Lecture 17

Lecturer: Madhu Sudan *Scribe: Jon Feldman*

Today's lecture is on

- Linear Time Encoding and Decoding: Spielman codes

One of the main drawbacks of the Sipser/Spielman codes we saw last time is the encoder's running time, which is quadratic in the code word length n. Today we cover Spielman codes, which feature linear-time encoding and decoding. Spielman codes are also based on expanders, and require constructing expanders with very extreme expansion propeties; however, very recent work may lead to expanders that are actually useful for these codes.

In Sipser/Spielman codes, the expander is the bipartite graph representing the parity check matrix, where the left side represents to the bits of the code word, and the right side represents the parity constraints on the code word bits. Spielman applies an expander to the graph where the left side represents the *message* bits, and the right side are parity checks on the message that are sent along with the message as code bits.

Definition 1 (Code R_k) *For a bipartite graph $G = (L, R, E)$, $|L| = k$, $|R| = n - k$, R_k is the code defined as follows. A code word of R_k is an n-bit assignment to the nodes $L \cup R$, such that the assignment to every node of R is equal to the parity of its neighbors.*

To make encoding and decoding efficient, we take G to be a (c, d)-bounded expander (same definition as last lecture). Taken on its own, this code is a rate k/n systematic code (it is systematic because the bits assigned to L are sent as part of the codeword). Despite its expansion, the distance properties of R_k are not very good; consider the messages 0^k and 10^{k-1}. The encoding of 0^k is 0^n, and the encoding of 10^{k-1} has at most $c + 1$ ones (the systematic bit, plus the c parity checks that include the first bit). Therefore the distance is at most $c + 1$. So, to make this construction useful, we need to work a bit harder.

When is this code useful? Suppose that the channel was very nice to us and only flipped some small fraction of the message bits without touching the check bits. Then, we could apply the "flip algorithm" from last lecture, and correct all the message bits. If the channel was slightly meaner, it would flip message bits and check bits, but flip a number of each that is proportional to thier respective sizes in the code word. This motivates an augmentation to our code; suppose we applied another code to the check bits in R to increase their reliability?

So why not recursively apply R_{n-k} to the right side of G ? These codes are called "Cascade codes." For example, Let G be the graph with $\log k$ layers of nodes $(P_k, P_{k/2}, \ldots, P_1)$, where P_i contains i nodes. Layer $P_{i/2}$ is connected to P_i via the edges from R_i. A code word is an assignment to the $2k - 1$ nodes of G where the assignment to some node of $P_{i/2}$ equals the parity of its neighbors in P_i.

The bits from layers on the end of G are parities of parities of parities, etc. However, to put things in perspective, parities of parities are just parities, so the entire code could be modeled as a single bipartite message/check-bit graph. However, this graph would no longer have constant degree. If we view the code as a layered graph, then encoding is easily seen to be linear time; we just compute the parities one layer at a time. The running time is proportional to the number of edges, which is $O(n)$.

1 Encoding

It is very hard (if not impossible) to prove that the cascade code can correct a constant fraction of errors, which is our goal. So, we need another gadget to improve the situation. First, some definitions. Let R_k be defined as above, with rate $1/2$, and an (c, d)-bounded expander. Let $R_k(m)$ be the encoding of message m under R_k. The code word $R_k(m)$ is composed of k message bits and $k/2$ check bits. Now we will construct a new code C_k by defining the encoding operation $C_k(m)$. For some message m, let

$C_k(m) = m \circ c_1 \circ c_2 \circ c_3$ (the operator $\circ$ is just concatenation), where the check bits c_1, c_2, c_3 are defined as follows:

- c_1 = check bits of $R_k(m)$
- c_2 = check bits of $C_{k/2}(c_1)$
- c_3 = check bits of $R_{2k}(c_1 \circ c_2)$

Rate. We show by induction that the check bits of $C_k(m)$ have total size $3k$, and thus C_k has rate $1/4$. The first part c_1 has $k/2$ bits, since R_k has $k/2$ parity bits. By induction, c_2 has size $3k/2$. The size of c_3 is k, for a total of $3k$.

Encoding Running Time. Encoding C_k takes linear time. Our induction hypothesis is that it takes $3dk$ time to encode the check bits of C_k. Consider each step individually. Computing c_1 involves computing $R_k(m)$, which just involves $k/2$ parity computations of d bits each, for a total of $d(k/2)$ time. By induction, we compute c_2 in $(3dk/2)$ time. Computing c_3 takes dk time, for a total of $3dk$ time, proving the induction hypothesis.

2 Decoding

Error Reduction. A basic operation in our decoding process will be "error reducing" the code R_k. We've said that decoding message bits is easy once we have perfect parity check bits. In fact, even if the parity check bits have some small number of errors, we will still be able to reduce the number of errors in the message bits. We formalize this notion below:

Definition 2 **((ϵ, γ)-error-reduction algorithm)** *Let (m, c) be a valid code word for R_k, and let (x, y) be such that:*

- $\Delta(m, x) \leq \gamma n$
- $\Delta(c, y) \leq \gamma n$.

