If there are few heads, most intervals must have all heads very far from them at any given time. So for most intervals, the second player has to write most output bits.

Thus for most intervals, if the second player does not get as much information about an interval as the interval contains, then for two different messages, the second player must get the same information for both messages. Then the second player must output the same codeword bits for both. If the second player also writes most of the codeword, then most of the codeword will be the same for those two messages. Thus these two messages will have close codewords, so the code will not have good distance.

So we want that most intervals both have most output bits written when no head was near them, and that most intervals only went from having a head in them to having all heads far away a small number of times. Thus all the bits written when all heads were far away from an interval must have the same output for two separate messages.

In particular, we set the number of bits in each interval to be around $I = \frac{\delta N}{8h}$ so that most intervals have at least one interval between them and any head at any point in time. Since it takes at least I time steps to move a head into an interval that is distance I from every head, the number of times an interval transitions from being far from any heads to having a head on it is at most $\frac{T}{I}$.

For non-adaptive encoders, the argument follows fairly directly. An interval must have been visited at least $\frac{I}{S}$ times for player 2 to know enough about that interval to write different things for every possible contents of that interval. So only $\frac{ST}{I^2}$ many intervals can be visited enough to have any distance when a head isn't near them. Only $\frac{3h}{\delta}$ of the intervals can have a head near it when more than δ fraction of output bits are written. Thus a constant fraction of intervals, $\frac{N}{I} - \frac{3h}{\delta} = \frac{5h}{\delta} = \frac{5N}{8I}$, has at least a δ fraction of bits written when no head is near them.

Thus to have distance δ, we need the number of intervals visited often enough, $\frac{ST}{I^2}$, to be at least the number of intervals that often don't have a head near them, $\frac{5N}{8I}$. Otherwise, for one of the rarely visited intervals, for two different messages, the encoder must write the same thing when all heads are far from that interval. Thus the code wouldn't have good distance. Thus

$$\begin{aligned}\frac{ST}{I^2} &\geq \frac{5N}{8I}\\ ST &\geq \frac{5}{8}NI\\ ST &\geq \frac{5}{8}N\frac{\delta N}{8h}\\ hST &\geq \frac{5}{64}\delta N^2\end{aligned}$$

2.1.2 Adaptive Lower Bound

Adaptive lower bounds are trickier since the queries can change for different messages. As an example, our prior lower bound even works for messages where the entire message is zero, except for one interval. A non-adaptive algorithm can not even encode this extremely restricted message into codewords with large distance. But a non-adaptive algorithm that knows we will use this kind of counterexample will search for such an interval, then only encode that interval. This cuts down the number of bits the adaptive algorithm will have to encode by a factor of $\frac{\delta}{8h}$ which allows our adaptive encoder to outperform a non-adaptive encoder on these particular kinds of counterexamples.

The issue in this counterexample is that the encoder gets to know everything outside a target interval for free. So instead of always choosing the message outside an interval to be zero, our counterexample will fix it in some way that will depend on the encoder, so that way the algorithm can not know it ahead of time. So the idea essentially is to choose a random interval and a random restriction to everything outside the interval and then try to use a similar argument.

More specifically, we use a proof by contradiction. We say restriction of the message is good if it restricts everything but one interval, and for most assignments to variables in that interval, most bits are written when no head is near the interval, and heads don't enter the interval many times. If this holds for most assignments to the interval, the same prior arguments work. But if no good restrictions exist, it must be because on average the intervals are visited too many times for the number of heads and the time of the algorithm. But since we have a bound on the time and number of heads of the algorithm, this can not happen.

2.1.3 Upper Bound

The lower bounds from the previous section is tight since we can construct codes that match them, up to small factors. We construct these codes using a tensor code product of any code that has a space efficient, non-adaptive encoder using random access to the message and any other time efficient code. There is a straightforward way to time and space efficiently compute the tensor code of any space efficient code with another code. As a reminder, a tensor code arranges the message into a table, then one code encodes each row to construct an intermediate table. Then the other code encodes each column of the intermediate table to get the final result.

Specifically, for target space S, we arrange the code into S columns, each of size about $\frac{N}{S}$. Then each row is encoded in any arbitrary time efficient linear code. And each column is encoded in our time and space efficient code that uses random access to the message.

The time efficient way to encode the tensor code is to literally construct the intermediate table by encoding every row, then encode every column of the intermediate table. But storing this intermediate table is not space efficient. If the code encoding the columns is space efficient and non-adaptive, there is an alternate, space efficient way to do this. This is to simulate the encoder for all the columns in parallel, and every time it needs to query one of the rows, we encode this row and then give the result to the space efficient code.

So our encoder places each of the h heads uniformly across the rows of the message so that any row is within $O(\frac{N}{h})$ of a head. As long as there are fewer heads than rows, since the row encoder is time efficient, moving to the row will take more time than encoding it. Then every query to a row during the column encoding only takes time $O(\frac{N}{h})$. And since the column encoder is time efficient, we only need to query the rows around $\frac{N}{S}$ times. So the total time is around $T \simeq \frac{N^2}{hS}$, thus $hST \simeq N^2$. The space is similar to S since the column encoder is space efficient and only needs to store the state of S columns at once.

Finally, to get the tradeoff with distance, we use the same tensor code. Then we only bother to encode the message in size $\Omega(\delta N)$ intervals and encode each interval independently. If each of these smaller codes have constant relative distance, then these codes together have relative distance $\Omega(\delta)$. Using the prior time space efficient algorithms, for space S and heads h, you can compute a code with constant distance on a size (δN) message in time about $\frac{(\delta N)^2}{hS}$. Doing that $1/\delta$ times only requires time about $\frac{\delta N^2}{hS}$, which is what we want.

2.2 Random Access To The Message

First, we recall some relevant definitions. For any alphabet Σ with a special zero character, and any $x \in \Sigma^N$, we say the weight of x, denoted $weight(x)$, is the number of non-zero symbols in x. Similarly the relative weight of x is $\frac{weight(x)}{N}$. A linear code is any code whose encoding function is a linear function. Then the distance of a linear code is the weight of its smallest non-zero codeword. See Section 5 for more details.

So the goal is to construct some linear function which is fast and space efficient to compute, but has high weight for any non-zero message. Our basic strategy is to construct a different linear function for every possible (order of magnitude of) the message weight. Then we combine these linear functions into one linear function, such that if any linear function outputs a high weight output, the combined linear function will too.

This high level approach is nearly identical to the codes with depth 2 circuits of Gál, Hansen, Koucký, Pudlák, and Viola [Gá+13]. In fact, our code can also be viewed as an almost linear sized, depth 2 circuit. But our codes are explicit, and our results require several new ideas. For a detailed comparison, see Section 3.1.

2.2.1 Weight Fixers From Condensers

For any input range $R \subseteq [N]$ and relative weight $\delta > 0$, we call a function $F : \{0,1\}^N \to \Sigma^M$ an R to δ weight fixer if for every message x with weight within R, we have that $F(x)$ has relative weight at least δ. So our first goal is just to construct $[2^{i-1/2}, 2^{i+1/2}]$ to δ weight fixers for some $\delta > 0$ for every $i \in [\log(N)] = [n]$. For this we use lossless condensers.

One way of looking at lossless condensers is as bipartite graphs [RSW00; TSUZ01]. A lossless condenser is a regular graph with left vertices L, right vertices R, with left degree $D_L = 2^d$. We call d the seed length. We call the graph a lossless condenser for min-entropy k if for every $S \subseteq L$ with $|S| = 2^k$, we have that the neighbor set of S, denoted $N(S)$, has size $|N(S)| \geq |S|D_L(1-\epsilon)$. We say the condenser has constant entropy gap if $|N(S)| = \Omega(|R|)$, that is, a constant fraction of R has a neighbor in S.

If ϵ is small, then most of elements of $N(S)$ have one unique neighbor in S. Here, we view L as the input bits ($|L| = N$), and R as the output bits ($|R| = M$). If the message, x, has weight 2^k, we can let S be the one bits of the message. If the condenser has constant entropy gap, then a constant fraction of the output bits have a neighboring one bit. In fact a constant fraction of the output bits have exactly one neighbor that is a one bit.

Our weight fixer will just xor all an output bit's neighbors to compute its value. Then all of the output bits with a unique one bit neighbor will output one. Since for a weight 2^k message this is a constant fraction of the output bits, this weight fixer will give a constant relative weight output on a weight 2^k input. See Fig. 1 for an example of a condenser and its corresponding weight fixer.

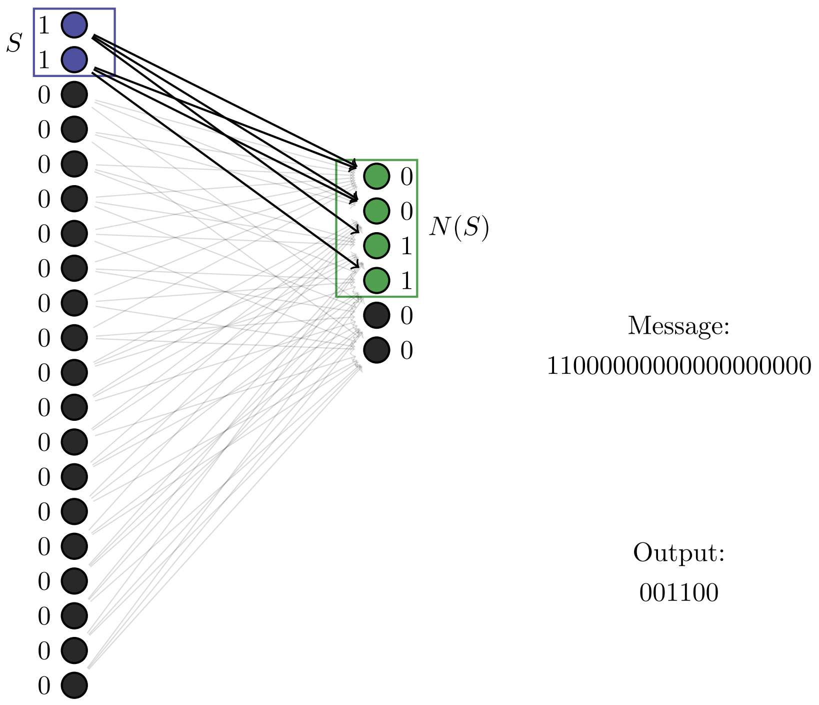

Figure 1: Lossless Condenser And $\{1,2\}$ to $\frac{1}{3}$ Weight Fixer Example

The lossless condensers we construct gives us, for any i, a $[2^{i-1/2}, 2^{i+1/2}]$ to δ weight fixer $F_i : \{0,1\}^N \to \{0,1\}^M$ where $M = 2^i 2^{O(\log(\log(N))^3)}$. This $2^{O(\log(\log(N))^3}$ factor is related to the seed length of the condenser. Each bit in the output of F_i can be computed in time about $N/2^i$ and space $O(\log(N)^2)$, so the total time to compute the entire output of an F_i is $N2^{O(\log(\log(N))^3)} = N^{1+o(1)}$. See Section 2.3 for more details.

2.2.2 Mixing Weight Fixers

Now we can perform a tensor like operation to mix all these weight fixers into one code. The idea is that a code always maps non-zero inputs into a string with high weight. So if we can partition the output of our weight fixers into small clusters so that most clusters are non zero, applying the code to each cluster gives a large weight output.

For any input weight message, some weight fixer F_i will have large weight. So all we need to do is make sure each cluster contains a good sample of the output of each F_i. This is easy, just have each cluster contain a single output symbol from each F_i in any way as long as each F_i has each of its outputs in the same number of clusters. To do this time and space efficiently, we just use the first symbol from F_i as many times as we need to, than the second symbol and so on.

We can visualize this mixing technique with a table. First put the output of each F_i into a row in a table. Repeat the symbols in each row until every row is the same length. Then encode the columns of that table with any asymptotically good code. The resulting weight will be at least the minimum weight of any row times the distance of the code. See Fig. 2 for a diagram.

To run this mixer time and space efficiently, we encode one column at a time, and store the current symbol for each of the $n = \log(N)$ weight fixers. Then to compute the next column, we only need to compute weight fixers who have changed their value since the last column. For most columns steps, most weight fixers won't use a new symbol because most weight fixers have their symbols repeated many times. This only requires $O(\log(N))$ space for the current output bit of each F_i, plus the space to compute any single F_i. It only requires the time to compute each F_i and to run C.

One issue with the approach we have described thus far is that the output length will be at least the output length of the largest F_i, which is $N2^{O(\log(\log(N))^3)}$. Thus the output length of the code we have just

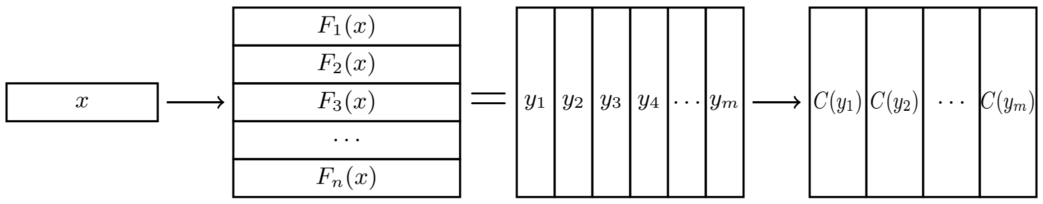
x
$F_1(x)$
$F_2(x)$
$F_3(x)$
$\cdots$
$F_n(x)$
y_1 y_2 y_3 y_4 $\cdots$ y_m
$C(y_1)$ $C(y_2)$ $\cdots$ $C(y_m)$

Figure 2: Mixing Weight Fixers $F_1, F_2, \ldots, F_n$ with code C

described is $N2^{O(\log(\log(N))^3)}$, but we need an output length of $O(N)$ to get constant rate. The weight fixers for larger message weights are the issue. So instead of using our condenser based weight fixers for every message weight, we only use them for small weight messages. We use a different weight fixer for messages with large weights and mix the two results.

2.2.3 Weight Fixers For Large Weight Messages

For large weight messages, we use a code extremely similar to Spielman codes [Spi95]. Spielman codes use a recursive approach, where every level of recursion is on a smaller input and increases the distance. We use the same codes, except that instead of recursing all the way down to a code on a constant sized message, we stop early with a weight fixer that is just a bit smaller than N. We call these Spielman style weight fixers.

Spielman style weight fixers always have constant rate, but the weight of messages they can fix increases with each level of recursion. One can compute the output of Spielman style weight fixers space efficiently, but the time increases exponentially in the number of recursions. Thus by choosing an appropriate recursion depth, we can balance the time used by the Spielman style weight fixers with the rate of the condenser style weight fixers.

The Spielman style weight fixer uses a lossless condenser to get a function, $A : \{0,1\}^N \to \{0,1\}^{N/2}$, such that for any input with weight less than some constant α, A gives an output with the same weight as the input. For the lossless condensers in the Spielman style weight fixers, we use the condensers from [Cap+02]. Then the idea is to repeatedly apply A in several levels until you have constant relative weight, then mix all of these levels.

If one stops the recursion early, we will not have had enough rounds to concentrate the small weight messages to be a constant fraction of the bits at any level. But each round will still let you fix weights a constant fraction smaller. The output bits at any given level are just some particular xor of message bits, but the number of bits you xor increases exponentially with the depth, which is why this algorithm has time exponential in its depth.

Now choosing appropriate number of recursions for the Spielman style weight fixers gives fixers for the very heavy messages. Light messages are handled by our weight fixers based on condensers. But this only gives us codes with *some* constant distance, and we would like codes that have arbitrarily large constant distance.

2.2.4 Distance Amplification

A first try would be to use an off the shelf distance amplification procedure, like that of Alon, Bruck, Naor, Naor and Roth (ABNNR)[Alo+92]. While we use the ideas of ABNNR, we can not use it directly as it does not preserve the space and time complexity of the encoder. First we explain ABNNR, and its limitations. Then we explain how to modify it to work for us.

The idea of ABNNR is to take a base code and a bipartite expander graph, and identify the left hand side with the message bits, and the right hand side with the output bits. Then the output of a right hand side vertex is just the concatenation of all its adjacent message bits. This increases the alphabet, and if the graph is an expander, increases the weight.

Now if we just apply ABNNR directly to our final code, it is unclear how to encode the resulting code efficiently. There is a straightforward, time efficient way to encode the code: just compute the whole left side code, than concatenate the symbols on the right hand side. But if one wants to do this space efficiently, then we do the following instead: for each right hand side vertex, we compute all the left hand side bits

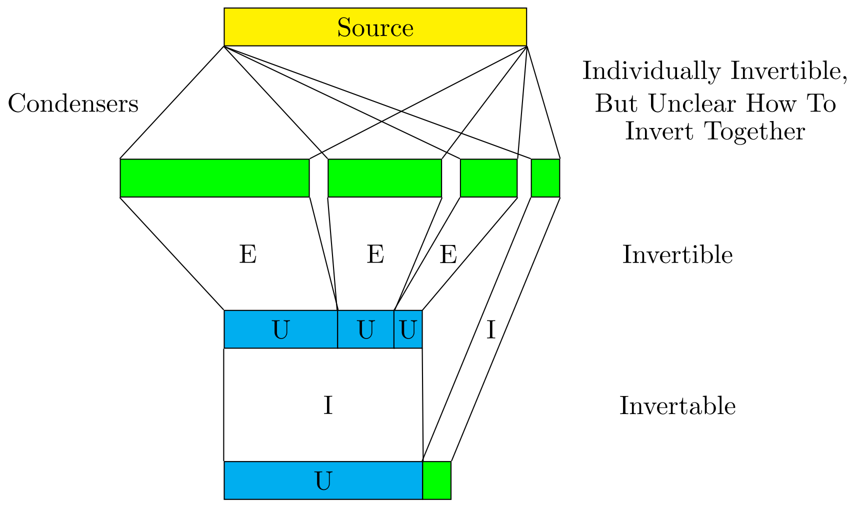

Figure 3: Standard Condense And Extract. Source is condensed from many times in parallel, and the results are extracted in parallel. Legend: E, Extractor. I, identity map. U, Uniform Bits.

incident to it and append them together. The issue is that each of the left hand vertices are a function of a constant fraction of the message bits, thus take time $O(n)$ to compute. Spending $O(n)$ time to compute each of the $O(n)$ output vertices takes quadratic time, which is too much.

Instead, we need to run distance amplification on each of the $O(\log(N))$ weight fixers before we combine them. Then when the bits of the weight fixer are expensive to compute, there are fewer output bits, so the distance amplification only increases encoder time by a constant factor. Then our mixer preserves this distance, as long as the code in the mixer also has good distance. This allows us to get arbitrary good weight.

2.3 Invertible Condensers

Now we discuss how to modify existing condenser and extractor constructions to get invertible condensers. To understand these condenser constructions, we need to explain an alternative, equivalent definition of condensers. Instead of thinking of condensers as bipartite graphs, one can also think of them as functions that take low entropy sources over long bit strings to sources with a similar entropy over short bit strings.

Explicitly, a k entropy lossless condenser is a function $C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ such that for any random variable $x \in \{0,1\}^n$ with min entropy k and $U_d \in \{0,1\}^d$ an independent, uniform distribution, we have that $C(x, U_d)$ is close to a distribution with min entropy $k + d$. We call $m - (k + d)$ the entropy gap since it is the difference between the amount of entropy that could be in an output with m bits and how much entropy is in that output. The entropy gap of the condenser is related to the weight of the related weight fixer. We want constant entropy gap.

The state of the art condensers based on Pavarash-Vardy codes and Multiplicity codes [GUV07; KTS22] are lossless, but require large seeds to get small entropy gaps. One can think of extractors as a kind of condenser with no entropy gap at all. We know explicit extractors with small seeds [Tre99]. Unfortunately extractors must have entropy loss, and we need lossless condensers.

One might hope for a way to combine lossless condensers and extractors to get a the benefits of both: small entropy loss for extractors, and small entropy gap for condensers. This is possible using the condense and extract framework [RSW00; TSUZ01]. Next we describe the condense and extract framework.

