ML decoding via mixed-integer adaptive linear programming

Stark C. Draper
Mitsubishi Electric Research Labs
Cambridge, MA 02139 USA
draper@merl.com

Jonathan S. Yedidia
Mitsubishi Electric Research Labs
Cambridge, MA 02139 USA
yedidia@merl.com

Yige Wang
Dept. of EE, Univ. of Hawaii at Manoa
Honolulu, HI 96822 USA
yige@spectra.eng.hawaii.edu

***Abstract*— Linear programming (LP) decoding was introduced by Feldman et al. (*IEEE Trans. Inform. Theory* Mar. 2005) as a novel way to decode binary low-density parity-check codes. Taghavi and Siegel (*Proc. ISIT* 2006) describe a computationally simplified decoding approach they term "adaptive" LP decoding. Adaptive LP decoding starts with a sub-set of the LP constraints, and iteratively adds violated constraints until an optimum of the original LP is found. Usually only a tiny fraction of the original constraints need to be reinstated, leading to huge efficiency gains compared to ordinary LP decoding.**

Here we describe a modification of the adaptive LP decoder that results in a maximum likelihood (ML) decoder. Whenever the adaptive LP decoder returns a pseudo-codeword rather than a codeword, we add an integer constraint on the least certain symbol of the pseudo-codeword. For certain codes, and especially in the high-SNR (error floor) regime, only a few integer constraints are required to force the resultant mixed-integer LP to the ML solution. We demonstrate that our approach can efficiently achieve the optimal ML decoding performance on a (155,64) LDPC code introduced by Tanner et al.

I. Introduction

Standard decoders for low-density parity-check (LDPC) codes are based on iterative belief-propagation (BP) decoding. In [4] Feldman et al. introduce an alternate decoding algorithm based on linear programing (LP). LP decoding has some attractive features. An LP decoder deterministically converges and when it converges to an integer solution, one knows that the maximum likelihood (ML) codeword has been found. When it converges to a non-integer solution, a well-defined "pseudo-codeword" has been found. Unfortunately, LP decoding is more complex than BP decoding. In the formulations originally introduced by Feldman et al., the number of constraints in the LP decoding problem is linear in block-length, but exponential in the maximum check node degree. The resulting computational load can be prohibitive.

The computationally burden of LP decoding motivates the introduction of "adaptive" linear programming (ALP) decoding by Taghavi and Siegel in [5]. Instead of starting with the full gamut of LP constraints, Taghavi and Siegel instead first solve for the LP optimum of a problem where the values of the binary codeword symbols are only constrained to be greater than zero or less than one. At the resulting optimum, for each check node, they (efficiently) determine the local constraints that are violated. Adding the violated constraints back into the problem they then solve the resulting LP. They iterate this process until no local constraints are violated. The result is guaranteed to match the solution to the original LP, even though only a small fraction of the original constraints are used. Empirically they observe that the number of constraints used for linear codes is only a small multiple (1.1-1.5) of the number m of parity-check constraints, even for codes with high check degree, and that LP decoding becomes significantly (sometimes several orders of magnitude) more efficient.

When the solution to the original LP is non-integer, the ML codeword has not been found, and one is motivated to find a tightening of the original LP relaxation. This is the focus of the current paper. In contrast to earlier proposals that focused on, e.g., adding redundant parity-checks (RPCs) [4], [5], or "lift and project" [3], we instead add a small number of integer constraints to the decoding problem. Our methodology for choosing each binary constraint is simple, we put integer constraints on the "least reliable" symbol of the optimum pseudo-codeword after each ALP decoding. We observe that in very many cases only a few integer constraints are needed to force the decoder to the ML solution. Keeping the integer constraints in the LP is also considered in [4], but there all n integer constraints are enforced, rather than being added in incrementally, in the style of ALP decoding.

Another recent work that enforces integer constraints on the least certain bits of an initial LP solution is [8]. While our basic philosophy is very similar to [8], by efficiently integrating the addition of integer constraints into an ALP decoder, we are able to extend the useful range of ML decoding to much longer block lengths.

Our approach is also reminiscent of the "augmented BP" approach introduced by Varnica et al. [7]. However, instead of starting with BP, and progressively fixing the least certain bits as in [7], we start with adaptive LP as our base algorithm. As a consequence, our algorithm, in contrast with augmented BP, results in a decoder which provably gives ML (optimal) performance.

The rest of the paper is organized as follows. In Section II we describe LP decoding and the adaptive LP decoding algorithm. In Section III we describe our modification to the ALP decoding algorithm. We give numerical results for a (155,64) code introduced by Tanner et al. [6] in Section IV, and conclude in Section V.