An algorithm $A(x, y)$ that outputs x' as a decoded message is an (ϵ, γ)-error-reduction algorithm if $\Delta(m, x') \leq \epsilon \Delta(c, y)$.

Suppose that we have an $(1/2, \gamma)$-error-reduction algorithm A. We will prove the existence of such an algorithm later. To decode C_k, we are given $x \circ y_1 \circ y_2 \circ y_3$, which is the corrupted version of $m \circ c_1 \circ c_2 \circ c_3$, and we would like to output an $x' = m$. We assume that at most γk errors are caused by the channel, i.e., $\Delta(x \circ y_1 \circ y_2 \circ y_3, m \circ c_1 \circ c_2 \circ c_3) \leq \gamma k$. We define $D(x \circ y_1 \circ y_2 \circ y_3)$ to return x' as follows:

1. Apply $A(y_1 \circ y_2, y_3)$ to obtain $y_1' \circ y_2'$.
2. Apply $D(y_1' \circ y_2')$ to obtain y_1''.
3. Apply $A(x, y_1'')$ to obtain x'. Return x'.

Running Time of D. We assume A runs in linear time. Then, the running time of the above algorithm is the solution to the recurrance $T(k) = O(k) + T(k/2) + O(k)$, which solves to $T(n) = O(n)$.

Error Correcting Capability of D. Given our black box for A, showing that D decodes a constant fraction of errors is straightforward:

Theorem 3 *If A is a $(1/2, \gamma)$-error-reduction algorithm, then D corrects a $(\gamma/4)$ fraction of errors.*

Proof. Suppose $\delta(x \circ y_1 \circ y_2 \circ y_3, m \circ c_1 \circ c_2 \circ c_3) \leq (\gamma/4)n = \gamma k$. Then $\Delta(x, m), \Delta(y_1 \circ y_2, c_1 \circ c_2), \Delta(y_3, c_3)$ are all $\leq \gamma k$. This meets the preconditions of the error reduction capabilities of $A(y_1 \circ y_2, y_3)$. Thus, after we apply step (1), $\Delta(y_1' \circ y_2', c_1 \circ c_2) \leq (\gamma/2)k$. By induction, step (2) corrects all the errors of its message, i.e., $y_1'' = c_1$. Now the preconditions of $A(x, y_1'')$ are met, so $\Delta(x', m) \leq (1/2)\Delta(c_1, y_1'')$. Since $y_1'' = c_1$, $x' = m$.

3 Constructing an Error Reduction Algorithm

Thus it remains to show that we can construct a $(1/2, \gamma)$-error-reduction algorithm for R_k that runs in linear time. The algorithm is the same as the Sipser/Spielman "Flip Algorithm" from last lecture. We let $G = (L, R, E)$ be a bipartite graph, where L represent message bits, and R represents parity bits. Given (x, y), the algorithm maintains a bit assignment x' on the nodes of L (initialized to x), and assigns the nodes of R to y. The bit assignments to R don't change, but the ones on L get flipped by the algorithm. A node of R is considered *satisfied* if the parity of its neighbors is equal to its value; for example, if a node in R has a 1 on it, then its neighbors must contain an odd number of 1's for it to be satisfied. The algorithm A proceeds as follows:

1. While $\exists$ node i in L with more unsatisfied neighbors than satisfied ones, flip x_i'.
2. If no such node exists, output x'.

Running time of A. The algorithm A runs in linear time if all nodes have constant degree. This is because each parity check takes constant time, and each time a value of x_i is flipped, the number of unsatisfied neighbors decreases by at least one. Data structure tricks ensure that the node can be found quickly.

Error Reduction Capability of A. We will use an extension of the proof from last lecture to show that A can indeed be made into an error reduction algorithm given a good enough expander. Now for the main theorem:

Theorem 4 *If G is a $(c, 2c)$-regular $(\frac{7}{8}c, \delta)$-expander for some $\delta > 0$, then the flip algorithm is an (ϵ, γ)-error reduction algorithm for $\epsilon = \frac{4}{c}$ and $\gamma = \frac{c\delta}{2(c+2)}$.*

Proof. Fix x', y, m, c, and let:

- $S' = \{i \in L \mid x_i \neq m_i\}$, the nodes of L that are assigned incorrect bits.
- $T = \{j \in R \mid y_j \neq c_j\}$, the nodes of R that are assigned incorrect bits.
- $U = \{j \in R \mid j \text{unsatisfied}\}$, the unsatisfied nodes of R.
- $A = \{j \in R \mid j \text{hasoneneighborin } S\}$.
- $B = \{j \in R \mid j \text{hasatleasttwoneighborsin } S\}$.

We have $A - T \subseteq U$, since all nodes of A that have correct parity bits must be unsatisfied. We also have $U \subseteq A \cup T$ [why?]. The key lemma, proven last time, is:

$$0 < |S'| \leq \delta k \Rightarrow u > (2\alpha - c)|S'| - |T|$$

We combine this with the following two lemmas:

Lemma 5 $|S'| \leq \delta k$

Lemma 6 *$s > \frac{4t}{c}$ implies $\exists j \in S$ with more than $c/2$ neighbors in U*

The theorem follows.