In the condense and extract framework, one first condenses the message to concentrate the entropy, then extracts much of it. Then there is still some remaining entropy in the message, conditioned on the output of the extractor, so we condense it again to a much smaller message, which we can then more efficiently extract from. Then we repeat until almost all the entropy has been extracted. Then to convert the final result to a lossless condenser, we condense the remaining entropy one final time without extracting it.

Things get a bit more complicated when one requires invertibility. First, we require our component condensers and extractors to be invertible, which is doable. But then we run into an issue of extracting

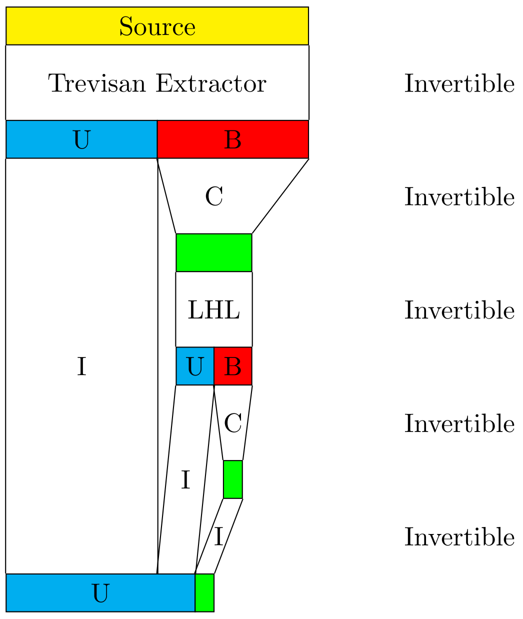

Figure 4: Our Condenser. It has two extractors, a Trevisan extractor and a Left-over Hash Lemma (LHL) based extractor. Both extractors output some uniform bits and a buffer with the left over entropy. Before the buffer can be extracted from efficiently, it needs to be condensed first. Legend: C, Condense. I, identity map. U, Uniform Bits. B, Buffer.

and condensing from the message many times in parallel. Now even if we can invert each of the condensers and extractors by themselves efficiently, it remains difficult to space and time efficiently determine which messages can give the expected output of each extractor and condenser simultaneously.

This is solved by changing our extractors into buffered extractors, like those of [Cap+02], and then condensing from that buffer. A buffered extractor is just an extractor with a second output, called a buffer, which contains all the entropy the first output missed. With this change, inversion is straightforward. For the same reason, the extractors and condensers of [Cap+02] are also efficiently invertible.

Now it only remains to choose appropriate invertible extractors and condensers to compose to get our final condenser. For condenser we choose the multiplicity code based condensers as these are both efficient and are easy to invert. For extractors, we use the Trevisan extractor [Tre99; RRV99] to extract most of the entropy, and then a left-over hash lemma based extractor to get the rest. So the final condenser runs the Trevisan extractor, condenses, runs the left-over hash lemma extractor, and then condenses again.

The Trevisan extractor is efficiently invertible because it is a linear function conditioned on the seed, thus can be inverted by Guassian elimination. This requires quadratic space and polynomial time (in $n = \log(N)$). One subtle issue is that we define invertibility of an extractor as a literal function inversion of an extractor along with a buffer. But the Trevisan extractor with a fixed seed is not always full rank, thus is not an invertible function. To handle this, our extractor detects such bad seeds (which don't extract well anyway) and just use any arbitrary invertible function with them.

This Trevisan extractor is the main limitation in our encoders time and space. Getting an extractor with shorter seed length will improve the time of the encoder, and a more space efficient inversion process would improve the space of the encoder.

3 Comparison With Other Codes

3.1 Comparisons With Codes For Shallow Circuits

While we investigate time and space efficiency of encoding, other works have investigated the circuit complexity of encoding codes. For instance, while Spielman codes [Spi95] are often cited as linear time encodable, they are also encodable by a uniform, fan-in 2, log depth circuit with linear size. A later work by Gál, Hansen, Koucký, Pudlák, and Viola [Gá+13] considered unbounded fan-in circuits with arbitrary gates. For any depth, Gál et al. gave tight bounds on the size of a circuit required to encode a code with constant relative distance.

Gál et al. show that encoding any good code with depth 2 circuits require $\Omega(n(\log(n)/\log(\log(n)))^2)$ wires and depth 3 circuits require $\Omega(n\log(\log(n)))$ wires. As depth increases further, the number of wires required sharply decreases, with linear sized circuits at depth $\log^*(n)$. We emphasize that [Gá+13] give both lower bounds and matching upper bounds. However, their codes are non-explicit.

Our code constructions and the depth 2 circuits of [Gá+13] are, conceptually, extremely similar. Their circuit constructions use what they call "range detectors" which are equivalent to weight fixers. Its depth 2 circuits do exactly what we do: make weight fixers for each order of magnitude, then mix them. Our encoders could also be stated as uniform, almost linear sized, depth 2 circuits.

Derandomization of the codes of [Gá+13] was left as an open problem. They could only achieve partial derandomization. Their partial derandomization is a variation of our mixer (compare Theorem 6.3 with [Gá+13, Claim 37]), but they had no explicit constructions for weight fixers. Our weight fixers can actually be expressed as parity gates, and our encoders can be described as explicit depth 2 circuits of almost linear size. Thus we solve the open problem in [Gá+13] of finding explicit codes with depth 2 circuits of almost linear size.

Gál et al. never analyzed the time and space required to encode their codes. It is not obvious that the codes of [Gá+13] should be encodable in almost linear time and logarithmic space. While it is true that any constant depth, almost linear sized circuit can be evaluated in either almost linear time and almost linear space, or logarithmic space and polynomial time, they can't always be computed in almost linear time and logarithmic space simultaneously. The fully randomized codes with depth 2 encoders from [Gá+13] do not seem to have almost linear time and log space encoders, only their partially derandomized codes do.

The main differences between their construction and ours come from the fact that ours are explicit. We use condensers to make our weight fixers explicit, and the best known condensers cannot make weight fixers that are as good as the randomized construction of Gál et al. So we need several new ideas to get our codes. A few difficulties and solutions include:

1. For weight K inputs, our lossless condensers give weight fixers with output length $\Omega(K2^{poly(\log(\log(N)))})$. When K is close to N, this is larger than N. If we used these weight fixers for large weight messages, the output would have super linear size, so our code would not be constant rate. So we need to construct weight fixers for large input weight messages in a different way (through Spielman style weight fixers).

   The weight fixers in [Gá+13] for weight K inputs have output length $O(\log\binom{N}{K})$. So their weight fixers always have less than linear output length.

2. Condensers cannot give us distance $1-\epsilon$ for small constant ϵ. To improve our distance, we need to use the distance amplification of ABNNR [Alo+92], but in a special way. ABNNR does not seem to simultaneously preserve time and space efficiency of encoding: it can do one or the other. So we have to apply it to our weight fixers *before* they are mixed.

   Gál et al. only gives codes with relative distance δ for some constant $\delta > 0$, it does not try to give large relative distance. However, if their randomized construction is modified to give weight fixers with multiple output bits, then the same approach gives codes with distance $1-\epsilon$, codeword length $O(\frac{N}{\epsilon^2})$, alphabet $\{0,1\}^{O(\log(1/\epsilon))}$ and encoders running in time $N\,poly(\log(N)/\epsilon)$ and space $O(\log(N)\log(1/\epsilon))$.

Thus while the high level code construction of [Gá+13] is similar to ours, we need several new ideas to make it explicit, keep the rate constant, and the distance close to one.

3.2 Why Spielman Codes Aren't Enough

Spielman codes [Spi95] are well known codes that can be encoded in linear time, so it is natural to ask whether they can also be encoded in small space. From Bazzi and Mitter [BM05], we know that it's linear time encoder cannot be sublinear space. We know of an alternate way to encode Spielman codes in logarithmic space, but this approach requires time $n^{1+\beta}$ for some $\beta > 0$. Standard Spielman codes have $\beta > 1.5$, but even optimizing their parameters one cannot get β approaching zero without the relative distance of the code also approaching zero. Subsequent improvements, like those of Guruswami and Indyk [GI05], still contain Spielman codes within them, and thus suffer from the same problems with encoding efficiency.

Now we give a brief explanation of this space efficient evaluation of Spielman codes, and why it can't be time efficient. One can view Spielman codes [Spi95] as a parity circuit. Each layer in the parity circuit is given by some family of regular bipartite expanders A_i for $i \in [O(\log(N))]$. For simplicity, you can think of the circuit as identifying the gates of layer i of the circuit with the left vertices of A_i, and the gates in layer $i+1$ of the circuit with the right vertices of A. Then the edges in A denote the inputs to the parity gates.

The obvious space efficient way to evaluate such a circuit is to recursively evaluate a gate's value by iterating over each input to that gate. For a depth depth circuit with fan in f with final layer size L, this only takes space $O(\mathsf{depth}\log(f))$, but requires time $f^{\mathsf{depth}} \cdot L$. Then fan-in, f, is the degree of the expanders, and depth, depth, decreases with the expansion of the expanders. Improving one hurts the other. One can improve the tradeoff by making the expanders more imbalanced, but that hurts the distance of Spielman codes. Our one shot approach allows us to use very imbalanced expanders without hurting our distance.

4 Open Problems

There are many open problems related to time and space efficiency of encoding.

1. Get better condensers to construct codes with more efficient encoders using random access to the input. Our codes need, for every $k \in [n]$, a lossless $(n, k) \to_\alpha (m, k+d)$ condenser, $C_k : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^{m_k}$, with constant entropy gap ($m_k = k + d + b$ for $b = O(1)$) and approximation error, α, less than one half. Our codes have messages of length $N = 2^n$.

   - Improve the space of our encoder.

     If each C_k is invertible in space S, then our code can be encoded in space space $S + O(n)$. Our condenser is constructed with a Trevisan style extractor [Tre99; RRV99], which we only know how to invert by exploiting its linearity, which takes space n^2. If one uses a more efficiently invertible condenser, that will improve the space required by the encoder. We suspect that other condenser designs, like [GUV07], could be made space $O(n)$ and $poly(n)$ time invertible, but have not checked.

   - Improve the time of our encoder.

     If each C_k is invertible in time T and has seed length d, then the time of our encoder is $O(N\log(N)2^d T)$. So if one can get efficiently invertible, lossless condensers which condense all (except $O_\alpha(1)$) bits of entropy with seed length $O(\log(n))$, then there is a code with an encoder that runs in time $N\,polylog(N)$.

     We note that if one can give an extractor with seed length $O(\log(n))$ which extracts all the entropy (except $O_\alpha(1)$ bits), then we have a lossless condenser with a similar seed length and an encoder that runs in time $N\,polylog(N)$. The best known explicit extractors [GUV07] require seed length $O(\log(n)^2)$ to extract all (but $O_\alpha(1)$ bits) of the entropy of a source. This seed length is not short enough to get a quasilinear time, polylog space encoder.

   - Improve the dependence on ϵ.

     A simple version of our codes only achieve constant relative distance. To get relative distance $1-\epsilon$ for any constant ϵ, we use extractors. For the entropy gap $b = O(1)$, for every k, we need an $(m_k - b, \epsilon)$ extractor $E : \{0,1\}^{m_k} \times \{0,1\}^{d'} \to \{0,1\}^{m'_k}$ to get a code with distance $1-\epsilon$. If E is time T and space S invertible, then this distance amplification increases the encoding time by a factor of $O(2^{d'}T)$ and increases space by $O(S)$.

     For our distance amplification, we use the extractors from [Cap+02]. These use some small, non explicit conductors of size $poly(1/\epsilon)$. Since we only find these using brute force, this adds an extra time of $2^{poly(1/\epsilon)}$ and an extra space of $poly(1/\epsilon)$. Using a different extractor that does not require a brute force search could give a better dependence on ϵ.

2. Give codes with deterministic decoders that run in small time and space, or show they can not exist. It is already known that any deterministic decoder running in small time and space must be adaptive

[Gro06], but it remains open if these adaptive decoders exist. If not, this would be an interesting case where randomized algorithms are provably more powerful than deterministic ones, since locally decodable codes have efficient randomized decoders.

3. Give a code with an encoder *and* randomized decoder that run in small time and space with random access to the input, or show they can not exist. We now know that there exist codes with a time and space efficient encoder, and codes with a time and space efficient, randomized decoder. Can a single code be both time and space efficient to encode and decode?

4. There are other interesting properties of codes, like

   - Local Testability.
   - Local Decodability.
   - Relaxed Local Decodability.

   In particular, let X be the interesting property (such as local test ability or local decodability), then for the different kinds of access, we ask

   (a) In the sequential access setting, is there a code with X and an encoder running in time T and space S with $h = O(S)$ sequential heads to access the input such that $ST \ll N^2$?

   We proved in Theorem 1.3 that we can do better than this with sequential access to the message for some codes. In particular, if $S = h = \sqrt{N}$, then there is a code that can be encoded in almost linear time. So if a code cannot be encoded time and space efficiently, it must be because of X not just because we require constant relative distance.

   (b) In the random access setting, is there a code with X that can be encoded in almost linear time and polylogarithmic space?

   We know codes with time and space efficient encoders can't have some interesting properties. For example, self-dual codes require encoders running in time T and space S to have $ST = \Omega(N^2)$ [SV06].

5. Derandomize the Repeat Accumulate codes (RA codes).

   Repeat accumulate codes have a simple description: first, take an input, x, and repeat it k times (think of $k = O(\log(N))$) to get y. Then, for some fixed random permutation π (this is why the code is not explicit), permute y by π to get z. Finally, for every $i \in [kN]$, the ith output bit is the xor of bits in z before index i: if the resulting codeword is C, than $C_i = \bigoplus_{j \leq i} z_j$.

   Non-explicit RA codes have faster encoders than condenser codes and are simpler to describe. RA codes run in time $O(N \log(N))$ and use space $O(\log(N))$. Even with optimal condensers, condenser codes *cannot* be made to run in $O(N \log(N))$ time. This is because condenser codes can also be described as depth 2 circuits, and Gal et al [Gá+13] proved depth 2 circuits encoding a code require size $\Omega(N \log(N)^{1.999})$. So there may be a simpler, more efficient, explicit code based on RA codes.

   The best known derandomization of RA codes [GM08] only have distance $O(\log(N))$, i.e. relative distance $O(\frac{\log(N)}{N})$. This low distance is inherent to the technique: the distance is the girth of a three regular graph, and all three regular graphs have girth $O(\log(N))$.

5 Preliminaries

In this paper, it will be convenient to use linear codes as it has a simple way to characterize its distance. But we also want larger alphabet sizes as achieving high distance is easier with larger alphabet. So we will define all of our functions as if they are over a binary alphabet, but for distance we will use a larger alphabet. This doesn't change any of the actual codes, but simplifies some of the analysis.

A code is just a function whose outputs differ in most locations. Here is a formal definition of an error correcting code.

Definition 5.1 (Code, Distance, and Rate)**.** *Let Σ_1 and Σ_2 be any alphabets over binary bits: $\Sigma_1 = \{0,1\}^a$ and $\Sigma_2 = \{0,1\}^b$ for some integers a and b. Then for any function $C : \Sigma_1^N \to \Sigma_2^M$, we say C is a code with relative distance δ if for any two $x, y \in \Sigma_1^N$ we have that $C(x)$ and $C(y)$ differ on at least δ fraction of indexes.*

We say that C has rate $\frac{N}{M}$. We say that an element $u \in \Sigma_2^M$ is a codeword of C if for some $x \in \Sigma_1^N$ we have that $C(x) = u$.

All of our codes will be linear, so we need to define a linear function.

Definition 5.2 (Linear Function)**.** *For any function $L : \{0,1\}^N \to \{0,1\}^M$, we say L is linear if every output bit of L is just a parity of some specific set of input bits.*

Let Σ_1 and Σ_2 be any alphabets over binary bits: $\Sigma_1 = \{0,1\}^a$ and $\Sigma_2 = \{0,1\}^b$ for some integers a and b. Then we say that any function $L' : \Sigma_1^N \to \Sigma_2^M$ is linear if L' viewed as a function on individual bits is linear.

Linear codes, that is codes who are themselves linear functions, have many nice properties. One nice property is that the distance of the code is equal to the weight of its smallest, nonzero output. The weight of a vector is just its number of nonzero elements. Here is a formal definition of the weight of a string.

Definition 5.3 (Weight of a String)**.** *Let Σ be any alphabet over binary bits: $\Sigma = \{0,1\}^a$ for some integer a. Then for any $x \in \Sigma^N$ for some integer a, we define weight(a) to be the number of symbols in x that are not the all 0 symbol: 0^a.*

The relative weight of x is $\frac{weight(x)}{N}$.

Now we can show a useful characterization of the distance of linear function.

Lemma 5.4 (Distance of a Linear Code)**.** *Let Σ_1 and Σ_2 be any alphabets over binary bits: $\Sigma_1 = \{0,1\}^a$ and $\Sigma_2 = \{0,1\}^b$ for some integers a and b. Let $C : \Sigma_1^N \to \Sigma_2^M$ be a linear function. Then C is a code whose relative distance, δ, is the weight of its smallest non-zero output:*

$$\delta = \min_{x \in \Sigma_1^N, (0^a)^N \neq x} \frac{weight(C(x))}{M}.$$

Proof. See that for any two elements $u, v \in \Sigma_2^M$, for any index $i \in [m]$ we only have $(u-v)_i = 0^b$ if u and v are equal on index i. Thus the distance between two outputs of C is just the weight of their difference. Thus the distance of the C can be written as

$$\delta = \min_{x,y \in \Sigma_1^N, x \neq y} \frac{weight(C(x) - C(y))}{M}.$$

But since C is linear, we can just simplify $C(x) - C(y)$ as $C(x-y)$. Thus the distance between $C(x)$ and $C(y)$ is just the weight of $C(x-y)$. So by letting $z = x - z$, we can write

$$\delta = \min_{z \in \Sigma_1^N, z \neq (0^a)^N} \frac{weight(C(z))}{M}.$$

□

To construct our explicit codes, we need to use pseudorandom objects called extractors and condensers. The goal of these objects is to take inputs with some randomness and a lot of correlations, and give a shorter output with a similar amount of randomness. Extractors want the shorter output to look almost uniform, but often are so short they lose some randomness. Condensers want the shorter output to contain almost all the randomness, but may not be short enough to be close to uniform.

To formally define extractors and condensers, we need to define min entropy: H_∞. Intuitively, the min entropy is the number of bits of information one always gets from a single output of a distribution. This is in contrast to the standard notion of entropy, which is like the average amount of information a single output gives. Min entropy is a more convenient notion of entropy for us.

Definition 5.5 (Min Entropy)**.** *For any distribution, X, over any alphabet, Σ, we define the min entropy, H_∞, of X as*

$$H_\infty(X) = \max_{\sigma\in\Sigma} - \log(\Pr[X=\sigma]).$$

If X is a uniform distribution over $K = 2^k$ different elements, we call X a flat k source, and $H_\infty(X) = k = \log(K)$.

Now we often don't actually have a low min entropy source. Often there may be one or two inputs that have a large chance of appearing, and we still need to work with these. So we relax our requirements on distributions to not necessarily be high min entropy themselves, but to be close to something with high min entropy. So let us define the distance of two distributions.

Definition 5.6 (Statistical Distance)**.** *For any distributions, X, Y over some alphabet Σ, the distance of X to Y is*

$$\Delta(X,Y) = \frac{1}{2}\sum_{\sigma\in\Sigma} || \Pr[X=\sigma] - \Pr[Y=\sigma]||.$$

If $\Delta(X,Y) \le \epsilon$, we say X is an ϵ approximation of Y or that X is ϵ close to Y.

Distances in this sense compose neatly with functions in the sense that if X is an ϵ approximation of Y, then for any function f, we have that $f(X)$ is also an ϵ approximation of Y.

One subtlety of extractors and condensers is that there is no general function that is able to take any input with high entropy and be able to give a smaller output without losing just as much entropy. To do this, we need some extra structure on the input entropy. Here, we provide that structure by giving our extractors and condensers a second, very small, uniform input called a "seed". In this work, we assume all condensers and extractors are seeded.