II. LP and adaptive LP decoding

Consider a binary length-n linear code $\mathcal{C}$. A codeword $\mathbf{x} \in \mathcal{C}$ is transmitted over a binary-input memoryless symmetric channel and the destination observes $\mathbf{y}$. The probability that any particular $\hat{\mathbf{x}} \in \mathcal{C}$ was sent given $\mathbf{y}$ is received is $\Pr[\hat{\mathbf{x}}|\mathbf{y}]$. Assuming that codewords are equally likely the ML decoding problem is

$$\arg\max_{\hat{\mathbf{x}}\in\mathcal{C}} \Pr[\mathbf{y}|\hat{\mathbf{x}}] = \arg\max_{\hat{\mathbf{x}}\in\mathcal{C}} \sum_{i=1}^{n} \log \Pr[y_i|\hat{x}_i] \quad (1)$$

$$= \arg\max_{\hat{\mathbf{x}}\in\mathcal{C}} \sum_{i=1}^{n} \log \frac{\Pr[y_i|x=1]}{\Pr[y_i|x=0]}\hat{x}_i + \log \Pr[y_i|x=0], \quad (2)$$

where $\hat{x}_i$ is the ith symbol of the candidate codeword $\hat{\mathbf{x}}$, and y_i is the ith symbol of the received sequence $\mathbf{y}$. Since the second term in (2) is common to all $\hat{\mathbf{x}}$ the ML decoding problem is equivalent to the following linear program

$$\text{minimize} \quad \gamma^T \hat{\mathbf{x}} \quad (3)$$

$$\text{subject to} \quad \hat{\mathbf{x}} \in \mathcal{C} \quad (4)$$

where γ is the known vector of negative log-likelihoods defined whose ith entry is defined as

$$\gamma_i = \log\left(\frac{\Pr[y_i|x=0]}{\Pr[y_i|x=1]}\right). \quad (5)$$

When the channel is binary-symmetric (BSC), $\gamma_i = \log[p/(1-p)]$ if the received bit $y_i = 1$, and $\gamma_i = \log[(1-p)/p]$ if the received bit $y_i = 0$.

The constraints in (4) are integer. In [4] a relaxed version of the problem is proposed. Each parity check describes a number of local linear constraints that the codewords must satisfy. The intersection of these constraints defines the polytope over which the LP solver operates. The integer vertices of the polytope correspond to codeword in $\mathcal{C}$. When the LP optimum is at such a vertex, (4) is satisfied and the ML solution is found. Non-integer solutions are termed pseudo-codewords.

The explicit description of the LP polytope used in [5] consists of first enforcing

$$0 \le \hat{x}_i \le 1 \text{ for all } i \in \{1, 2, \ldots n\}. \quad (6)$$

Then, for every check $j \in \{1, 2, \ldots, m\}$ every configuration of the set of neighboring variables $\mathcal{N}(j) \subset \{1, 2, \ldots n\}$ must satisfy the following parity-check constraint: for all $j \in \{1, 2, \ldots, m\}$, for all subsets $\mathcal{V} \subseteq \mathcal{N}(j)$ such that $|\mathcal{V}|$ is odd,

$$\sum_{i\in\mathcal{V}} \hat{x}_i - \sum_{i\in\mathcal{N}(j)/\ \mathcal{V}} \hat{x}_i \le |\mathcal{V}| - 1. \quad (7)$$

Working with this relaxation Taghavi and Siegel [5] define a cut as a violated constraint and an active constraint as a constraint satisfied with equality. For example a constraint such as (7) that generates a cut would mean that

$$\sum_{i\in\mathcal{V}} \hat{x}_i - \sum_{i\in\mathcal{N}(j)/\ \mathcal{V}} \hat{x}_i > |\mathcal{V}| - 1. \quad (8)$$

In deriving the ALP decoding algorithm two important properties of cuts are shown in [5]. First, at any given point $\hat{\mathbf{x}} \in [0, 1]^n$ for any check j at most one constraint (7) can be a cut. Second, (8) implies that $0 < \sum_{i\in\mathcal{N}(j)/\ \mathcal{V}} \hat{x}_i < \hat{x}_j$ for all $j \in \mathcal{V}$. This means that the variable nodes in the cut are those with the largest values in $\mathcal{N}(j)$. This in turn makes it easy to search for the cut of each check node (or to determine if one does not exist).