Definition 5.7 (Seeded Extractor Definition)**.** *Let $E : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ be any function. Let U_d be the uniform distribution over $\{0,1\}^d$.*

We say E is a (k,ϵ) extractor if for any distribution, X, over $\{0,1\}^n$, with min entropy k, we have that $E(X,U_d)$ is ϵ close to the uniform distribution over $\{0,1\}^m$.

Condensers are defined similarly, except that we don't enforce the output to be close to uniform, but just some high entropy distribution. If the distribution the output is close to has all the entropy of the original distribution plus the entropy of the seed, we call the condenser lossless since it didn't lose any of the input entropy.

Definition 5.8 (Seeded Condenser Definition)**.** *Let $C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ be any function. Let U_d be the uniform distribution over $\{0,1\}^d$.*

We say C is a $(n,k) \to_\epsilon (m,k')$ condenser if for any distribution, X, over $\{0,1\}^n$ with min entropy k, we have that $C(X,U_d)$ is ϵ close to some distribution over $\{0,1\}^m$ with min entropy k'.

If $k' = k + d$, we say that C is a lossless condenser.

To prove our lower bounds for sequential access to the input, we will use a tool called random restrictions. The idea of a random restriction is to fix some of the inputs to a function, and not others.

Definition 5.9 (Restriction)**.** *A restriction of length n is just a string $x^* \in \{0,1,*\}^n$. Any index where x^* is either 0 or 1 is fixed, and any index where x^* is not fixed, we say it is free.*

If x^* is a restriction with k free indexes, and there is a binary string $y \in \{0,1\}^k$, then we define the string $x_y = y \circ x^*$ to be the string that agrees with x^* on indexes where x^* is fixed, and on the jth index that x^* is free agrees with y_j.

6 From Condensers To Codes

In this section, we will show how to construct codes using condensers. We will later show how to construct the condensers we need.

Our codes are constructed by mixing weight fixers, so we will start by defining weight fixers.

6.1 Weight Fixers

A weight fixer is a linear function that outputs strings with a constant relative weight, as long as the weight of the message is within a specified range.

Definition 6.1 (Weight Fixer)**.** *For any two alphabets, Σ_1 and Σ_2, any subset $R \subseteq [N]$ and constant $\delta \in (0, \frac{1}{2})$, a linear function $F : \Sigma_1^N \to \Sigma_2^M$ is said to be an R to δ weight fixer if, given any x with weight$(x) \in R$ then weight$(F(x)) \geq \delta M$.*

We say that R is the input weight range, δ is the relative output weight, N is the input length, and M is the output length.

A straightforward corollary of this definition is that any $[N]$ to δ weight fixer is a code with relative distance δ, since the distance of a linear code is the weight of its smallest non zero codeword. Thus our strategy will be to combine several weight fixers that cover different parts of the weight range into a single weight fixer that covers the entire weight range.

For large weight messages, we can perform a repeated weight amplification type procedure to get the appropriate weight. The exact way we perform weight amplification is similar to Spielman's codes and described in Section 8.

Theorem 6.2 (Weight Fixer For Heavy messages)**.** *Take any integer K, and any integer N such that $N = 2^i K$ for some i. Then, for some constant $\alpha > 0$, there is a $[K/2^i, N]$ to $\alpha/4$ weight fixer $F : \{0,1\}^N \to \{0,1\}^{4N}$. Further, for some constant, c, any bit in the output of F can be computed in time c^i polylog(N) and space $O(i + \log(N))$.*

Now that we have defined weight fixers, we will show how to efficiently mix them to get better weight fixers, and eventually codes.

6.2 Weight Fixer Mixer

Now we show how to combine many weight fixers with different input ranges, and combine them to get a weight fixer whose input range is their union. The idea is to arrange the output of each weight fixer as rows in a table, where each weight fixer is repeated until they all have the same length, then encode the columns with any arbitrary code.

While this is a geometrically easy way to think about the combination, and what we do in the following theorem, we note it is not optimal if the weight fixers have very different outputs. In particular for our weight fixers. Once can improve the result by splitting very large weight fixers into multiple rows, so that all the small weight fixers don't need to be copied many, many times. This is what is done in [Gá+13, Claim 37], but for simplicity, we do not do it here.

Theorem 6.3 (Fixer Mixer)**.** *Suppose for some ℓ, for each $i \in [\ell]$ there is an R_i to δ weight fixer $F_i : \Sigma_1^N \to \Sigma_2^{M_i}$. Let $C : \Sigma_2^\ell \to \Sigma_3^c$ be any linear code with relative distance δ' computable in time T' and space S'.*

Suppose for some R we have $R \subseteq \bigcup_{i \in [\ell]} R_i$ and for some length M for each $i \in [\ell]$ we have $M_i | M$. Then there is an R to $\delta\delta'$ weight fixer $F : \Sigma_1^N \to \Sigma_3^{cM}$.

Further, if for each $i \in [\ell]$ any individual output element of F_i is computable in time T_i and space S_i, then the full output of F is computable in space

$$O(\ell \log(|\Sigma_2|) + \log(M)) + \max\{S', \max_{i \in [\ell]} S_i\}$$

and time

$$(T' + O(1))M + \sum_{i \in \ell} M_i T_i.$$

Alternatively, if for each $i \in [\ell]$ the full output of F_i is computable in time T_i and space S_i, then the full output of F is computable in space

$$O(\ell \log(|\Sigma_2|) + \log(M)) + S' + \sum_{i \in [\ell]} S_i$$

and time
$$(T' + O(1))M + \sum_{i \in \ell} T_i.$$

Proof. The idea of the code is simple. First, we lengthen the output of each weight fixer so that it has length M. Then we apply C element wise to the output of each weight fixer. More specifically, for $a \in [M]$ we define $y_a \in \Sigma_1^\ell$ to be the ath column in the table. That is, for $i \in [\ell]$ we have
$$(y_a)_i = F_i(x)_{\lceil aM_i/M \rceil}.$$
Then F just applies C to each column, y_a, and concatenates them. That is for any $a \in [M]$ and $b \in [c]$ column a row b of the output is just the bth output of $C(y_a)$. That is,
$$F(x)_{(a-1)*c+b} = C(y_a)_b.$$

Suppose x has weight $w \in R$. Then for some i, we know $w \in R_i$. Thus $F_i(x)$ has relative weight δ. Thus for at least δ fraction of a, we have $y_a \neq 0$. For each such a, by the distance of C, for δ' fraction of b, we have $C(y_a)_b \neq 0$. Therefore, for at least $\delta\delta'$ fraction of (a, b) pairs we have $F(x)_{(a-1)*c+b} \neq 0$. Thus F has relative weight at least $\delta\delta'$.

To encode F, we compute the code for each F_i in parallel and apply C in a straightforward way.

If the individual bits of each weight fixer is efficient to compute, then the space can be reused between the different weight fixers, keeping only the current bit of each F_i and some indexes in memory. This takes space $O(\log(M)+\ell\log(|\Sigma_2|))+\max\{S', \max_{i\in[\ell]} S_i\}$. Similarly the time is just the sum of the time to encode with C for M times, plus some book keeping, plus time to compute every bit of every weight fixer. This is time $(T' + O(1))M + \sum_{i\in\ell} M_i T_i$..

If the full output of each L_i is efficient to compute, the encoding algorithms is essentially the same, only now we cannot reuse space between the different weight fixers. This is because we need to pause each weight fixer after it outputs a bit and resume it when we need its next one. This requires space $O(\ell \log(|\Sigma_2|) + \log(M)) + S' + \sum_{i\in[\ell]} S_i$ and time $(T' + O(1))M + \sum_{i\in\ell} T_i$, noting that in this case T_i is the time to output all bits, not just a single one. $\square$

Now if we had weight fixers covering every input weight, then we could mix them to get codes. Now we show how to get weight fixers from lossless, invertible condensers.

6.3 Weight Fixers From Invertible Condensers

Most of our weight fixers will be constructed through condensers. One additional, non standard property our condensers will need is invertibility. This is because the final weight fixer will enumerate through all the message bits whose index map to an output bits index and xor them together to get that output bit. So it needs to be efficient to, given any output bit index, enumerate through all the message indexes that map to that output. We call a condenser invertible if this can be done efficiently.

To simplify our definitions and proofs slightly, we restrict ourselves to condensers that output an index function along with its condensed output such that the two together is an efficiently invertible function. That is, not only can we enumerate through the inputs that give an output, but given the index of the input that gives an output, compute that specific input efficiently.

Definition 6.4 (Invertible Condenser)**.** *Suppose $C' : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ is an $(n, k) \to_\epsilon (m, k')$ condenser. Suppose there is a function $I : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^{n+d-m}$, and define $C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n+d-m}$ by*
$$C(x, s) = (C'(x, s), I(x, s)).$$
Suppose C is a bijection with $C^{-1} : \{0,1\}^m \times \{0,1\}^{n+d-m} \to \{0,1\}^n \times \{0,1\}^d$ its inverse. Then we say C is an invertible $(n, k) \to_\epsilon (m, k)$ condenser.

We say C is time T and space S invertible if C^{-1} can be computed in time T and space S. We call C^{-1} the inverse of C, we call C' the condenser part of C, and I the index function of C. We still call d the seed length and m the output length.

To use condensers to create weight fixers, we need the following properties. To get a short output, we need small seed length. Since we have Theorem 6.2, to handle very heavy messages, seed length $polylog(n)$ suffices for our work. To get high weight, we need a lossless condenser with only constant entropy gap. That is, the condenser needs to output all the entropy and the number of bits in the output needs to be at most a constant number many more bits than the amount of entropy. Finally, they need to be invertible in polynomial time. Such good condensers exist.

Now we show that lossless condensers, when used as described above, give weight fixers.

Theorem 6.5 (Lossless Condensers give Weight Fixers)**.** *Suppose you have an invertible $(n,k) \to_\epsilon (k+d+b,k)$ lossless condenser*

$$C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^{k+d+b} \times \{0,1\}^{n-k-b}$$

with seed length d that is time T and space S invertible. Let $m = k+d+b$.

Then there is a $[2^{k-1/2}, 2^{k+1/2}]$ to $\left(\frac{1}{2} - 2\epsilon\right)2^{-b}$ weight fixer, $F : \{0,1\}^{2^n} \to \{0,1\}^{2^m}$, with input length $N = 2^n$ and output length $M = 2^m$ whose individual output bits can be computed in time $(T + O(1))2^{n-k-b}$ and space $S + O(d+b+n)$.

Proof. Our linear function F identifies every message bit with an n bit index, i, and every output bit with an m bit index, j. Then the output bit at index j is the parity of all message bits x_i where for some seed s we have $C(i,s)_1 = j$. More formally:

$$F(x)_j = \bigoplus_{i,s:C(i,s)_1=j} x_i = \bigoplus_{i\in\{0,1\}^{n-k-b}} x_{C^{-1}(j,i)}.$$

By inspection, one can see that the output length is $2^m = M$.

To compute $F(x)_j$, we simply have to invert C to find all 2^{n-k-b} message bits that map to j and xor the corresponding bits together. Since C is time T and space S invertible, this only takes time $T2^{n-k-b}$ to perform each inversion plus $O(2^{n-k-b})$ for book keeping. Similarly for space we just need to keep track of which bit we are outputting, which neighbor of that bit we are at, and the space for the inversion. This is space $O(m) + O(n-k-b) + S$.

To see that F is weight fixing, choose any x with weight $w \in [2^{k-1/2}, 2^{k+1/2}]$. Now we show that for a large fraction of the $j \in \{0,1\}^m$, there is only one index ℓ and seed s with $x_\ell \neq 0$ such that $C(\ell,s) = j$. This would imply that for such j that

$$F(x)_j = \bigoplus_{i,s:C(i,s)=j} x_i = x_\ell \neq 0.$$

If $w \geq 2^k$, then we will show that any specific set of 2^k ones of x approximately map to unique outputs and the extra ones can't cancel out too much.

Take any $X \subseteq \{0,1\}^n$ such that $|X| = 2^k$ and for all $i \in X : x_i = 1$. Then there must be at least $(1-\epsilon)2^{k+d}$ distinct indexes j such that for some index i and seed s we have $C(i,s) = j$. Otherwise, we have a k entropy flat source whose output in expectation over the seed differs from any $k+d$ source by more than ϵ. Then at most $\epsilon 2^{k+d}$ of the index i seed s pairs map to a j for a second time. Thus at least $(1-2\epsilon)2^{k+d}$ output indexes j have a unique index $i \in X$ seed s that maps to i and with $x_i = 1$. The rest of the at most $(\sqrt(2)-1)2^k$ ones in x and 2^d seeds can only hit $(\sqrt(2)-1)2^{k+d}$ of these.

So at least

$$(1 - 2\epsilon - \sqrt(2) + 1)2^{k+d} > (1/2 - 2\epsilon)2^{k+d}$$

of the output indexes j have a distinct i and s such that $C(i,s) = j$. Thus the output has weight at least $(1/2 - 2\epsilon)2^{k+d}$. This is relative weight $(1/2 - 2\epsilon)2^{-b}$.

If $w \leq 2^k$, then we will show that any specific super set of 2^k ones containing those of x approximately map to unique outputs and the missing ones can't can't be too many of these.

Take any $X \subseteq \{0,1\}^n$ such that $|X| = 2^k$ and for all $i \in X : x_i = 1 : x \in X$. Then, as in the last case, at least $(1-2\epsilon)2^{k+d}$ output indexes j have a unique $i \in X$ with x_i and seed s that maps to them. Now x is only missing $(1 - \frac{1}{\sqrt{n}})2^k$ of the ones in X. These missing indexes only contribute $(1 - \frac{1}{\sqrt{n}})2^{k+d}$ ones to these output pairs.

So at least

$$(1 - 2\epsilon - 1 + 1/\sqrt(2))2^{k+d} > (1/2 - 2\epsilon)2^{k+d}$$

of the output indexes j have a distinct i and s such that $C(i, s) = j$. Thus the output has weight at least $(1/2 - 2\epsilon)2^{k+d}$. This is relative weight $(1/2 - 2\epsilon)2^{-b}$. $\square$

Now since good invertible condensers exist, and good invertible condensers give good weight fixers, good weight fixers exist.

Lemma 6.6 (Weight Fixers For Small Weight Messages). *For some constant $\beta > 0$, for every $N = 2^n$ and $K = 2^k$, there is a $[2^{k-1/2}, 2^{k+1/2}]$ to β weight fixer $F : \{0,1\}^N \to \{0,1\}^M$ where $M = O(K2^{O(\log(\log(N))^3)})$. Further, any bit of F can be computed in time $O(polylog(N)\frac{N}{K})$ and space $O(\log(N)^2)$.*

Proof. This is a direct application of Theorem 6.5 to Theorem 1.4. So for $\epsilon = 1/10$, we have from Theorem 1.4 a time $poly(n)$, space $O(n^2)$ invertible, lossless $(n, k) \to_\epsilon (m, k + d)$ condenser

$$C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n+d-m}$$

with $d = O(\log(n/\epsilon)^3) = O(\log(n)^3)$ and $m = k + d + O(\log(1/\epsilon))$. Let $b = m - k - d = O(\log(1/\epsilon)) = O(1)$

Then from Theorem 6.5 there is a $[2^{k-1/2}, 2^{k+1/2}]$ to $\left(\frac{1}{2} - 2\epsilon\right)2^{-b}$ weight fixer, $F : \{0,1\}^{2^n} \to \{0,1\}^{2^m}$, with message length 2^n and output length $2^m = O(K2^{\log(n)^3})$ whose individual output bits can be computed in time $poly(n)2^{n-k-b} = O(polylog(N)\frac{N}{K})$ and space $O(n^2 + d + b + n) = O(\log(N)^2)$. See that the output weight $\left(\frac{1}{2} - 2\epsilon\right)2^{-b}$ is a positive constant since b is constant and $2\epsilon < \frac{1}{2}$. $\square$

Now if we assume we have condensers, we have weight fixers. But not for arbitrarily large constant weight. We handle that next.

6.4 Distance Amplification

So now we have weight fixers that are time and space efficient to compute for every order of magnitude, but these weight fixers only have some constant output weight. It could be very small. We want weight close to 1. So we apply a final weight fixer to them that takes constant relative weight inputs and amplifies them to outputs with weight close to 1.

We note that we have to do this amplification on the individual weight fixers before we combine them, rather than afterward. This is because our distance amplification weight fixer queries it's input in a random order, not sequentially. So the time to compute the amplified weight fixer is proportional to the length of it's output, and the cost to compute a random symbol of it's input. If applied after mixing all of the weight fixers, this time per input symbol will be close to N with close to N outputs, which will take N^2 time. But when applied to an individual weight fixer, it will only increase the time it takes to output a symbol by a constant factor.

Our weight fixer for very heavy messages uses an extractor to group message bits such that all but ϵ fraction of groups has a one in it. Then we just output all the bits in a group as a symbol in the alphabet. While we could output a code of all the bits in a group, we don't need to for our results and doing so would be unnecessarily complicated.

For this to work, we need to define an invertible extractor, similar to an invertible condenser.

Definition 6.7 (Invertible Extractor). *Suppose $E' : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ is (k, ϵ) extractor. Suppose there is a buffer function $B : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^{n+d-m}$, and define $E : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n-m}$ by*

$$E(x, s) = (E'(x, s), B(x, s)).$$

Suppose E is invertible with inverse $E^{-1} : \{0,1\}^m \times \{0,1\}^{n-m} \times \{0,1\}^d \to \{0,1\}^n$. Then we say E is an invertible extractor.

We say E is time T and space S invertible if E^{-1} can be computed in time T and space S. We call E^{-1} the inverse of E, we call E' the extractor part of E, and B the buffer, or index function of E.

Now we show that using our extractors as described before gives a weight fixer.

Theorem 6.8 (Extracters give Weight Fixers)**.** *Suppose you have an invertible (k, ϵ) extractor*

$$E : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n+d-m}$$

with seed length d that is time T and space S invertible.

Let $\Sigma = \{0,1\}^{2^{n+d-m}}$.

Then there is a $[2^k, 2^n]$ to $1-\epsilon$ weight fixer, $F : \{0,1\}^{2^n} \to \Sigma^{2^m}$, with message length 2^n and output length 2^m whose individual output bits can be computed in time $(T+O(1))2^{n+d-m}$ and space $S+O(n+d+2^{n+d-m})$ with just 2^{n+d-m} queries to the message.

Proof. This is the most crude form of ABNNR [Alo+92]. For every output bit $i \in \{0,1\}^m$, we let $F(x)_i$ be the concatenation of every bit x_j where for some $s \in \{0,1\}^d$ we have $E(j,s)_1 = i$. This can be easily computed by inverting E for 2^{n+d-m} many times.

For any message with at least 2^k ones, by the extractor property of E, must hit at all but ϵ fraction of outputs, otherwise the corresponding distribution could not be ϵ close to uniform. ☐

The following extractor is a specific instantiation of [Cap+02, Theorem 7.2], which is efficiently invertible. See the discussion in Section 7.1.

Lemma 6.9 (Invertible, Very High Entropy Extractors Exist)**.** *For every n, k, and $\epsilon > 0$, there exists a time $poly(n \log(1/\epsilon)))$ space $O(n) + poly(2^{n-k}/\epsilon)$ invertible (k, ϵ) extractor*

$$E : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^k \times \{0,1\}^{n-k+d}$$

with seed length $d = O(\log(n-k) + \log(1/\epsilon))$.

This extractor requires $2^{poly(2^{n-k}/\epsilon)}$ preprocessing time.

Proof. This just instantiates [Cap+02, Theorem 7.2] with $t = n - k$. ☐

This implies weight fixers for arbitrarily large output weights, starting from any constant weight. We will always use the following lemma where K is within a constant factor of N.

Corollary 6.10 (Weight Fixers with Large Output Weight)**.** *For every $N = 2^n, K = 2^k$, and $\epsilon > 0$, there exists a $[K, N]$ to $1-\epsilon$ weight fixer, $F : \{0,1\}^N \to \Sigma^K$, with message length N and output length K whose individual output bits can be computed in time $poly(\log(N)\frac{N}{\epsilon K})$ and space $O(\log(N)) + poly(\frac{N}{\epsilon K})$ using $poly(\frac{N}{\epsilon K})$ queries to the message.*

Here, $\Sigma = \{0,1\}^{poly(\frac{N}{K\epsilon})}$. There is also an additional $2^{poly(\frac{N}{\epsilon K})}$ preprocessing time.

Proof. First, we apply Lemma 6.9 to get a time $poly(n, \log(1/\epsilon))) = poly(\log(N)\log(1/\epsilon))$ space $O(n) + poly(2^{n-k}/\epsilon) = O(\log(N)) + poly(\frac{N}{\epsilon K})$ invertible (k, ϵ) extractor

$$E : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^k \times \{0,1\}^{n-k+d}$$

with seed length $d = O(\log(n-k) + \log(1/\epsilon))$.

See that

$$\begin{aligned} 2^{n+d-k} =& \frac{N}{K} poly(\frac{\log(N/K)}{\epsilon}) \\ =& poly(\frac{N}{K\epsilon}). \end{aligned}$$

Then we can apply Theorem 6.8 to get a $[N = 2^n, K = 2^k]$ to $1-\epsilon$ weight fixer, $F : \{0,1\}^{N=2^n} \to \Sigma^{K=2^k}$, with message length $N = 2^n$ and output length $K = 2^k$ whose individual output bits can be computed in time

$$poly(\log(N)\log(1/\epsilon))2^{n+d-k} = poly(\log(N)\frac{N}{\epsilon K})$$

and space

$$\begin{aligned}S + O(n + d + 2^{n+d-k}) =& S + O(\log(N/K\epsilon) + \frac{N}{K} poly(\frac{\log(N/K)}{\epsilon}))\\ =& S + poly(\frac{N}{K\epsilon}).\end{aligned}$$

with just $2^{n+d-k} = poly(\frac{N}{K\epsilon})$ queries to the message.

Here

$$\begin{aligned}\Sigma =& \{0,1\}^{2^{n+d-k}}\\ =& \{0,1\}^{poly(\frac{N}{K\epsilon})}.\end{aligned}$$

There is also an additional $2^{poly(\frac{N}{\epsilon K})}$ preprocessing time. □

Now we can apply this to both our Spielman style weight fixer and our condenser based weight fixer to get weight fixers for any order of magnitude message, and any constant weight output. This corollary just applies Corollary 6.10 to the output of Theorem 6.2.

Corollary 6.11 (Heavy Message, Very Heavy Output Fixers)**.** *Take any integer $K = 2^k$, and any integer $N = 2^n$ such that $N \geq K$. Then for any $\epsilon > 0$, there is a $[K, N]$ to $1 - \epsilon$ weight fixer $F : \{0,1\}^N \to \Sigma^M$ where $\Sigma = \{0,1\}^{poly(1/\epsilon)}$ and $M = O(N)$. Further any bit in the output of F can be computed in time $poly(\frac{N\log(N)}{K\epsilon})$ and space $O(\log(N) + poly(1/\epsilon))$.*

There is also an additional $2^{poly(\frac{1}{\epsilon})}$ preprocessing time.

In the same way, this corollary just applies Corollary 6.10 to the output of Lemma 6.6.

Corollary 6.12 (Small Message Weight, Large Output Weight Fixers)**.** *For any constant $\epsilon > 0$, for every $N = 2^n$ and $K = 2^k$, there is a $[2^{k-1/2}, 2^{k+1/2}]$ to $1 - \epsilon$ weight fixer $F : \{0,1\}^N \to \Sigma^M$ where $M = 2^m = O(K 2^{O(\log(\log(N))^3)})$ and $\Sigma = \{0,1\}^{poly(1/\epsilon)}$. Further, any bit of F can be computed in time $O(\frac{N}{K} poly(\frac{\log(N)}{\epsilon}))$ and space $O(\log(N)^2 + poly(1/\epsilon))$.*

There is also an additional $2^{poly(1/\epsilon)}$ preprocessing time.

6.5 Putting it all together

Now that we have weight fixers for every input range, and they are good, all that is left is to assemble our final code.

Now we can combine all the condenser based weight fixers to get a single weight fixer that works on all small messages. We recall that because of our long seed, we do not get linear length output for all weight ranges. Thus we only combine up to some message weight that is about $\frac{N}{2^d}$ where d is the seed length of the condenser. That is, we invoke the following theorem with $K \frac{N}{2^d}$.

We also don't mix the condenser based weight fixers with the Spielman style ones at this step either, since our weight fixer mixer adds a small overhead to the output length. As noted before, we can fix this by giving a better weight fixer mixer. But we instead mix our small weight fixers first, then mix our large weight fixer with the result.

Lemma 6.13 (Mixing Small Weight Fixers To Get One Weight Fixer)**.** *For any $\epsilon > 0$, for any $N = 2^n$ and $K = 2^k$, there is a $[K]$ to $1 - \epsilon$ weight fixer $F : \{0,1\}^N \to \Sigma^M$ with $M = O(K\, poly(1/\epsilon) 2^{O(\log(\log(N))^3)})$ and $\Sigma = \{0,1\}^{poly(1/\epsilon)}$. Further, the full output of F can be computed in space*

$$O(\log(N)^2 + \log(N)\, poly(1/\epsilon))$$

and time

$$N 2^{O(\log(\log(N))^3)} poly(1/\epsilon) + 2^{poly(1/\epsilon)}.$$

Proof. The proof works by invoking Corollary 6.12 for every $i \leq K$ and combining them with Theorem 6.3.

So specifically, for every $i \leq k$, Corollary 6.12 gives a $[2^{i-1/2}, 2^{i+1/2}]$ to $(1-\epsilon/2)$ weight fixer $F_i : \{0,1\}^N \to \Sigma_1^{M_i}$ where $M_i = O(2^i 2^{O(\log(\log(N))^3)})$. Further, any bit of F_i can be computed in time $O(\frac{N}{2^i}\, poly(\frac{\log(N)}{\epsilon}))$ and space $O(\log(N)^2 + poly(1/\epsilon))$. Here $\Sigma_1 = \{0,1\}^{poly(1/\epsilon)}$.

There is also an additional $2^{poly(1/\epsilon)}$ preprocessing time. This preprocessing is the same for each i.

Finally, to use Theorem 6.3, we need the existence of some efficient code, linear, code from $k\, poly(1/\epsilon)$ bits to $k\, poly(1/\epsilon)$ symbols of $O(poly(1/\epsilon))$ bits with distance $1-\epsilon/2$. Since this is a code on only $k\, poly(1/\epsilon)$ bits, we can afford to use a less efficient code. So we can for instance use a Spielman code [Spi96] with [Alo+92] (this is the same code as Corollary 6.11 with $K = 1$, evaluated in a more time, less space efficient way) to get such a code, call it $C : \Sigma_1^k \to \Sigma_2^{k'}$ where $\Sigma_2 = \{0,1\}^{poly(1/\epsilon)}$ and $k' = O(k\, poly(1/\epsilon))$.

Note that since each M_i is a power of 2, we can upper bound the least common multiple of the M_is by some $M' = K2^{O(\log(\log(N))^3)}$.

Now we can apply Theorem 6.3 to get a $[K]$ to $(1-\epsilon/2)^2 > (1-\epsilon)$ weight fixer $L : \{0,1\}^N \to \Sigma_2^{O(M'k')}$. Let

$$\begin{aligned} M =& M'k' \\ =& O(K \log(K)\, poly(1/\epsilon) 2^{O(\log(\log(N))^3)} \\ =& O(K\, poly(1/\epsilon)) 2^{O(\log(\log(N))^3)}. \end{aligned}$$

Further the full output of L is computable in space

$$O(k \log(|\Sigma_2|) + \log(M) + \log(N)^2 + poly(1/\epsilon)) = O(\log(N)\, poly(1/\epsilon) + \log(N)^2)$$

and time

$$O(M\, polylog(N) + \sum_{i \in k} 2^i 2^{O(\log(\log(N))^3)} \frac{N}{2^i}\, poly(\frac{\log(N)}{\epsilon})) = O(N 2^{O(\log(\log(N))^3)}\, poly(1/\epsilon)).$$

Adding in the $2^{poly(1/\epsilon)}$ preprocessing time, this gives a total time of

$$N 2^{O(\log(\log(N))^3)}\, poly(1/\epsilon) + 2^{poly(1/\epsilon)}.$$

□

Now that we have a weight fixer for light messages and a weight fixer for heavy messages, we can combine them to get a weight fixer for every message weight, which must be a code since weight fixers are linear. We now prove Theorem 1.1.

Theorem 1.1 (Explicit Almost Linear Time, Polylog Space Encodable Codes). *For any $\epsilon > 0$, and N, there exists a linear code*

$$C : \{0,1\}^N \to \Sigma^M$$

that has relative distance $1-\epsilon$, output length $M = O(N)$ and alphabet $\Sigma = \{0,1\}^{poly(1/\epsilon)}$. Further C is computable in time $N\, poly(2^{\log(\log(N))^3}/\epsilon) + 2^{poly(1/\epsilon)}$ and space $O(\log(N)^2 + \log(N)\, poly(1/\epsilon))$ with random access to the message.

For constant ϵ, we have constant alphabet size, $\Sigma = \{0,1\}^{O(1)}$, and further C is computable in time $N^{1+o(1)}$ and space $O(\log(N)^2)$.

Proof. The basic idea is to combine Lemma 6.13 with Corollary 6.11 setting

$$K = \frac{N}{\log(1/\epsilon) 2^{O(\log(\log(N))^3)}}.$$

This is the setting at which Lemma 6.13 stops having linear length, and Corollary 6.11 still has almost linear time. In particular, choose c to be the constant so that the output length of Lemma 6.13 is $M \leq cK(1/\epsilon)^c 2^{c \log(\log(N))^3}$, and set

$$K = \frac{N}{c(1/\epsilon)^c 2^{c \log(\log(N))^3}}$$

so that

$$M \leq cK(1/\epsilon)^c 2^{c\log(\log(N))^3}$$
$$= N.$$

Then by Lemma 6.13 there is a $[K]$ to $1-\epsilon$ weight fixer $F_1 : \{0,1\}^N \to \Sigma'^M$ with $\Sigma' = \{0,1\}^{poly(1/\epsilon)}$ and $M = O(K\,poly(1/\epsilon)2^{O(\log(\log(N))^3)})$. Further, the full output of F_1 can be computed in space

$$O(\log(N)^2 + \log(N)\,poly(1/\epsilon))$$

and time

$$N2^{O(\log(\log(N))^3)}\,poly(1/\epsilon) + 2^{poly(1/\epsilon)}.$$

Using the same K with Corollary 6.11 gives a $[K,N]$ to $1-\epsilon$ weight fixer $F_2 : \{0,1\}^N \to \Sigma'^{M'}$ where $\Sigma' = \{0,1\}^{O(poly(1/\epsilon))}$ and $M' = O(N)$. Further any bit in the output of F can be computed in time $poly(\frac{N\log(N)}{K\epsilon}) = poly(2^{\log(\log(N))^3}/\epsilon)$ and space $O(\log(N) + poly(1/\epsilon))$. So one can say the entire output of F_2 can be computed in time $O(N\,poly(2^{\log(\log(N))^3}/\epsilon))$ and space $O(\log(N) + poly(1/\epsilon))$.

There is also an additional $2^{poly(1/\epsilon)}$ preprocessing time.

Now for the code needed by Theorem 6.3, we use the trivial code that just outputs one symbol containing the entire message. This has distance 1, and has output alphabet $\Sigma = \Sigma'^2 = \{0,1\}^{poly(1/\epsilon)}$.

Then by Theorem 6.3 there is an $[N]$ to $1-\epsilon > 1$ weight fixer $L : \{0,1\}^N \to \Sigma^{2M'=M=O(N)}$. That is, L is a linear code with distance $1-\epsilon$.

Further, the full output of L is computable in space

$$O(\log(N)^2 + \log(N)\,poly(1/\epsilon))$$

and time

$$N\,poly(2^{\log(\log(N))^3}/\epsilon)) + 2^{poly(1/\epsilon)}.$$

□

7 Constructing Invertible Condensers

Our condensers need to have constant entropy gap, polylogarithmic seed length, and be efficiently invertible. All the condenser constructions we know of do not achieve all three.

The condensers of Capalbo, Reingold, Vadhan, and Wigderson [Cap+02] require non-explicit gadgets which take too long to find and too much space to store if the starting entropy gap is too large. The condensers of Guruswami, Umans, and Vadhan [GUV07] or Kalev and Ta-Shma [KTS22] do not have small seed length while having constant entropy gap. Another approach would be to use very good extractors, like those of Ta-Shma, Umans, and Zuckerman [TSUZ01], then apply a condenser to concentrate the remaining entropy to get a lossless condenser with small entropy gap. Unfortunately, the extractors of [TSUZ01] or [GUV07] don't appear to be invertible.

The issue with the standard condense and extract framework is that they run many condensers on the same message in parallel. Thus even if individually each condenser is efficiently invertible, it is unclear how to invert them all together efficiently. To see what we mean, consider the condenser which takes a message, x, applies an extractor E to x to output a $0.9k$ bits of entropy, then applies a condenser, C, to x to output a length $0.11k$ bit output containing the remaining $0.1k$ bits of entropy. Then the final result, $C'(x) = (E(x), C(x))$ is indeed a lossless condenser with output length $1.01k$, so has an entropy gap of $0.01k$.

It is not clear how to both time and space efficiently invert C'. For any (y_1, y_2), there are about $2^{n-0.9k}$ values of x such that $E(x) = y_1$, and about $2^{n-0.11k}$ values of x such that $C(x) = y_2$. It is not space efficient to hold all such x in memory, so for every x such that $E(x) = y_1$, we need to check it against every value of x such that $C(x) = y_2$. This would take time around $2^{2n-1.01k}$ to enumerate through every input that condenses to a given output. When $n = \log(N)$, this is around quadratic time, which is too much for our application.

$$(C', I')(x, s_1 \circ s_2 \circ s_3 \circ s_4) = (y_1 \circ y_2 \circ y_3, w_1 \circ w_2).$$

$$\begin{aligned}
(y_1, z_1) =& (E_1, B_1)(x, s_1) && \text{Trevisan Extractor} \\
(z_2, w_1) =& (C_1, I_1)(z_1, s_2) && \text{Multiplicity Condenser} \\
(y_2, z_3) =& (E_2, B_2)(z_2, s_3) && \text{Left-over Hash Extractor} \\
(y_3, w_2) =& (C_2, I_2)(z_3, s_4) && \text{Iterated Multiplicity Condenser.}
\end{aligned}$$

Figure 5: Our Condenser Diagram

This issue is fixed if C does not condense from x directly, but instead condenses from some index function of E, or a buffer as [Cap+02] would call it. Then as long as E and C are invertible, we can invert C' in a sequential way. With this change, condensers made using a condense then extract framework can be made efficiently invertible, as long as the component condensers and extractors are.

Since we are not too concerned about the seed length, our construction will use the extractors of Trevisan [Tre99; RRV99]. To conserve space in the algorithm we use the explicit, log space weak combinatorial designs of Hartman and Raz [HR03]. To make sure that Trevisans extractor is invertible, we need to restrict how large k can be, and handle certain "bad" seeds.

We will also need to use a variant of the condensers of [KTS22], and the left-over hash lemma based extractors [ILL89; NZ96] to finish extracting the rest of the bits in the input, and turn it into a condenser. So the final condenser runs the Trevisan extractor, E_1, to extract all but $O(\log(n/\epsilon)^3)$ bits of entropy. Then we run the multiplicity condenser, C_1, followed by the hash based extractor, E_2, to extract all but $O(\log(1/\epsilon))$ bits of entropy. Finally, we run an iterated version of the multiplicity code condenser, C_2, to condense the remaining entropy into $O(\log(1/\epsilon))$ bits: $y_3 = C_2(z_3, s_4)$. See Fig. 5 for more explicit equations.

Before we start proving the provided condensers exist, we will give our composition theorems.

7.1 Composition Theorems

To construct our condenser, we will compose invertible condensers and extractors together. So we first need to define invertible extractors. Invertible extractors are the same as permutation extractors of [Cap+02], which are themselves a special case of buffered extractors, with the extra condition that the buffered extractor is efficient to invert. Equivalent composition theorems are found in [Cap+02], our only change is including inversion time and space as a consideration. Thus all the condensers of [Cap+02] are also efficiently invertible.

For our extractor composition to work, we need the following observation of distributions. This seems to be well known folklore, so we will not prove it here.

Lemma 7.1 (Approximate Uniform Marginals Keep Entropy). *Suppose for some joint distribution, X, Y, with min entropy k, if X, Y is ϵ close to some joint distribution U, V where U is uniform, then X, Y is also ϵ close to some distribution U, W with min entropy k for the same, uniform U. Additionally, for any $x \in \{0,1\}^{|U|}$, we have $W|U = x$ has entropy $k - |U|$.*

Our next lemma just says if you apply an invertible condenser to the buffer of the invertible extractor, you get a better invertible condenser.

Lemma 7.2 (Invertible Condensers Compose With Extractors). *Suppose there is a time T_1 and space S_1 invertible, (k, ϵ_1), invertible extracter $E : \{0,1\}^n \times \{0,1\}^{d_1} \to \{0,1\}^{m_1} \times \{0,1\}^{n-m_1}$ and a time T_2, space S_2 invertible, $(n - m_1, k') \to_{\epsilon_2} (m_2, k - m_1)$ invertible condenser $C : \{0,1\}^{n-m_1} \times \{0,1\}^{d_2} \to \{0,1\}^{m_2} \times \{0,1\}^{n-m_1-m_2}$.*

Then there is a time $T_1+T_2+O(1)$, space $O(n)+\max\{S_1, S_2\}$ invertible, $(n, k) \to_{\epsilon_1+\epsilon_2} (m_1+m_2, m_1+k')$ condenser $C' : \{0,1\}^n \times \{0,1\}^{d_1+d_2} \to \{0,1\}^{m_1+m_2} \times \{0,1\}^{n+d_1+d_2-m_1-m_2}$.

Proof. For any $x \in \{0,1\}^n$, $s_1 \in \{0,1\}^{d_1}$, and $s_2 \in \{0,1\}^{d_2}$, define

$$\begin{aligned} y_1 =& E^E_{s_1}(x) \\ u =& E^I_{s_1}(x) \\ y_2 =& C^C_{s_2}(u) \\ z =& C^I_{s_2}(u) \\ C'^C_{(s_1,s_2)}(x) =& (y_1, y_2) \\ C'^I_{(s_1,s_2)}(x) =& z. \end{aligned}$$

Then observe that one can compute $C'^{-1}_{(s_1,s_2)}((y_1,y_2),z)$ by

$$\begin{aligned} u =& C^{-1}_{s_2}(y_2, z) \\ x =& E^{-1}_{s_1}(y_1, u). \end{aligned}$$

The space here is just the space to store u plus the max of the space to invert C and E. Similarly the time is the time to invert C and the time to invert E.

Now to show that C'^C is a condenser, first see that given any X with entropy k, since E^E is a strong extractor, we have that for uniform s_1, distribution s_1, y_1 is ϵ close to uniform. Further, since E is invertible, the distribution s_1, y_1, u has the same entropy as X, thus has entropy $k + d$.

By Lemma 7.1 we have that s_1, y_1, u is ϵ close to some distribution s_1, U, u' where s_1 and U are uniform, and u' has entropy $k - m_1$ conditioned on s_1 and U. Let $y_2' = C^C_{s_2}(u')$. Then since C is a condenser, we have that $s_2 y_2'$ is ϵ_2 close to a distribution that has entropy $d_2 + k'$ for every correlated value of s_1 and U. Thus $s_1 U s_2 y_2'$ is ϵ_2 close to a $d_1 + m_1 + d_2 + k'$ source. Thus $s_1, y_1, s_2 y_2$ is $\epsilon_1 + \epsilon_2$ close to being a $d_1 + m_1 + d_2 + k'$ source. Therefore, C^C is a strong $(n,k) \to_{\epsilon_1+\epsilon_2} (m_1 + m_2, k')$, invertible extractor. □

The following theorem just says that if you apply an invertible condenser to the output of a condenser you get a better condenser.