The iterative ALP decoding algorithm starts from the solution of a vertex of an easily-defined initial problem, and iteratively adds cut constraints and re-solves until no constraints are violated. Initial constraints are placed on each of the n variables as

$$\begin{array}{ll} 0 \le \hat{x}_i & \text{if} \quad \gamma_i > 0, \\ \hat{x}_i \le 1 & \text{if} \quad \gamma_i < 0. \end{array} \quad (9)$$

The LP optimum solution of this initial problem is immediately found by hard-decision decoding. The general algorithm (Algorithm 2 in [5]) is as follows:

1) Setup the initial problem according to (9).
2) Run the LP solver.
3) Search for all cuts of the current solution (see [5] to see how to do this efficiently using the Taghavi and Siegel's Algorithm 1).
4) If one or more cuts are found, add them to the problem constraints and go to step 2.

III. ML via mixed integer adaptive LP decoding

LP decoding fails to "pseudo-codewords" that are particularly easily defined–they are the non-integer optimum vertices of the feasible polytope. Since the LP decoder has converged at this point, and the decoder can immediately detect such a decoding error, we can constrain the LP further to try to get to the ML solution.

If the LP decoder fails, we add an integer constraint. We identify the symbol $\hat{x}_i$ whose value is closest to 0.5; i.e., the least certain symbol. We term this symbol i^*, defined as:

$$i^* = \arg\min_i |\hat{x}_i - 0.5|. \quad (10)$$

We then add the constraint $\hat{x}_{i^*} \in \{0, 1\}$ to the problem and re-run the ALP decoder. Since many LP solvers can accommodate integer constraints (we use GLPK [1] managed by a Python script) these constraints are easy to add. This choice of least-reliable bits is the same as is used in [8].

Overall we have the following algorithm:

1) Setup the initial problem according to (9)
2) Run the LP solver.
3) Search for all cuts of the current solution.
4) If one or more cuts are found, add them to the problem constraints and go to step 2.
5) If the ALP solution is non-integer, identify i^* according to (10), add in the constraint $\hat{x}_{i^*} \in \{0, 1\}$, and go to step 2.

The complexity of a mixed-integer linear program will grow exponentially with the number of enforced integer constraints.

Therefore, our algorithm will succeed in decoding in a reasonable time if and only if the required number of added integer constraints is not too large. Fortunately for some codes (notably LDPC codes) in some regimes (in the low-noise regime) a relatively small number of integer constraints are required. Thus, we sometimes obtain a practical ML (optimal) decoder, even though the general ML decoding problem is NP-hard.

IV. Numerical results

In this section we present the results of using our mixed-integer ALP approach on a $(N = 155, k = 64, d = 20)$ LDPC code presented in [6]. This code has an excellent minimum distance for its dimension and block-length, but is plagued by pseudo-codewords that greatly impair the performance of a BP or LP decoder (see [7] and [2] for more details).

Here we compare the word-error rate (WER) of "basic" ALP decoding, i.e., the original ALP relaxation of (6)–(7) without any additional integer constraints, to ML decoding obtained via our mixed-integer ALP decoder. We also provide statistics on the computation time requirements of our ML decoder and the number of integer constraints required.

To simplify our analysis, we work on the binary symmetric channel and study the performance for a fixed number of bit flips. Note that our decoder also works on other channels like the additive-white Gaussian noise (AWGN) channel. Because the minimum distance of this code is 20, an ML decoder is guaranteed to succeed if 9 or fewer bits are flipped. When 10 or more bits are flipped, an ML decoder may fail because another codeword is more likely than the transmitted codeword. We find that the number of required integer constraints and ALP decoding iterations grows with the number of bit flips, but is manageable for all bit flips up to 23. We employ a cap of 200 ALP decoding iterations (defined as the overall number of linear programs solved – pure linear programs or mixed-integer LPs) before giving up on a particular received word as taking too long to decode.

Since the rate of the $(155, 64)$ code is 0.4129, even if we could operate near capacity, for this relatively short code we could only expect to correct about 22 bit flips. Therefore we simulate up to 23 bit flips and simply assume decoding will fail with probability 1 for the very high noise regime of more than 23 bit flips. (This is slightly pessimistic given that the ML WER is about "only" .73 for 23 bit flips but is also realistic given that for 24 or more bit flips, the decoder runs like molasses.) We ran decoding experiments at each number of bit flips from 23 down to 12 until we accumulated 200 ML decoding errors at each bit flip level.