Theorem 7.3 (Invertible Condensers Compose With Condensers)**.** *Given a time T_1 and space S_1 invertible $(n,k) \to_{\epsilon_1} (m_1, k_1)$ invertible condenser $A : \{0,1\}^n \times \{0,1\}^{d_1} \to \{0,1\}^{m_1} \times \{0,1\}^{n-m_1}$ and a time T_2 space S_2 invertible, $(m_1,k_1) \to_{\epsilon_2} (m_2,k_2)$ condenser $B : \{0,1\}^{m_1} \times \{0,1\}^{d_2} \to \{0,1\}^{m_2} \times \{0,1\}^{m_1-m_2}$.*

Then there is a time $T_1 + T_2 + O(1)$ space $O(n) + \max\{S_1, S_2\}$ invertible, $(n,k) \to_{\epsilon_1+\epsilon_2} (m_2,k_2)$ invertible condenser

$$C : \{0,1\}^n \times \{0,1\}^{d_1+d_2} \to \{0,1\}^{m_2} \times \{0,1\}^{n+d_1+d_2-m_2}.$$

Proof. For any $x \in \{0,1\}^n$, $s_1 \in \{0,1\}^{d_1}$, and $s_2 \in \{0,1\}^{d_2}$, define

$$\begin{aligned} u =& A^C_{s_1}(x) \\ z_1 =& A^I_{s_1}(x) \\ y =& B^C_{s_2}(u) \\ z_2 =& B^I_{s_2}(u) \\ C^C_{(s_1,s_2)} =& y \\ C^I_{(s_1,s_2)} =& (z_1, z_2). \end{aligned}$$

Then observe that one can compute $C^{-1}_{(s_1,s_2)}(y,(z_1,z_2))$ by

$$\begin{aligned} u =& B^{-1}_{s_2}(y, z_2) \\ x =& A^{-1}_{s_1}(u, z_1). \end{aligned}$$

This only takes space that is the max of the space to invert B and the space to invert A plus space to hold u. It also only takes the time to invert B and A plus some book keeping.

Since A^c is a strong condenser, s_1, u is an ϵ_1 approximation of some $d_1 + k_1$ source. Then since B^c is a strong condenser, we have that s_1, s_2, y is an $\epsilon_1 + \epsilon_2$ approximation of a $d_1 + d_2 + k_2$ source. □

7.2 Our Base Condenser

For our base condensers that we compose with extractors to make our final condenser, we use the condenser of Kalev and Ta-Shma [KTS22]. These condensers are based on multiplicity codes, which we find easier to understand, and thus invert, then the condensers based on Pavarash-Vardy codes of [GUV07]. While these condensers are great at condensing inputs with k bits of entropy into $O(k)$ bits of output, they can not efficiently condense inputs with k bits of entropy to $k + O(\log(1/\epsilon)$ bits, which is what we need. This is why we need to perform composition to get our final condenser.

Lemma 7.4 (Invertible Lossless Expander). *For every field $\mathbb{F}_q$ and integers $\ell, s \in \mathbb{N}$ with $15 \leq s \leq \ell \leq char(\mathbb{F}_q)$, there exists an explicit graph $\Gamma : \mathbb{F}_q^\ell \times \mathbb{F}_q \to \mathbb{F}_q^s$ which is a (K, A) expander for every $K > 0$ with*

$$A = q - \frac{\ell s}{2}(qK)^{\frac{1}{s}}.$$

Further, Γ is invertible in time $O((\ell \log(q))^2)$ and space $O(\ell \log(q))$.

Proof. This expander is from [KTS22, Theorem 1.3], all we need to show is that Γ is efficiently invertible. Examining Γ, we see it is actually a very straightforward construction based on multiplicity codes. It views its first input as a degree at most ℓ polynomial, p, and its second input as an evaluation point, x. Then it outputs the first s Hasse derrivatives of p evaluated at y.

The expander Γ can be made invertible by also evaluating the remaining $\ell - s$ potentially non zero Hasse derrivatives at y to create the index function. Then one can reconstruct the original polynomial using a Taylor expansion at y. This only takes time $O((n \log(q))^2)$ and space $O(n \log(q))$. ☐

Kalev and Ta-Shma [KTS22] carefully choose parameters to get very good output length, at the cost of severe restrictions on α, k, n, and ϵ. The proof is in [KTS22].

Lemma 7.5 (Kalev and Ta-Shma Condensers). *For every set of integers $k \leq k_{max} \leq n$ and $\epsilon > 0$ with $\frac{16\log(\frac{n}{\epsilon})}{\sqrt{k}} \leq \alpha \leq 1$, there is a time $O(n^2)$ space $O(n)$ invertible $(n, k) \to_\epsilon (m, k + d)$ lossless, invertible condenser $C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n-m}$ with $d = (1 + 1/\alpha)\log(nk_{max}/\epsilon) + O(1)$ and $m \leq (1 + \alpha)k_{max}$.*

These limitations on the parameters of α, k, n and ϵ are inconvenient, so we give an alternate choice of parameters that gives a worse condenser, but are easier for us to work with.

Lemma 7.6 (Our Basic Condenser). *For every set of integers $k \leq k_{max} \leq n$ and $1 > \epsilon > 0$ with $26\log(2n/\epsilon) \leq k_{max} \leq \frac{n}{2}$, there is a time $O(n^2)$ space $O(n)$ invertible $(n, k) \to_\epsilon (m, k + d)$ lossless, invertible condenser*

$$C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n-m}$$

with $d = O(\log(nk_{max}/\epsilon))$ and $m \leq 2k_{max} + O(\log(n/\epsilon))$.

Proof. Let $q' = \left(\frac{2nk_{max}}{\epsilon}\right)^2$, and q be any prime between q' and $2q'$. Let $K = 2^{k_{max}}$. Then we choose s to be the smallest integer so that $q^{s-2} \geq K^2$. Finally we set $\ell = \lceil \frac{n}{\log(q)} \rceil + 2$.

Now we show that we can apply Lemma 7.4. For this, we need to show $15 \leq s \leq \ell \leq q$. To show that

$15 \leq s$, we just need to show that $q^{13} < K^2$. This can be shown by

$$\begin{aligned}\log(q^{13}) =&13\log(q)\\ <&13\log(2q')\\ =&13\log\left(2\left(\frac{2nk_{max}}{\epsilon}\right)^2\right)\\ \leq&13\log\left(2\left(\frac{n^2}{\epsilon}\right)^2\right)\\ <&13\log\left(\left(2\frac{n}{\epsilon}\right)^4\right)\\ <&52\log(2n/\epsilon)\\ <&2k_{max}.\end{aligned}$$

Finally exponentiating both sides gives $q^{15-2} < K^2$, thus s must be at least 15.

To see that $s \leq \ell$, it suffices to show that $q^{\ell-2} \geq K^2$. But this must be true since $q^{\ell-2} \geq 2^n \geq 2^{2k_{max}} = K^2$. Finally it is clear that $q \geq n$ since $q \geq q' > n$. Thus we can apply Lemma 7.4.

Due to Lemma 7.4, there is an explicit graph $\Gamma : \mathbb{F}_q^\ell \times \mathbb{F}_q \to \mathbb{F}_q^s$ which is a (K, A) expander for every $K > 0$ with

$$A = q - \frac{\ell s}{2}(qK)^{\frac{1}{s}} = q\left(1 - \frac{1}{q}\frac{\ell s}{2}(qK)^{\frac{1}{s}}\right)$$

Further, Γ is invertible in time $O((\ell\log(q))^2) = O(n^2)$ and space $O(\ell\log(q)) = O(n)$.

Now we want to show that A, the expansion rate of size K sets, is very close to q, the degree, to get a lossless condenser. To do this, we show that $\frac{1}{q}\frac{\ell s}{2}(qK)^{\frac{1}{s}}$ is at most epsilon. See that since $q^{s-2} \geq K^2$, we also have $K \leq q^{s/2-1}$. We also have that $q^{s-3} < K^2$, thus $s < \frac{2k_{max}}{\log(q)} + 3 < 2k_{max}$ since $k_{max}, q > 16$. We also have that $\ell < n$. Thus

$$\begin{aligned}\frac{1}{q}\frac{\ell s}{2}(qK)^{\frac{1}{s}} \leq&\frac{1}{q}\frac{\ell s}{2}(qq^{s/2-1})^{\frac{1}{s}}\\ \leq&\frac{\ell s}{2\sqrt{q}}\\ <&\frac{n2k_{max}}{2\sqrt{q'}}\\ =&\epsilon/2.\end{aligned}$$

Thus we can bound A by

$$A \geq q(1 - \epsilon/2).$$

Also see that $q^\ell > n$. Then our final condenser just interprets the input as an element of $\mathbb{F}_q^\ell$, and, its seed as an element of $\mathbb{F}_q$, and outputs an element of $\mathbb{F}_q^s$. On a technical note, q is not a power of 2, and in fact may be far from a power of 2. While we could work with this, to get our stated result, we will need to sample the same element of $\mathbb{F}_q$ for multiple seeds so that our distribution of seeds is approximately uniform. We can $\epsilon/2$ approximate a uniform distribution over $\mathbb{F}_q$ with $\log(q) + O(\log(1/\epsilon))$ extra bits, which we do. But to avoid collisions, we need to include the extra $O(\log(1/\epsilon))$ bits of seed in the output.

We claim this is a $(n, k) \to_\epsilon (m, k + d)$ condenser with seed length is $d = \log(q) + O(\log(1/\epsilon)) = O(\log(n/\epsilon))$, and output bit length $s\log(q) + O(\log(1/\epsilon)) = 2k_{max} + O(\log(q) + \log(1/\epsilon)) = 2k_{max} + O(\log(n/\epsilon))$. The seed length and output bit is given directly from Γ and the extra padding needed to work over bits.

To see that it is a condenser, we simply observe that any flat k source is a uniform distribution over an element of $\mathbb{F}_q^\ell$, and if our seed was uniform over $\mathbb{F}_q$, then the output could only have at most $\epsilon/2$ fraction of collisions, thus is an $\epsilon/2$ approximation of a $k + \log(q)$ source. Since our seed is an $\epsilon/2$ approximation of an element of $\mathbb{F}_q$ and the extra $d - \log(q)$ entropy is copied directly to the output, our output is an ϵ approximation of a $k + d$ source. □

We want to use our condenser both where k is around $O(\log(n/\epsilon))$ and when $k = O(\log(1/\epsilon))$. Lemma 7.6 is already good enough when $k = O(\log(n))$, but the restriction of $k_{max} = \Omega(\log(n/\epsilon))$ makes our condenser output length too long when $k = O(\log(1/\epsilon))$. By composing this condenser with itself $\log^*(n)$ many times we get a condenser that handles even constant entropy k. Since each successive instance is so much smaller than the first, this gives the same asymptotic time, space, and seed length.

Corollary 7.7 (Iterated Basic Condenser). *For every set of integers $k \leq k_{max} \leq \frac{n}{2}$ and $\frac{1}{2} > \epsilon > 0$ with $100\log(1/\epsilon) \leq k_{max} \leq n$, there is a time $O(n^2)$ space $O(n)$ invertible, $(n,k) \to_\epsilon (m, k+d)$ lossless, invertible condenser $C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n-m}$ with $d = O(\log(nk_{max}/\epsilon))$ and $m \leq 2k_{max} + d + O(\log(1/\epsilon))$.*

7.3 Our Condenser for Polylogarithmic Entropy

While the proceeding condensers are quite good, they cannot by themselves get a constant entropy gap. That is, they can't give number of output bits with $m = k + O(\log(1/\epsilon))$ unless $k = O(\log(1/\epsilon))$. Here, we show we can give a lossless condenser, albeit with very long seed length, that outputs all $k+d$ bits of entropy into a length $k + d + O(\log(1/\epsilon))$ bit output.

This will just use the left-over hash lemma based extractor [ILL89] composed with our iterated condenser, Corollary 7.7 to get a condenser with an $O(\log(1/\epsilon))$ entropy gap. So first, we will state the extractor given by the left-over hash lemma.

Lemma 7.8 (Base Case Extractor). *For any $n > k > 0$ and ϵ, there is a time $O(n^2)$, space $O(n)$ invertible (k, ϵ) extractor*

$$E : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n-m}$$

with seed length $d = O(n)$, and $m = k + d - O(\log(1/\epsilon))$.

Proof. This is an extractor based on a pairwise independent hash function. The soundness is based on the well known leftover hash lemma. For invertibility, we just use the hash function that views input x as an element of $\mathbb{F}_{2^n}$, and the seed as two elements $a, b \in \mathbb{F}_{2^n}$ and outputs the m least significant bits of

$$ax + b$$

and the seed. And the index function are the $n - m$ most significant bits of $ax + b$. Then inversion is straightforward. □

Now we can use our extractor with our multiplicity based condenser to get a new condenser.

Corollary 7.9 (Base Case Condenser). *For any $n > k > 0$ and ϵ, there is a time $O(n^2)$, space $O(n)$ invertible, $(n,k) \to_\epsilon (m, k+d)$ condenser*

$$C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n+d-m}$$

with seed length $d = O(n)$, and $m = k + O(\log(1/\epsilon))$.

Proof. By Lemma 7.8, there is an extractor a time $O(n^2)$, space $O(n)$ invertible $(k, \epsilon/2)$ extractor

$$E : \{0,1\}^n \times \{0,1\}^{d_1} \to \{0,1\}^{m_1} \times \{0,1\}^{n-m_1}$$

with seed length $d_1 = O(n)$, and $m_1 = k + d_1 - O(\log(1/\epsilon))$. Let $k_1 = k + d_1 - m_1 = O(\log(1/\epsilon))$

By Corollary 7.7, there is a time $O(n^2)$ space $O(n)$ invertible, strong $(n - m_1, k_1) \to_{\epsilon/2} (d_2 + O(k_1), d_2 + k_1)$ invertible condenser

$$C_1 : \{0,1\}^n \times \{0,1\}^{d_2} \to \{0,1\}^{m_2} \times \{0,1\}^{n-m_2}$$

with $d_2 = O(\log(nk_{max}/\epsilon))$ and $m_2 \leq 2k_1 + d_2 + O(\log(1/\epsilon))$.

By Lemma 7.2 the final result is a time $O(n^2)$, space $O(n)$ invertible $(n,k) \to_\epsilon (m_1 + m_2 = k + d_1 + d_2 + O(\log(1/\epsilon)), m_1 + d_2 + k_1 = k + d_1 + d_2)$ condenser

$$C : \{0,1\}^n \times \{0,1\}^{d_1+d_2} \to \{0,1\}^{m_1+m_2} \times \{0,1\}^{n-m_1-m_2}.$$

Finally see that the seed length of C is $d_1 + d_2 = O(n)$. □

7.4 Final Condenser

Now to construct our final condenser, we first start with an invertible extractor that can extract most of the entropy (all but *polylog*(n/ϵ)). We start with the time and space efficient variation of Trevisan's extractor [Tre99; RRV99] by Hartman and Raz [HR03]. This extractor is invertible because it is linear, a fact commonly used by non-malleable extractors [CDS12; Li17]. We note that this choice of extractor (and the techniques used to invert it) are the main bottleneck to getting log space and quasilinear time encoders. Better seed length and linear space inversion of our condenser would give us better encoders.

The Trevisan extractor algorithm depends on two constructions, a code with good distance, and a weak combinatorial design. We use the same code and weak design as [HR03].

Lemma 7.10 (Invertible Trevisan Extractor)**.** *For every n, k and ϵ such that $k \le n$ and $\epsilon > 2^{1-n/\log^*(n)^{\log^*(n)}}$, there is a time poly$(n)$, space $O(n^2)$ invertible (k, ϵ) extractor*

$$E : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n+d-m}$$

with seed length $d = \Theta(\log(n/\epsilon)^3)$ and $m = k$

Proof. We start with the extractor in [HR03, Theorem 10] using $\epsilon/2$ in place of ϵ.

This immediately gives us a $(k, \epsilon/2)$ extractor

$$E' : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$$

with seed length $d = O(\log(n/\epsilon)^3)$ and $m = k$. Unfortunately, E' is probably not invertible. To convert it into an invertible extractor, we need to first explain how E' is computed.

The extractor E' is built using a code, $C : \{0,1\}^n \to \{0,1\}^{n'}$, where n' is a power of two and a weak combinatorial design, $S : [m] \to [d]^{\log(n')}$. Given an input x and a seed s, the extractor E' outputs

$$y = (C(x)|_{s(S(1))}, C(x)|_{s(S(2))}, \ldots, C(x)|_{s(S(m))})$$

where $C(x)|_{s(S(i))}$ means to first concatenate the bits in s indicated by $S(i)$, and then use the resulting string to index into C.

Importantly, S can be computed in $\log(n)$ space and time *poly*(n). And C is a Reed-Solomon code concatenated with a Hadamard code. In particular, for some a with $\log(n/\epsilon) < a < 4\log(n/\epsilon)$, code C is the Reed Solomon codes over $\mathbb{F}_{2^a}$ with degree at most n/a composed with the Hadamard code. The important thing about this extractor is that after a given seed is chosen, the extractor is a linear function whose generator matrix can be found in polynomial time and space $O(n^2)$.

So if for a given seed s, all the output bits are linearly independent, then one can create a full rank matrix where the first k rows output the extractor by a greedy search in *poly*(n) time and $O(n^2)$ space. Applying this matrix, and then appending the seed, gives the buffered extractor. And by Guassian elimination, one can invert this matrix again in *poly*(n) time and $O(n^2)$. Thus for these 'good' seeds, one where the output bits are linearly independent, one can invert this extractor efficiently.

When the seed is 'bad', that is the output bits are not linearly independent, one can also detect this in time *poly*(n) and space $O(n^2)$, again using Guassian elimination. For these bad seeds, the extractor already fails, so we just output the entire input. This is trivially invertible.

So in our final invertible extractor, we take an input x and a seed s and first check if it is bad. If it is, we give up and output the input and the seed. Otherwise, we output for our extractor part $E'(x, s)$ and for our buffer part s along with the rest of the information need to invert E'. The buffer here only needs to output a d bit seed, plus the information about x missing from the extractor, which is just $n - k$ bits, exactly what we need: an $n + d - m$ bit buffer.

To see the result is an extractor, all we need to note is that the seed is bad rarely. This is because E' outputs an $\epsilon/2$ approximation of the uniform distribution and when a seed is bad, since the extractor is linear, it E' can only hit at most half the space of outputs. So when the seed is bad, that seed has distance $1/2$ from uniform. So E only doubles the error on bad seeds and maintains the same error on all other seeds. Thus the error of E is at most ϵ. □

Now we can construct our final condenser by composing our invertible Trevisan Extractor, Lemma 7.10 with our multiplicity condenser Lemma 7.6 and our base case condenser Corollary 7.9. We now prove Theorem 1.4.

Theorem 1.4 (Good Invertible Condensers Exist)**.** *For every n, k and ϵ such that $\epsilon > 2^{3-n/\log^*(n)^{\log^*(n)}}$, there is a time poly$(n)$, space $O(n^2)$ invertible, lossless $(n,k) \to_\epsilon (m, k+d)$ condenser*

$$C : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m \times \{0,1\}^{n+d-m}$$

with $d = O(\log(n/\epsilon)^3)$ and $m = k + d + O(\log(1/\epsilon))$.

Proof. For small enough constant n, we can just use the probabilistic method to find the condenser. So take n to be a sufficiently large value. If $k > n/2$, we can just increase n to $2k$ by padding inputs. The resulting condenser will work equally well on length n inputs.

We start by stating our extractor and two condensers we will compose along with the parameters we use. Then we compose them from smallest to largest to get the final condenser.

By Lemma 7.10, we have a time *poly*(n), space $O(n^2)$ invertible $(k, \epsilon/4)$ extractor

$$E : \{0,1\}^n \times \{0,1\}^{d_1} \to \{0,1\}^{m_1} \times \{0,1\}^{n+d_1-m_1}$$

with seed length $d_1 = \Theta(\log(n/\epsilon)^3)$ and $m_1 = k$. Let $n_1 = n + d_1 - m_1$ and $k_1 = d_1$.

Now to use Lemma 7.6, we need $26 \log(64n_1/\epsilon) \le k_1 \le \frac{n_1}{2}$. Well since $m_1 = k \le n/2$ and, for large enough n, we have $k_1 = d_1 = \Theta(\log(n/\epsilon)^3) < n/2$ we have that

$$\begin{aligned} n_1 =& n + d_1 - m1 \\ \ge& n/2 + d_1 \\ >& 2k_1. \end{aligned}$$

Similarly for large enough n, we know that $k_1 = d_1 = \Theta(\log(n/\epsilon)^3) > 26\log(64n_1/\epsilon)$.

So by Lemma 7.6, there is a time $O(n^2)$ space $O(n)$ invertible $(n_1, k_1) \to_{\epsilon/4} (m_2, k_1 + d_2)$ invertible condenser

$$C_1 : \{0,1\}^{n_1} \times \{0,1\}^{d_2} \to \{0,1\}^{m_2} \times \{0,1\}^{n_1 - m_2}$$

with $d_2 = O(\log(n_1 k_1/\epsilon))$ and $m_2 \le 2k_1 + O(\log(n_1/\epsilon))$.

And lastly, by Corollary 7.9 there is a time $O(m_2^2) = O(n^2)$ space $O(m_2) = O(n)$ invertible $(m_2, k_1 + d_2) \to_{\epsilon/2} (m_3, k_1 + d_2 + d_3)$ condenser

$$C_2 : \{0,1\}^{m_2} \times \{0,1\}^{d_3} \to \{0,1\}^{m_3} \times \{0,1\}^{m_2+d_2-m_3}$$

with seed length $d = O(m_2) = O(\log(n/\epsilon)^3)$ and output length $m_3 = k_1 + d_2 + d_3 + O(\log(1/\epsilon))$.

Now to compose these condensers. First, we compose C_1 and C_2 using Theorem 7.3 to get a time $O(n^2)$ space $O(n)$ invertible $(n_1, k_1) \to_{\frac{3\epsilon}{4}} (m_3, k_1 + d_2 + d_3)$ invertible condenser

$$C_3 : \{0,1\}^{n_1} \times \{0,1\}^{d_2+d_3} \to \{0,1\}^{m_3} \times \{0,1\}^{n_1+d_2+d_3-m_3}.$$

Now we compose C_3 with E using Lemma 7.2 to get a time *poly*(n), space $O(n^2)$ invertible $(n,k) \to_\epsilon (m_1 + m_3, m_1 + k_1 + d_2 + d_3)$ condenser

$$C' : \{0,1\}^n \times \{0,1\}^{d_1+d_2+d_3} \to \{0,1\}^{m_1+m_3} \times \{0,1\}^{n+d_1+d_2+d_3-m_1-m_3}.$$

Finally, see that see that since $k_1 = d_1$, that for $d = d_1 + d_2 + d_3$ we have that $m_1 + m_3 = k + d + O(\log(1/\epsilon))$ and the output entropy is $m_1 + d = k + d$. ∎

8 Spielman Style Weight Fixers

If one does not consider decoding, one can naturally characterize Spielman codes in terms of weight amplifiers. This perspective is useful for us, as we cannot afford to use full Spielman codes. Instead, we only use them to amplify the weight of already heavy messages and leave lighter messages to be fixed by our condenser based weight fixers.

The idea is to amplify the weight while reducing the output size. If you do this a few times, it will amplify the weight until you have a heavy string at some stage. You need to reduce the size each time so that you end up with a linear length string. Now that you know some stage of the repeated condensing has large weight, now you apply the same weight amplifier again on each stage and the outputs of the smaller stages, starting from the bottom, to pull that weight back up. This is the same thing Spielman's code does, but our analysis can be much simpler since we are not trying to decode.

Our basic tool is a weight amplifier, which is only promised to increase the weight of any light enough inputs by a constant factor. Notably, it could output zero weight for inputs that are already very large. This is necessary since we also want outputs that shrink the input.

Definition 8.1 (Weight Amplifier). *For any $R < N$, any alphabets Σ_1 and Σ_2 composed of binary bits, and constant $\delta > 1$, a linear function $A : \Sigma_1^N \to \Sigma_2^M$ is said to be an R to δ weight amplifier if, given any x with x with $weight(x) \leq R$ then $weight(A(x)) \geq \delta\, weight(x)$.*

We say that R is the input weight range, δ is the relative output weight, N is the input length, and M is the output length.

The main component of our weight amplifiers are lossless expanders. The following lossless expanders are from [Cap+02, Theorem 7.1].

Lemma 8.2 (Constant Degree, Lossless Expanders). *For some constant c, for every $N, T \leq N$ and $\epsilon > 0$, there is a D to DT regular bipartite graph $G : [N] \times [D] \to [N/T]$ that is a $\left(\frac{cN}{DT\epsilon}, D(1-\epsilon)\right)$ expander where $D = poly(T/\epsilon)$.*

Further, G is invertible in time $poly(\log(N), 1/\epsilon, T)$ and space $O(\log(N) + poly(T/\epsilon))$. By invertible, we mean that for some $G' : [N] \times [D] \to [N/T] \times [DT]$, where G' restricted to its first component is G, the function G' is invertible in this time and space.

There is also an additional $2^{poly(1/\epsilon)}$ preprocessing time.

Proof. The same theorem is given in [Cap+02, Theorem 7.1], except that they use the language of conductors instead of expanders. These are equivalent just by taking the log or exponent of their parameters appropriately. All we note here is that all of the component conductors in [Cap+02] are efficiently invertible, and thus so are their compositions. See Section 7.1 for more details. ☐

Now using the lossless expanders above, we can give weight amplifiers. Our weight amplifiers, as all of our weight fixers, just output for every vertex on the right the xor of all its adjacent message bits on the left. This structure is important to know for doing fine grain analysis of the space used by a weight fixer that applies many weight amplifiers as subroutines.

Lemma 8.3 (Shrinking Weight Amplifiers Exist). *For some constant $\alpha > 0$ and constants d, c, for every even N, there exists an αN to 2 weight amplifier $A : \{0,1\}^N \to \{0,1\}^{N/2}$.*

Further, given any output bit, $i \in [N/2]$, the ith output bit of A is just the xor of at most c many input bits to A, and any of those input indices can be computed in $polylog(n)$ time and $O(\log(n))$ space.

Proof. This is given by first using Lemma 8.2 by setting with $\epsilon = \frac{1}{4}$ and $T = 2$ to get a function $G : [N] \times [D] \to [N/2]$ for some constant D that is an $(\alpha N, \frac{3}{4}D)$ expander for some constant $\alpha > 0$. Then A just maps every bit according to G, and takes the parity of every bit mapped to a right hand vertex. Computing this parity is efficient because G is efficiently invertible and only requires constantly many queries since D is constant.

To see that it is a weight amplifier, we first observe that since it is a $\frac{3}{4}D$ expander, we must have $D \geq 4$. Thus for any set, S, with less than αN left verticies, has at least $D/2 \geq 2$ of its neighbors with a unique neighbor in S. Therefore, the parities in these bits must be 1, and thus the output of L have at least $2|S|$ ones in it.

We note the preprocessing only takes constant time since ϵ is constant. ☐

Now one can use this shrinking weight amplifier recursively to make a constant rate, constant distance code. This is what Spielman does. But such a code is not simultaneously space and time efficient. Alternatively, we can apply this recursion a bounded number of times to get a constant rate, constant weight, weight fixer that fixes already high weight messages. Using more levels of recursion increases the range of weights that can be fixed.

Lemma 8.4 (Single Step In Spielman Style Recursion)**.** *For the constants $\alpha > 0$, c and d from Lemma 8.3, suppose for some N and $M \leq N$ there is an $[M, N]$ to $\alpha/4$ weight fixer $F : \{0,1\}^N \to \{0,1\}^{4N}$. Further, suppose that any output bit of F is an xor of at most ℓ different input bits to F, and any of these input bit indexes can be computed in time T and space S.*

Then there is an $[M/2, 2N]$ to $\alpha/4$ weight fixer $F' : \{0,1\}^{2N} \to \{0,1\}^{8N}$. Further any output bit of F' is an xor of at most $c^2\ell$ input bits, and any of these input bit indexes can be computed in time $T + O(polylog(N))$ and space $O(1) + \max\{S, O(\log(n))\}$.

Proof. Our weight fixer is the same as Spielman's: we take an input x, and first apply the weight amplifier of Lemma 8.3 to get an output y. Then we apply the weight fixer from the lemma assumption to y to get an output, z. Finally, we apply Lemma 8.3 to z to get the output w. Then the final output of our weight fixer is (x, z, w). See that $|x| = 2N$, $|y| = N$, $|z| = 4N$ and $|w| = 2N$.

To see that this works, we break our problem into cases.

1. $weight(x) > 2\alpha N$. Then since x is included in the output, the relative weight of our new fixer is at least $\alpha/4$.
2. $M/2 \leq |x| \leq 2\alpha N$, then we have that $y \geq M$. Thus by the weight fixing property of F, we have that z has weight at least $weight(z) \geq \frac{\alpha}{4}|z| = \alpha N$. Then we break this into two more cases.
   (a) $weight(z) \geq 4\alpha N$. Then the relative weight of the output is at least $\alpha/2 > \alpha/4$.
   (b) $\alpha N \leq weight(z) \leq 4\alpha N$. Then $weight(w) \geq 2\, weight(z) \geq \alpha 2N$. Thus the relative weight of the output is at least $\alpha/4$.

Now we show time and space needed to compute F'. Output bits from x are just input bits and can be given directly. Output bits from z are just an xor of at most ℓ different elements from y, who are themselves an xor of at most c elements of x. Thus a bit of z just an xor of $c\ell$ elements of x. Any individual bit of which can be looked up in time $T + polylog(n)$ due to the lookup time of F and Lemma 8.3. The space is just to either compute the neighbor in Lemma 8.3, or from F, which can be reused. Similarly, bits in w are just the xor of c bits of z, which are xor of $c\ell$ bits of x. So bits in w are xors of $c^2\ell$ bits of x, whose indexes be efficiently computed for the same reason. □

Now applying this recursion a bounded number of times, we can get a weight fixer which is time and space efficient, but only fixes the weights of messages that are already heavy. We now prove Theorem 6.2.

Theorem 6.2 (Weight Fixer For Heavy messages)**.** *Take any integer K, and any integer N such that $N = 2^i K$ for some i. Then, for some constant $\alpha > 0$, there is a $[K/2^i, N]$ to $\alpha/4$ weight fixer $F : \{0,1\}^N \to \{0,1\}^{4N}$. Further, for some constant, c, any bit in the output of F can be computed in time $c^i\, polylog(N)$ and space $O(i + \log(N))$.*

Proof. This comes from applying Lemma 8.4 for i many times to the trivial weight fixer that just repeats a K bit input 4 times. We can see that each application of Lemma 8.4 increases the number of bits in the input by a factor of 2, and decreases the lower bound of the fixer range by a factor of 2, giving the claimed fixer input range and output weight.

To see the performance, see that the outputs of F must be the xor of at most c^{2i} input bits, any of which can be computed in time $i\, polylog(n)$ and space $O(i + \log(n))$ by recursion. Then one need only iterate through each of the c^{2i} bits and xor them, which only takes time c^{2i} times the time to compute a single bits index, and space $O(2i)$ to store which bit we are on, plus the space to compute a single bit index. □

9 Encoders With Sequential Access To The Message

Now we discuss the resources needed for encoding codes in a model of computation where the program only has sequential like access to the message. This model of computation is useful for low space, black box composition of two low space algorithms. First we will show lower bounds in this model. Then we will give codes that can be encoded in nearly the same time and space as those lower bounds, proving they are tight. Finally, we will give some basic relationships between different kinds of sequential access using these upper and lower bounds.

9.1 Lower Bounds

Before we start our lower bounds, let us first clarify what we mean by space and time of an algorithm.

Remark (Space And State Of An Algorithm)**.** *To simplify our proof here, we will refer to the space of an algorithm as the size needed to hold its entire state, including:*

- *Its work tape.*
- *All head locations.*
- *Number of bits written.*

We assume that an algorithm always prints its output bits in order, so the number of bits written is enough to know which bit will be output next. Space does not *include the bits printed so far, or the input bits.*

If one interprets the space, S, to just be the size of the work tape, since we assumed $S \geq h\log(N)$, using our alternative definition of space only increases S by a constant factor, so our final results hold. So for simplicity, we assume S is the space needed to store its entire state.

Remark (On Time And Uniformity)**.** *In this specific section on sequential lower bounds, we will consider a non-uniform model of computation. We allow the algorithm with sequential access to the input to use any function to define it's state transitions and head movements. The only condition is that such transitions are only functions of the working tape and whatever bits are under the heads to the input.*

In particular, the time in our algorithm lower bounds is actually the number of head movements.

Our lower bounds work by partitioning the message into intervals and showing that most intervals need to be visited many times for the code to have good distance. This is because our space is bounded, so not much can be remembered about an interval when the heads leave it. So it must be visited many times for each of the different possible messages in that interval to have different things written to the code when no head is in that interval. Then the fact that our access to the message is sequential and we have few heads makes visiting an interval slow, requiring a long time to visit each interval enough times.

First, we need to formalize the idea that our algorithm can not have one of its heads enter new intervals in the message very often. But this is not true if we just count how many times the algorithm moves a head into an interval it was not in. As a counterexample, suppose a head is right next to the boundary of two intervals. Then the head can move in and out of it once every two time steps to visit it many times. In fact every interval may have been visited many times. So we need a more strict notion of visiting an interval.

So instead, we want to only count the number of times an interval transitions from having no head near it (so in one of its neighboring intervals) to having a head in it. In this setup, our algorithm really has to spend a full intervals length worth of time transitioning from having every head far from an interval to having one inside it. So to formally describe this, we introduce interval marking, where we mark an interval when a head enters it, and only unmark it when all heads are far. It requires a lot of time to mark an interval after it has been unmarked.

Before we define a marking, we need to define the distance of an interval to a head. This is defined in the obvious way.

Definition 9.1 (Distance)**.** *For any set $S \subseteq [N]$ and any head locations $H \subseteq [N]$ we define our distance between S and H as*

$$\Delta(H, S) = \min_{h\in H, s\in S} |h - s|.$$

Now we can define our interval markings for an algorithm.

Definition 9.2 (Interval Marking). *Let A be an algorithm running in time T and space S with sequential head access to the message and an interval length I.*

For a length N message x, let $M = \lceil \frac{N}{I} \rceil$. Partition the message into M length I intervals: $B_1, \ldots, B_m$ where $B_i = (I(i-1), Ii]$ with $B_m = (I(m-1), n]$. A marking of the intervals is just a set $a \in \{0,1\}^m$.

For an algorithm A on a message x, for $t \in [T]$, let H^t be the set of intervals that A running on message x has a head in at time t.

Then we inductively define a marking of A on message x. a^0 has nothing marked: $a^0 = 0^i$. At any subsequent time step t with head positions H, we define a^t by

$$a_i^t = \begin{cases} 1 & H^t \cap B_i \neq \emptyset \\ 0 & \Delta(H^t, B_i) \geq I \\ a_i^{t-1} & \textit{otherwise.} \end{cases}$$

The sequence $a^0, \ldots, a^T$ is the I marking of A on message x.

We say interval B_i is covered at time t during algorithm A if $a_i^t = 1$. We say A marks interval B_i at time t if $a_i^t = 1$ but $a_i^{t-1} = 0$ and A marks interval B_i at time t if $a_i^t = 0$ but $a_i^{t-1} = 1$. We say there is a marking at time t if A marks any interval at time t, and there was an unmarking at time t if A inmarks any interval at time t.

We now emphasize, we use markings to refer to when an interval changes from uncovered to covered, and unmarking to when an interval changes from covered to uncovered. Marking always refers to this *change*, while cover always refers to how things *are*. For example, number of markings is how many times intervals *change* to be covered.

Now using our terminology, we can formalize our argument. First, its straightforward to observe that if there are few heads, most intervals are uncovered as no heads are near them.

Lemma 9.3 (Max Number Of Covered Intervals). *For any algorithm A running in time T, space S, and h heads, at any $t \in [T]$ the total number of intervals covered in an I marking of A on a length N message are at most $3h$.*

Proof. See that if for any i, interval B_i is only covered if $\Delta(H, B_i) < I$. For any head h A has at time t, there are only at most 3 intervals that can be within I of h. Specifically, the one that h is in and the two next to it. Thus each individual head only covers at most 3 intervals, and there are only h heads, so only $3h$ intervals can be covered. ☐

Now we show that it takes a long time to mark many intervals.

Lemma 9.4 (Max Number of Markings/Unmarkings). *For any algorithm A running in time T, space S, and h heads, any marking of A on a length N message makes at most $1 + T/I$ markings, and $1 + T/I$ unmarkings.*

Proof. The idea is that after the first step and the first marking, each interval will take time I to get through to mark the next one. Then we only unmark an interval once it will take time I to mark it again.

We only bound the number of markings, since the number of unmarkings is less than the number of markings.

We formally bound the number of markings by showing each time step can only "contribute" to marking one interval, and that every marking of an interval needs at least I "contributions" every time it is marked.

So we say any step t contributed to a marking of interval i if the head A moves at time t is in an interval adjacent to interval i and A moves that head toward interval i. We see that by this definition, A only contributes to one interval i per time step, since A can only move one head, and it either moves that head towards the interval above or below it. Let i_t be the interval that A contributes to at time t.

Suppose at any time t, the head positions of A are H^t. Then for any interval B_i see that if $\Delta(H^t, B_i) \leq I$ and $\Delta(H^{t+1}, B_i) < \Delta(H^t, B_i)$ then it must be because $i_t = i$. Otherwise the closest head to B_i must not have moved toward it. Thus for the distance to decrease beyond I, it must be due to contributions from A.

Further, if $\Delta(H^{t+1}, B_i) < \Delta(H^t, B_i)$, then $\Delta(H^{t+1}, B_i) = \Delta(H^t, B_i) - 1$ since heads can only move one position per time step. Thus to change $\Delta(H^t, B_i) \geq I$ to $\Delta(H^{t+t'}, B_i) = 0$ requires at least I contributions from A to B_i.

Finally, see that after the first marking of the first interval that the distance of every uncovered interval to a head starts out at least I. And that every interval that is unmarked also starts with distance I from any head.

Thus, after the first interval is marked, every new marking of an interval requires at least I contributions from A. And A can only contribute to one interval per time step. Thus A can only have at most $1 + T/I$ markings. Similarly, every unmarking must come from exactly one other marking, we also have at most $1 + T/I$ unmarkings. □

Now we can handle a special case: when the encoder is non-adaptive. A non-adaptive algorithm is an algorithm that always reads the same bits in the same order (for a given input size), regardless of the contents of those bits. As a warm up, we will prove our lower bounds for the non-adaptive case.

Theorem 9.5 (Lower Bounds For Encoders With Sequential Access)**.** *Suppose C is a code with distance δ encoding N bits. Suppose A is a non-adaptive algorithm computing C running in time T space S and using h sequential heads to access the message. Further assume $S > h\log(N)$. Then*

$$hST = \Omega(N^2\delta).$$

Proof. The idea is that if any message interval is not marked often enough, than the state of the algorithm when that interval is not covered will look the same for two different contents of that interval. Thus we can find two messages that will give exactly the same output for any bits written while that interval is not covered. Thus if any interval is both

1. unmarked too few times and

2. uncovered when too many output bits are written

then two different messages will have encodings with little distance. We will show that most intervals must have both if $hST \ll N^2\delta$, so in particular some interval has both.

First we set the interval size to be $I = \frac{\delta N}{8h}$ so there are at least $\frac{N}{I} = \frac{8h}{\delta}$ intervals. Now we can show that most intervals are covered when at most a δ fraction of output bits are written. Since each head can only cover at most 3 intervals in a time step at any given time, at any time only at most $3h$ intervals are covered. So at most a $\frac{3\delta}{8}$ fraction of intervals are covered when any bit is written. Thus the number of intervals that are covered when at least a δ fraction of the time steps output bits are written is at most $\frac{3}{8}$.

Now we show that few intervals are unmarked frequently. Intuitively, each time an interval is unmarked, it reveals at most S bits about that interval, so we need an interval to be visited $\frac{I}{S}$ times for the entire interval to be revealed. So we want to bound the number of intervals that have been unmarked $\frac{I}{S}$ times.

We know from Lemma 9.4 that there are at most $1 + T/I \leq 2\frac{T}{I}$ unmarkings. Then the number of intervals that are unmarked at least $\frac{I}{S}$ times is at most $\frac{2ST}{I^2}$. Since there are at least $\frac{N}{I}$ intervals, at most $\frac{2ST}{NI} = \frac{16STh}{\delta N^2}$ fraction of intervals are unmarked more than $\frac{I}{S}$ times.

Now if we assume for contradiction that $hST < \frac{\delta N^2}{32}$, we have that at most half of the intervals are unmarked more than $\frac{I}{S}$ times. Since at most a $\frac{3}{8}$ fraction of the intervals have at least a δ fraction of output bits written when they are covered, that means that a $\frac{1}{8}$ fraction of intervals are both

1. unmarked less than $\frac{I}{S}$ times and

2. covered when less than a δ fraction of output bits are written.

Take one such interval (which is not the final interval, so has length I), call it interval B_i.

Now we will create two adversarial messages for the algorithm that will not have good distance for the code. We do this by showing that two messages look the same when interval B_i is not covered. Let x^* be the restriction that sets everything outside interval B_i to zero.

Now for any assignment, y, to interval B_i, define $R(y)$ to be the tuple of the states of the algorithm on input $x_y = y \circ x^*$ every time B_i is unmarked. Since B_i is unmarked less than $\frac{I}{S}$ times, the output of f has length less than I. Thus for two different y, call them y_1 and y_2, we have that $R(y_1) = R(y_2)$. Then for input $x_1 = y_1 \circ x^*$ and $x_2 = y_2 \circ x^*$, we have that A acts on x_1 and x_2 exactly the same when B_i is uncovered. Then since less than a δ fraction of output bits are written when B_i is covered and A has the same output for both when B_i is uncovered, $C(x_1)$ can only differ from $C(x_2)$ on at most a δ fraction of codeword bits.

Thus C has distance less than δ. Contradiction, so we must have $hST \geq \frac{\delta N^2}{32} = \Omega(\delta N^2)$. □

The argument becomes a bit more complex when we allow the algorithm to be adaptive. Specifically, which intervals are marked frequently may change depending on the message, as are the intervals that are uncovered frequently. Specifically, our adversarial inputs set everything outside of one interval to 0. If the algorithm knows that everything outside one interval will be zero, than it can detect which interval is non-zero and spend all its time on that interval.

To get around these issues, we choose messages randomly. For any random message, for a random interval, with high probability that interval is only covered when a small fraction of the output bits are written, and that interval is unmarked few times. Equivalently, we can select a random interval, randomly restrict everything outside that interval, than randomly assign that interval. These sample the same distribution, so we also have that for most random intervals chosen, B_i, and most restrictions outside B_i, it must be that for most assignments to B_i we have that B_i is unmarked few times and uncovered when most output bits are written.

So we define a good restriction to be such a restriction, and then show that good restrictions give distinct messages that have too close code words. Finally we show good restrictions exist when $hST \ll \delta N^2$.

Definition 9.6 (A Good Restriction of the Message)**.** *For any algorithm A with time T and space S, given distance δ and interval length I, we say a restriction x^* is good for interval i with distance δ on algorithm A if the following holds.*

1. *x^* fixes every bit outside of the interval B_i, and leaves every bit inside B_i unfixed.*
2. *Let Y be the set of assignments for B_i such that for $x = y \circ x^*$ we have:*
   - *(a) A on message x writes at most δ fraction of its bits when B_i is covered.*
   - *(b) A on message x unmarks B_i at most $\frac{8T}{N}$ times.*

   We also require that $|Y| > 2^I/2$.

Now we show that a good restriction is enough to give our lower bounds.

Lemma 9.7 (Good Restrictions Give Lower Bounds)**.** *Suppose A is an algorithm with time T, space S, and h sequential heads to the message. If x^* is good for interval i with interval size I and distance δ, then whatever code A outputs has distance at most δ if*

$$8ST < N(I-1).$$

Proof. The idea is just to consider the Y from Definition 9.6. For the x that come from Y, we have that B_i is unmarked rarely, so rarely in fact that multiple x must have the exact same states every time B_i is unmarked. Since the state includes the positions of the heads, they must write the same thing when the interval is uncovered. Since the state includes the number of bits written, they must be written at the same location when B_i is uncovered. Thus the distance between these messages is at most what is written when B_i is covered, which is at most a δ fraction of bits.

More rigorously, for any $y \in Y$, let $x_y = y \circ x^*$. By definition, A on message x_y marks B_i at most $\frac{8T}{N}$ times. Then define $R(y)$ to be the tuple of the state of A at every step B_i is uncovered when running on message x_y. Then since the state only has S bits and B_i is uncovered at most $\frac{8T}{N}$ times, we have

$$|R_y| \leq S\frac{8T}{N}.$$

Now see that since $|R(y)| \leq 8\frac{ST}{N}$, then the total number of distinct values for $R(y)$ is at most $2^{8\frac{ST}{N}}$. Since the restriction is good, the total number of distinct $y \in Y$ is at least $2^I/2$. Finally, by lemma premise, we have that $8ST < N(I-1)$. Thus we can show that

$$\begin{aligned} 8ST &< N(I-1) \\ 8\frac{ST}{N} &< I-1 \\ 2^{8S\frac{T}{N}} &< 2^I/2 \end{aligned}$$

Thus the number of distinct $R(y)$ is less than the number of distinct y, so by pigeonhole principle, there must be $y_1, y_2 \in Y$ such that $R(y_1) = R(y_2)$.

Now by definition of $R(y)$, when running A on message x_y we must have that everything written when interval B_i is uncovered is dependent only on $R(y)$ and x^*. In particular, $R(y_1)$ and $R(y_2)$ write the same bits when B_i is uncovered, at the same places.

Since $y_1, y_2 \in Y$, we have that at most δ fraction of the bits are written when B_i is covered, and these are the only bits where they can differ. So the distance between the outputs of A on x_{y_1} and x_{y_2} is at most δ. □

It remains to show that there must be some good restriction if I is chosen well. Specifically when $I < \frac{N\delta}{24h}$.

Lemma 9.8 (Good Restrictions Exist)**.** *Suppose A is an algorithm with time T, space S, and h sequential heads to access the message.*

Then if $I < \frac{N\delta}{24h} < N < T$ then there is some restriction which is good for some interval i with distance δ on algorithm A.

Proof. Suppose by way of contradiction that every restriction is not good for any interval with distance δ on algorithm A. The idea is to show that, in expectation, either more than $3h$ of intervals are covered whenever a bit is written (contradicting Lemma 9.3) or more than $1+T/I$ intervals are marked (contradicting Lemma 9.4).

So choose a random interval, i, a random restriction x^* for every variable outside B_i. We assumed that x^* is bad, so with probability at least 1/2 for a random assignment, y, of the variables in B_i will $y \notin Y$ for the Y defined in Definition 9.6. That is, with probability at least 1/2 will we have for $x_y = y \circ x^*$ that algorithm A on message x_y will unmark interval i more then $\frac{8T}{N}$ times *or* will be covered when at least δ fraction of the output bits written.

See that x_y is uniformly randomly distributed and so is i, and these are independent of each other. Thus in expectation over a randomly chosen interval, B_i, and a randomly chosen input, x, we have that with probability at least 1/2 algorithm A on input x either unmarks interval B_i more then $\frac{8T}{N}$ times *or* interval B_i be covered when at least a δ fraction of the output bits written. Now we show that x cannot have enough intervals covered or unmarked to achieve this.

By Lemma 9.4, the total number of markings is at most $T/I + 1 < 2T/I$. So at most $\frac{N}{4I}$ intervals are marked more than $\frac{8T}{N}$ times. There are at least N/I intervals, so only at most $\frac{1}{4}$ fraction of the intervals can be marked more than $\frac{8T}{N}$ times.

By Lemma 9.3, at any given time, at most $3h$ intervals can be covered. So the probability that a random interval is covered is at most

$$\begin{aligned} \frac{3h}{N/I} &= \frac{3hI}{N} \\ &< \frac{3hN\delta}{N24h} \\ &= \frac{\delta}{8}. \end{aligned}$$

Then by a Markov inequality, the probability that a random interval is covered greater than a δ fraction of the times an output bit is written is at most $\frac{1}{8}$.

Thus by a union bound, the total of fraction of intervals that are either unmarked more than $\frac{8T}{N}$ times or covered for at least δ fraction of the times an output bit is written is $\frac{3}{8} < \frac{1}{2}$. But by choice of x, these must occur for at least half i. Contradiction. So some restriction and interval must be good with distance δ. □

Now that we know good restrictions exist, and good restrictions imply our lower bounds, we can prove Theorem 1.2.

Theorem 1.2 (Lower Bounds For Encoders With Sequential Access)**.** *Suppose C is a code with relative distance δ encoding N bits. Suppose A is an algorithm computing C running in time T space S and using h sequential heads to access the message. Further assume $S > h\log(N)$. Then*

$$hST = \Omega(\delta N^2).$$

Proof. Let $I = \frac{N\delta}{50h} < \frac{N(\delta/2)}{24h}$. Then by Lemma 9.8, a good restriction with distance $\delta/2$ exists. Then by Lemma 9.7, since C has distance greater than $\delta/2$, it must be the case that $32ST \geq NI$. Thus

$$\begin{aligned} 8ST \geq& N(I-1) \\ 16ST \geq& NI \\ =& N\frac{N\delta}{50h} \\ 800STh \geq& N^2\delta \\ hST = \Omega(n^2\delta). \end{aligned}$$

□

9.2 Upper Bounds

The codes that achieve our upper bounds are just a tensor code. There is a natural way to compute tensor codes in a space efficient way when one has a limited number of sequential heads to access the message. By choosing one of the codes to be a code with time and space efficient encoders using random access to it's message, one can get a smooth trade off between the time and space required to encode a code.

Lemma 9.9 (Tensor Codes Are Efficient To Compute)**.** *Suppose there is a code $C_1 : \{0,1\}^{N_1} \to \Sigma_1^{M_1}$ with relative distance δ_1 computable in time T_1 and space S_1 where $N_1 \leq M_1$ and $\Sigma_1 = \{0,1\}^{\ell_1}$. Suppose there is another code $C_2 : \{0,1\}^{N_2} \to \Sigma_2^{M_2}$ with relative distance δ_2 computable in time T_2 and space S_2 with non-adaptive, random access to its message where $N_2 \leq M_2$ and $\Sigma_2 = \{0,1\}^{\ell_2}$.*

Denote the tensor code of C_1 and C_2 as $C : \{0,1\}^{N=N_1N_2} \to \Sigma^{M=M_1M_2}$, where $\Sigma = \{0,1\}^{\ell_1\ell_2}$. That is, C, is a tensor code on binary symbols, but we then group into the output bits into ℓ_1 by ℓ_2 squares. Then C has relative distance $\delta_1\delta_2$.

Further, for any number of heads $h \geq 2$, there is an algorithm running in time $O(T_2\left(\frac{N}{h} + T_1\right))$ and space $O(S_1 + S_2M_1\ell_1)$ using at most h non-reversible sequential heads to access the message that computes C.

Proof. In a tensor code, the message is arranged as a table, table 1, with N_2 rows, each containing N_1 columns. The tensor code is the code defined by first encoding each of the rows of table 1 with C_1 to get a new table, table 2, with N_2 rows and $M_1\ell_1$ columns. Then encode the columns of table 2 with C_2 to get table 3 with $M_2\ell_2$ rows and $M_1\ell_1$ columns. Finally, we group table 3 into ℓ_1 by ℓ_2 cells to get the final symbols of our final codeword. We assume the message is arranged by row, with all the first N_1 bits being row 1, the next N_1 bits being row 2, and so on.

Then our encoder first distributes $h-1$ heads evenly between the N_2 rows. Then we simulate C_2 in parallel for each of the $M_1\ell_1$ columns, and every time they need to query a bit from a row that has been encoded by C_1, we just move the nearest head to that row, encode it by C_1, and then give that row to each of the $M_1\ell_1$ parallel computations of C_2.

The time needed for this is just the time to encode C_2 the $M_1\ell \le T_1$ times, plus the number of rows that are queried (trivially bounded by T_1) times the time to move a head to that row (bounded by $\frac{N_2N_1}{h-1} = O(\frac{N}{h})$) and the time to encode that row T_1. This takes time at most

$$T_2M_1\ell + T_2(\frac{N_2N_1}{h-1} + T_1) = O(T_2(T_1 + \frac{N}{h})).$$

The space is just the space to compute C_1, plus the space to store the output of C_1, plus the space to hold $M_1\ell_1$ copies of S_2's algorithm. This is just space $O(S_1 + S_2M_1\ell_1)$. □

Now applying this tensor code with any good code and our explicit time space efficient codes shows that our lower bound on algorithms with sequential like access to the message is almost tight. We now prove Theorem 1.3.

Theorem 1.3 (Encoders With Sequential Access Meeting the Lower Bounds)**.** *For any number of heads $h \ge 2$, time $T \ge N$, space $S = \Omega(h\log(N))$, relative distance $\delta > 0$ with $hST = \Omega(\delta N^2)$, there exists a code with constant rate and relative distance $\Omega(\delta)$ encoded by a time $TN^{o(1)}$, space $S\,polylog(N)$ algorithm using h sequential heads to access the message.*

Proof. To make things simple, we assume that $4S$ divides δN and $N' = \delta N$ is an integer. If not, pad S and N and decrease δ until this is true. This can be done only changing constant factors, for instance by making each a power of 2.

Since we only need relative distance $\Omega(\delta)$, we just use an efficient, constant relative distance code on $1/\delta$ sets of δN message bits. Since all of our codes are linear, the min weight codeword must have weight $\Omega(\delta N)$, or relative weight $\Omega(\delta)$. Thus final code will have relative distance $\Omega(\delta)$.

Let our first code be the Spielman code $C_1 : \{0,1\}^S \to \{0,1\}^{4S}$, which has some constant relative distance, $\delta_1 > 0$, and is encodable in linear time and linear space. Then using Theorem 1.1, there exists a linear code

$$C_2 : \{0,1\}^{N'/4S} \to \{0,1\}^{M'}$$

that has some constant relative distance $\delta_2 > 0$, and output length $M' = O(N'/S) = O(\delta N/S)$. Note to get binary alphabet, we just output each bit of Σ individually. This hurts distance, but only by a constant factor. Further C is computable in time $\frac{N'}{S}\,poly(2^{\log(\log(N'))^3}) = \frac{\delta N}{S}N^{o(1)}$ and space $O(\log(N)^2)$.

Now applying Lemma 9.9 with C_1 and C_2, we get a tensor code

$$C : \{0,1\}^N \to \{0,1\}^{M'4S}$$

with constant relative distance $\delta_1\delta_2 > 0$ and output length $M'4S = O(\delta N)$. Further, C can be computed in time

$$O(\frac{\delta N}{S}N^{o(1)}\left(\frac{\delta N}{h} + S\right)) = \frac{\delta^2N^2}{Sh}N^{o(1)} + \delta NN^{o(1)}$$

and space

$$O(S + \log(N)^2S) = S\,polylog(N)$$

using only h sequential heads to access the message.

Now to compute the final code, we need only repeat this encoding procedure $1/\delta$ times, which uses the same space and same number of heads, but takes $1/\delta$ times longer to get final time of

$$\frac{\delta N^2}{Sh}N^{o(1)} + NN^{o(1)} = O(TN^{o(1)}).$$

□

9.3 Basic Relations of Sequential Access

In this section, we establish some relationships between variations of sequential access. We show whether reversibility or jumping adds power to time and space bounded algorithms with sequential access to the input.

While reversibility, being able to move heads backward, may seem powerful, it can actually be simulated with non-reversible, jumping heads with only a logarithmic factor overhead. We now prove Lemma 1.6.

Lemma 1.6 (Reversibility Can Be Efficiently Simulated With Jumping)**.** *A single sequential head to a length N input can be simulated with $O(\log(N))$ non-reversible sequential heads to the same input with an expected time of $O(\log(N))$ for each head movement, and $O(\log(N))$ space.*

More generally, k sequential heads to a length N input can be simulated with $O(k\log(N))$ non-reversible sequential heads to that same input with an expected time of $O(\log(N))$ for each head movement, and $O(k\log(N))$ space.

Proof. The idea is to store one to two non-reversible heads at every order of magnitude behind the reversible head being simulated. By being appropriately lazy with removing heads and adding them as the simulated head moves, this can be done with only $O(\log(N))$ heads and only $O(\log(N))$ time overhead. The extra $O(\log(N))$ bits of space are just to remember where the heads are.

More specifically, there are i levels and each responsible for its own order of magnitude. That order of magnitude only ever removes heads that are far from the simulated head, relative to its order of magnitude. So when a head is removed, the simulated head must use a lot of time to force it to be added again. This requires only storing at most two heads per order of magnitude.

Level i always stores heads that are 2^i distance apart, can either store 0, 1, or 2 heads, and these heads are always at the multiples of 2^i immediately behind the head being simulated. A new head is added to a level whenever the simulated head passes forward over a multiple of 2^i, and a head is removed if either the simulated head passes over it when moving backward, or when the level has more than 2 heads, in which case the first head is removed.

Now the trick is to move a simulated head left, instead of moving a head left (which is not allowed), we instead have the head jump to whichever level, i, has the nearest head behind it, and simulate it forward till it reaches one position before where it was before.

The space and number of heads is clear from the construction.

For time, we note that a left move can only trigger level i once every 2^{i-1} steps. To see this, first we claim that level i is only triggered when the simulated head is at a multiple of 2^{i-1} minus 1. This is because for every j, level j's earliest head is earlier than or equal to all of level $j-1$'s heads. This is straightforward from the construction as level $j-1$ removes heads before level j. Thus the last head that was passed before triggering level i must have been in level $i-1$.

After level i was triggered, there is a head in level $i-1$ at least $2^{i-1}-1$ before the simulated head. And we note that moving forward can never decrease the distance to the furthest head in level $i-1$ to below 2^{i-1}. Thus to trigger level i after a move would require 2^{i-1} moves.

Each level, i, takes only time $O(2^i)$ when it is triggered, and it is triggered at most once every $\frac{1}{2^{i-1}}$ steps. Thus each level only costs an expected constant time per simulated head movement. There are only $\log(N)$ levels. Thus the expected time per head movement is $O(\log(N))$.

If there are multiple heads being simulated, this actually just makes the problem easier as it increases our budget. When we move backward, we just use whichever head is closest from any of our simulated heads, and we don't remove any head at level i if it is within 2^{i+1} steps behind any simulated head. Jumps are cheap since we can share heads between the simulated head being jumped and the simulated head it is jumping to. In particular, you can't adversarially jump simulated heads to right before expensive backward operations as the resulting new heads will be shared with the simulated head that just jumped.

To make argument more formal in the multiple head case, we can count the time needed to trigger a particular head at a particular location, call it j, in level i. A head can only be triggered if there is a simulated head in front of it, and there is no head in a lower level at the same location. If the closest head in level $i-1$ was 2^{i-1} in front of j, then there must have been no simulated head within 2^i of j at some point, otherwise level $i-1$ would have kept a head at location j. Thus it would have taken 2^{i-1} steps to move a head backwards enough to trigger level i at location j. If the closest head in level $i-1$ was 2^i in front of j, there must have been no simulated head within $3 \cdot 2^{i-1}$ of j for a similar reason, and thus a simulated head must have moved 2^{i-1} times to be in position to trigger the head in level i at j. We call these backward movements of simulated heads within the interval $[j + 2^{i-1}, j + 3 \cdot 2^{i-1})$ as being dedicated to the head at j in level i.

So each position j at a level i takes at least 2^{i-1} backward movements dedicated to it specifically to trigger that head. From there, a similar argument holds as the single head case. □

Thus the reversible and non-reversible input are equivalent up to log factors. A similar phenomenon happens with reversible and non reversible computation. Bennett [Ben89] proved that any time T space S non-reversible computation can be turned into a reversible algorithm running in time $T^{1+\epsilon}$ and space $O(S \log(T))$ for any constant $\epsilon > 0$.

On the other hand, the jumping feature can not be efficiently simulated with non-jumping heads. It provably requires polynomially more time or space to solve some problems with non-jumping heads than jumping ones. One reason for this is that jumping can make it very efficient to move many heads far distances at once by moving one head, then jumping the others. The following result assumes our upper and lower bounds on codes in Theorem 1.2 and Theorem 1.3.

Lemma 9.10 (Jumping Can Not Be Efficiently Simulated With Reversibility)**.** *There exists a problem solvable in time $N^{1+o(1)}$ and space $N^{1/2+o(1)}$ with $O(\sqrt{N})$ jumping, non-reversible sequential heads to access the input, but for any constant ϵ can not be solved in time $o(N^{5/4-\epsilon})$ and space $o(N^{1/2+2\epsilon})$ with any number of non-jumping, reversible sequential heads to access the input.*

Proof. The problem is to just encode the second half of the input with the code of Theorem 1.3 using constant distance. This can be done easily with jumping, non-reversible heads in the time, space, and number of heads stated.

Now by Theorem 1.2, to encode this second half in time T and space S would require a number of heads that is at least

$$h \geq \Omega(\frac{N^2}{ST}).$$

But to actually use any of these heads, they must first pass the first half of the input. Since we have to move the heads one at a time, this takes time

$$\begin{aligned} T \geq& \Omega(hN) \\ \geq& \Omega(\frac{N^3}{ST}) \\ ST^2 \geq& \Omega(N^3) \end{aligned}$$

But suppose for way of contradiction that we had such a time $T = o(N^{5/4-\epsilon}$ and space $S = o(N^{1/2+2\epsilon})$ algorithm with non-jumping sequential access to the input. Then we have that

$$\begin{aligned} ST^2 \leq& o(N^{1/2+2\epsilon+2(5/4-\epsilon)}) \\ =& o(N^3) \end{aligned}$$

which is a contradiction. □

Taking $\epsilon = \frac{1}{8}$, we see that the algorithm with non-jumping heads gets both polynomially more time and polynomially more space than the algorithm with jumping heads, but still cannot compute the code.

Remark (Non-Jumping Heads With Preprocessing)**.** *The lower bounds of Lemma 9.10 depend on the fact that it takes a long time to perform initial positioning of heads. This lower bound no longer holds if we allow all the heads to be positioned in the second half of the input before the algorithm starts.*

But we can still get something even if we allow the algorithm to use any initial positioning of heads that does not depend on the input. The idea is essentially the same, encode some specific subset of the bits in a *code, but have the specific set of bits to be encoded be part of the input. We quickly sketch a proof outline here.*

To encode a random interval with $N^{3/4}$ fraction of message can be done by Theorem 1.3 with $h = S = N^{1/4}$ in time around $\frac{(N^{3/4})^2}{Sh} = N$ with sequential access to the input. Now consider an algorithm with non-jumping sequential access to the input with the same space and number of heads, with some initial, static positioning of the heads. With high probability, only a constant number of heads will be in the interval, so to encode it fast, our encoder needs to move more heads to that interval.

To move K heads to the interval to encode, in expectation, would take time around $\frac{NK}{h}K = N^{3/4}K^2$. Then by Theorem 1.2, to encode with these K heads would take time $\frac{(N^{3/4})^2}{SK} = \frac{N^{5/4}}{K}$. No matter the setting of K, this encoding will take time $\Omega(N^{13/12})$.

Thus jumping sequential access still has advantage over non-jumping sequential access even if a non-jumping algorithm is allowed to move heads into an initial position for free before it starts reading.

So we have proven that jumping heads are polynomially more powerful than non-jumping heads, and that jumping, non-reversible heads are within a log factor as powerful as jumping, reversible heads. Finally, one can show that non-jumping, reversible sequential heads are polynomially more powerful than non-jumping, non-reversible sequential heads. This is a direct consequence of Theorem 1.2.

Lemma 9.11 (Without Jumps, Reversible Heads Are More PowerFul than Non-Reversible Heads)**.** *Encoding codes with relative distance δ with non-jumping, non-reversible heads requires space S and number of heads h such that*

$$h^2 S = \Omega(\delta N).$$

Proof. This follows from the fact that Theorem 1.2 is actually a bound on the number of head movements. Since heads without backwards movements or jumps cannot move backward, there are only hN movements before all input heads are at the end of input. Thus $hN \geq T$ in Theorem 1.2. This gives that

$$h^2 S \geq \frac{hST}{N} = \Omega(\delta N).$$

□

In particular, without jumps or backwards moves, one cannot compute an asymptotically good code with only $S = n^{1/3}$ space and only $h = o(n^{1/3})$ heads, whereas with either jumping or reversibility[2], this can be achieved by using $N^{4/3+o(1)}$ time by Theorem 1.3.

Acknowledgments

Thanks to Ryan Williams for questions that eventually led to this result. Thanks to David Zuckerman, Justin Oh, and Jesse Goodman for some advice about which extractors might be used in our condenser construction. Thanks to Niels Kornerup for conversations about time space lower bounds.

References

[Abr91] Karl Abrahamson. "Time-space tradeoffs for algebraic problems on general sequential machines". In: *Journal of Computer and System Sciences* 43.2 (1991), pp. 269–289. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(91)90014-V. URL: https://www.sciencedirect.com/science/article/pii/002200009190014V.

[Alo+92] N. Alon, J. Bruck, J. Naor, M. Naor, and R.M. Roth. "Construction of asymptotically good low-rate error-correcting codes through pseudo-random graphs". In: *IEEE Transactions on Information Theory* 38.2 (1992), pp. 509–516. DOI: 10.1109/18.119713.

[2]This uses the fact that the only time jumping is used in Theorem 1.3 is for efficient, initial positioning. Initial positioning of h heads only takes $O(hN)$ time without jumping.

[BC82] A. Borodin and S. Cook. "A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation". In: *SIAM J. Comput.* 11.2 (1982), 287–297. ISSN: 0097-5397. DOI: 10.1137/0211022. URL: https://doi.org/10.1137/0211022.

[BK23] Paul Beame and Niels Kornerup. "Cumulative Memory Lower Bounds for Randomized and Quantum Computation". In: *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023).* Ed. by Kousha Etessami, Uriel Feige, and Gabriele Puppis. Vol. 261. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 17:1–17:20. ISBN: 978-3-95977-278-5. DOI: 10.4230/LIPIcs.ICALP.2023.17. URL: https://drops.dagstuhl.de/opus/volltexte/2023/18069.

[BM05] L.M.J. Bazzi and S.K. Mitter. "Endcoding complexity versus minimum distance". In: *IEEE Transactions on Information Theory* 51.6 (2005), pp. 2103–2112. DOI: 10.1109/TIT.2005.847727.

[BS+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. "On the Concrete Efficiency of Probabilistically-Checkable Proofs". In: *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing.* STOC '13. Palo Alto, California, USA: Association for Computing Machinery, 2013, 585–594. ISBN: 9781450320290. DOI: 10.1145/2488608.2488681. URL: https://doi.org/10.1145/2488608.2488681.

[Bea89] P. Beame. "A General Sequential Time-Space Tradeoff for Finding Unique Elements". In: STOC '89. Seattle, Washington, USA: Association for Computing Machinery, 1989, 197–203. ISBN: 0897913078. DOI: 10.1145/73007.73026. URL: https://doi.org/10.1145/73007.73026.

[Ben89] Charles H. Bennett. "Time/Space Trade-Offs for Reversible Computation". In: *SIAM J. Comput.* 18.4 (1989), 766–776. ISSN: 0097-5397. DOI: 10.1137/0218053. URL: https://doi.org/10.1137/0218053.

[CDS12] M. Cheraghchi, F. Didier, and A. Shokrollahi. "Invertible Extractors and Wiretap Protocols". In: *IEEE Trans. Inf. Theor.* 58.2 (2012), 1254–1274. ISSN: 0018-9448. DOI: 10.1109/TIT.2011.2170660. URL: https://doi.org/10.1109/TIT.2011.2170660.

[CG17] Mahdi Cheraghchi and Venkatesan Guruswami. "Non-malleable Coding Against Bit-Wise and Split-State Tampering". In: *J. Cryptol.* 30.1 (2017), 191–241. ISSN: 0933-2790. DOI: 10.1007/s00145-015-9219-z. URL: https://doi.org/10.1007/s00145-015-9219-z.

[CGL16] Eshan Chattopadhyay, Vipul Goyal, and Xin Li. "Non-malleable extractors and codes, with their many tampered extensions". In: *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing.* STOC '16. Cambridge, MA, USA: Association for Computing Machinery, 2016, 285–298. ISBN: 9781450341325. DOI: 10.1145/2897518.2897547. URL: https://doi.org/10.1145/2897518.2897547.

[Cap+02] M. Capalbo, O. Reingold, S. Vadhan, and A. Wigderson. "Randomness conductors and constant-degree lossless expanders". In: *Proceedings 17th IEEE Annual Conference on Computational Complexity.* 2002, pp. 8–8. DOI: 10.1109/CCC.2002.1004327.

[DDS23] Yotam Dikstein, Irit Dinur, and Shiri Sivan. *The linear time encoding scheme fails to encode.* 2023. arXiv: 2312.16125 [cs.CC].

[DJM98] Dariush Divsalar, Hui Jin, and Robert J. McEliece. "Coding theorems for 'turbo-like' codes". In: *Proceedings 36th Annual Allerton Conference on Communication, Control, and Computing.* Monticello, IL, USA, 1998, pp. 201–210. URL: https://api.semanticscholar.org/CorpusID:1045655.

[Din+22] Irit Dinur, Shai Evra, Ron Livne, Alexander Lubotzky, and Shahar Mozes. "Locally Testable Codes with Constant Rate, Distance, and Locality". In: *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing.* STOC 2022. Rome, Italy: Association for Computing Machinery, 2022, 357–374. ISBN: 9781450392648. DOI: 10.1145/3519935.3520024. URL: https://doi.org/10.1145/3519935.3520024.

[GI05] V. Guruswami and P. Indyk. "Linear-time encodable/decodable codes with near-optimal rate". In: *IEEE Transactions on Information Theory* 51.10 (2005), pp. 3393–3400. DOI: 10.1109/TIT.2005.855587.

[GM08] Venkatesan Guruswami and Widad Machmouchi. "Explicit interleavers for a Repeat Accumulate Accumulate (RAA) code construction". In: *2008 IEEE International Symposium on Information Theory.* 2008, pp. 1968–1972. DOI: 10.1109/ISIT.2008.4595333.

[GUV07] Venkatesan Guruswami, Christopher Umans, and Salil Vadhan. "Unbalanced Expanders and Randomness Extractors from Parvaresh-Vardy Codes". In: *Twenty-Second Annual IEEE Conference on Computational Complexity (CCC'07).* 2007, pp. 96–108. DOI: 10.1109/CCC.2007.38.

[Gal62] R. Gallager. "Low-density parity-check codes". In: *IRE Transactions on Information Theory* 8.1 (1962), pp. 21–28. DOI: 10.1109/TIT.1962.1057683.

[Gro06] André Gronemeier. "A Note on the Decoding Complexity of Error-Correcting Codes". In: *Inf. Process. Lett.* 100.3 (2006), 116–119. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2006.06.006. URL: https://doi.org/10.1016/j.ipl.2006.06.006.

[Gá+13] Anna Gál, Kristoffer Arnsfelt Hansen, Michal Koucký, Pavel Pudlák, and Emanuele Viola. "Tight Bounds on Computing Error-Correcting Codes by Bounded-Depth Circuits With Arbitrary Gates". In: *IEEE Transactions on Information Theory* 59.10 (2013), pp. 6611–6627. DOI: 10.1109/TIT.2013.2270275.

[HR03] Tzvika Hartman and Ran Raz. "On the Distribution of the Number of Roots of Polynomials and Explicit Weak Designs". In: 23.3 (2003), 235–263. ISSN: 1042-9832. DOI: 10.1002/rsa.10095. URL: https://doi.org/10.1002/rsa.10095.

[Ham50] R. W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[Hen65] F.C. Hennie. "One-tape, off-line Turing machine computations". In: *Information and Control* 8.6 (1965), pp. 553–578. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(65)90399-2. URL: https://www.sciencedirect.com/science/article/pii/S0019995865903992.

[ILL89] R. Impagliazzo, L. A. Levin, and M. Luby. "Pseudo-Random Generation from One-Way Functions". In: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing.* STOC '89. Seattle, Washington, USA: Association for Computing Machinery, 1989, 12–24. ISBN: 0897913078. DOI: 10.1145/73007.73009. URL: https://doi.org/10.1145/73007.73009.

[Juk09] S. Jukna. "A nondeterministic space-time tradeoff for linear codes". In: *Information Processing Letters* 109.5 (2009), pp. 286–289. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2008.11.001. URL: https://www.sciencedirect.com/science/article/pii/S0020019008003347.

[KSY14] Swastik Kopparty, Shubhangi Saraf, and Sergey Yekhanin. "High-Rate Codes with Sublinear-Time Decoding". In: *J. ACM* 61.5 (2014). ISSN: 0004-5411. DOI: 10.1145/2629416. URL: https://doi.org/10.1145/2629416.

[KTS22] Itay Kalev and Amnon Ta-Shma. "Unbalanced Expanders from Multiplicity Codes". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2022).* Ed. by Amit Chakrabarti and Chaitanya Swamy. Vol. 245. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 12:1–12:14. ISBN: 978-3-95977-249-5. DOI: 10.4230/LIPIcs.APPROX/RANDOM.2022.12. URL: https://drops.dagstuhl.de/opus/volltexte/2022/17134.

[Kop+16] Swastik Kopparty, Or Meir, Noga Ron-Zewi, and Shubhangi Saraf. "High-Rate Locally-Correctable and Locally-Testable Codes with Sub-Polynomial Query Complexity". In: *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing.* STOC '16. Cambridge, MA, USA: Association for Computing Machinery, 2016, 202–215. ISBN: 9781450341325. DOI: 10.1145/2897518.2897523. URL: https://doi.org/10.1145/2897518.2897523.

[Kos79] S. Rao Kosaraju. "Real-Time Simulation of Concatenable Double-Ended Queues by Double-Ended Queues (Preliminary Version)". In: *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing.* STOC '79. Atlanta, Georgia, USA: Association for Computing Machinery, 1979, 346–351. ISBN: 9781450374385. DOI: 10.1145/800135.804427. URL: https://doi.org/10.1145/800135.804427.

[Li17] Xin Li. "Improved Non-Malleable Extractors, Non-Malleable Codes and Independent Source Extractors". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing.* STOC 2017. Montreal, Canada: Association for Computing Machinery, 2017, 1144–1156. ISBN: 9781450345286. DOI: 10.1145/3055399.3055486. URL: https://doi.org/10.1145/3055399.3055486.

[MN96] D. J. C. Mackay and Radford M. Neal. "Near Shannon limit performance of low density parity check codes". In: *Electronics Letters* 33 (1996), pp. 457–458. URL: https://api.semanticscholar.org/CorpusID:122801915.

[NZ96] Noam Nisan and David Zuckerman. "Randomness is Linear in Space". In: *J. Comput. Syst. Sci.* 52.1 (1996), 43–52. ISSN: 0022-0000. DOI: 10.1006/jcss.1996.0004. URL: https://doi.org/10.1006/jcss.1996.0004.

[PSS80] Wolfgang J. Paul, Joel I. Seiferas, and Janos Simon. "An Information-Theoretic Approach to Time Bounds for on-Line Computation (Preliminary Version)". In: *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing.* STOC '80. Los Angeles, California, USA: Association for Computing Machinery, 1980, 357–367. ISBN: 0897910176. DOI: 10.1145/800141.804685. URL: https://doi.org/10.1145/800141.804685.

[RRV99] Ran Raz, Omer Reingold, and Salil Vadhan. "Extracting All the Randomness and Reducing the Error in Trevisan's Extractors". In: *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing.* STOC '99. Atlanta, Georgia, USA: Association for Computing Machinery, 1999, 149–158. ISBN: 1581130678. DOI: 10.1145/301250.301292. URL: https://doi.org/10.1145/301250.301292.

[RSW00] O. Reingold, R. Shaltiel, and A. Wigderson. "Extracting randomness via repeated condensing". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science.* 2000, pp. 22–31. DOI: 10.1109/SFCS.2000.892008.

[Ree00] I.S. Reed. "A brief history of the development of error correcting codes". In: *Computers & Mathematics with Applications* 39.11 (2000), pp. 89–93. ISSN: 0898-1221. DOI: https://doi.org/10.1016/S0898-1221(00)00112-7. URL: https://www.sciencedirect.com/science/article/pii/S0898122100001127.

[SS96] M. Sipser and D.A. Spielman. "Expander codes". In: *IEEE Transactions on Information Theory* 42.6 (1996), pp. 1710–1722. DOI: 10.1109/18.556667.

[SV06] Nandakishore Santhi and Alexander Vardy. "Minimum Distance of Codes and Their Branching Program Complexity". In: *2006 IEEE International Symposium on Information Theory.* 2006, pp. 1490–1494. DOI: 10.1109/ISIT.2006.262116.

[SV77] Walter J. Savitch and Paul M. B. Vitányi. "Linear Time Simulation of Multihead Turing Machines with Head-to-Head Jumps". In: *Proceedings of the Fourth Colloquium on Automata, Languages and Programming.* Berlin, Heidelberg: Springer-Verlag, 1977, 453–464. ISBN: 3540083421.

[Spi95] Daniel A. Spielman. "Linear-Time Encodable and Decodable Error-Correcting Codes". In: *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing.* STOC '95. Las Vegas, Nevada, USA: Association for Computing Machinery, 1995, 388–397. ISBN: 0897917189. DOI: 10.1145/225058.225165. URL: https://doi.org/10.1145/225058.225165.

[Spi96] D.A. Spielman. "Linear-time encodable and decodable error-correcting codes". In: *IEEE Transactions on Information Theory* 42.6 (1996), pp. 1723–1731. DOI: 10.1109/18.556668.

[TSUZ01] Amnon Ta-Shma, Christopher Umans, and David Zuckerman. "Loss-Less Condensers, Unbalanced Expanders, and Extractors". In: *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing.* STOC '01. Hersonissos, Greece: Association for Computing Machinery, 2001, 143–152. ISBN: 1581133499. DOI: 10.1145/380752.380790. URL: https://doi.org/10.1145/380752.380790.

[Tre99] Luca Trevisan. "Construction of Extractors Using Pseudo-Random Generators (Extended Abstract)". In: *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing.* STOC '99. Atlanta, Georgia, USA: Association for Computing Machinery, 1999, 141–148. ISBN: 1581130678. DOI: 10.1145/301250.301289. URL: https://doi.org/10.1145/301250.301289.

[Yes84] Yaacov Yesha. “Time-space tradeoffs for matrix multiplication and the discrete fourier transform on any general sequential random-access computer”. In: *Journal of Computer and System Sciences* 29.2 (1984), pp. 183–197. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(84)90029-1`. URL: `https://www.sciencedirect.com/science/article/pii/0022000084900291`.