For 10 and 11 bit flips, the ML decoder performed very well, but it was difficult to obtain enough failures from simulations. At 11 bit flips we only accrued 79 ML decoding errors. Therefore, we estimate the performance as follows. We start by noting that, e.g., in a 12-bit flip failure, at least 10 of the flips must overlap another codeword, or else the ML decoder would decode to the codeword that was transmitted. Empirically nearly all failures are produced when exactly 10 bits overlap;

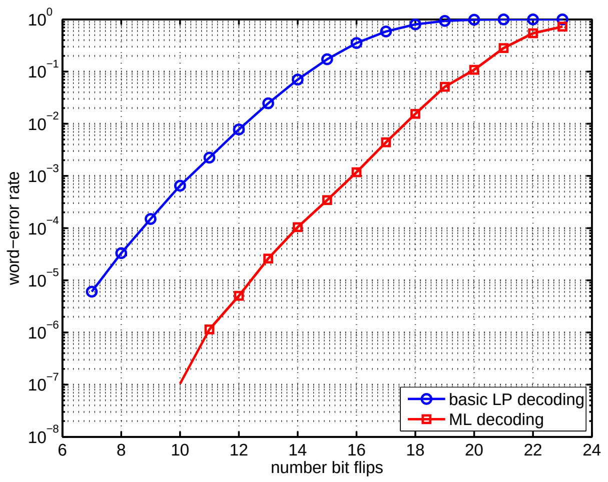

Fig. 1. Basic ALP and ML decoding word-error rates (WERs) as a function of the number of bit flips. Each ML data point corresponds to 200 decoding failures, except for the point corresponding to 11 bit flips where we found 79 failures.

11 bit and 12 bit overlaps are much less likely. Say that in such a case we start with a 12 bit failure pattern and reduce by one the number of bit flips. Then, the probability that we take away one of the two (non-overlapping) bits so that we would still have a failure is $(2/12)$. The resulting estimated error probability of 8.3e-$7 = (2/12)5.0$e-6 for 11 bit flips is in rough agreement with our experimental observation of 1.1e-6 based on only 79 decoding failures. We use the same idea to estimate the WER at 10 bit flips to be $(1/11)$th of the WER at 11 bit flips.

Basic ALP decoding continues to fail when there are 9 or fewer bit flips. Therefore we continue the simulation for LP decoding errors down to 7 bit flips. We again emphasize that an ALP decoder gives the same performance as other LP decoders–the only difference is that it is more efficient.

The results of these experiments are tabulated in Table I, and plotted in Fig. 1. The estimates of the ML WER at 10 bit flips is also tabulated and plotted, but is plotted without the squares that indicate measured data points. From the figure one can mark the huge performance improvement given by using the mixed-integer ALP decoder compared to the basic ALP decoder.

To estimate the ML WER for a given crossover probability, we note again that the WER is zero for nine or fewer bit flips and assume that the WER equals one for 24 or more bit flips. We then calculate the probability of realizing each number of bit flips for a particular crossover probability and averaged the WERs of Table I weighted by the appropriate Binomial coefficient. The combination of knowing that no ML errors occur for nine or fewer bit flips, and the error statistics for larger number of bit flips, allows us to estimate ML performance down to much lower WERs than would be possible if we generated the number of bit flips stochastically. The resultant WERs as a function of crossover probability are plotted in Fig. 2.

# bit flips	number trials	# LP errors	LP WER	# ML errors	ML WER
7	11e6	66	6.1e-6	0	0
8	6.0e6	200	3.3e-5	0	0
9	1.3e6	200	1.5e-4	0	0
10	0.3e6	200	6.5e-4		1.0e-7*
11	69e6	1.5e5	2.2e-3	79	1.1e-6
12	40e6	3.1e5	7.8e-3	200	5.0e-6
13	7.7e6	1.9e5	2.5e-2	200	2.6e-5
14	1.9e6	1.4e5	7.0e-2	200	1.0e-4
15	5.8e5	1.0e5	0.17	200	3.4e-4
16	1.7e5	6.0e4	0.35	200	1.2e-3
17	4.5e4	2.7e4	0.59	200	4.4e-3
18	1.3e4	1.0e4	0.80	200	0.015
19	3.9e3	3.7e3	0.94	200	0.051
20	1.9e3	1.8e3	0.98	200	0.11
21	711	711	1.0	200	0.28
22	369	369	1.0	200	0.54
23	275	275	1.0	200	0.73

TABLE I

DECODING RESULTS, USED TO PLOT FIG. 1. THE NUMBER NOTED WITH $*$ IS AN ESTIMATE, AS DISCUSSED IN THE TEXT

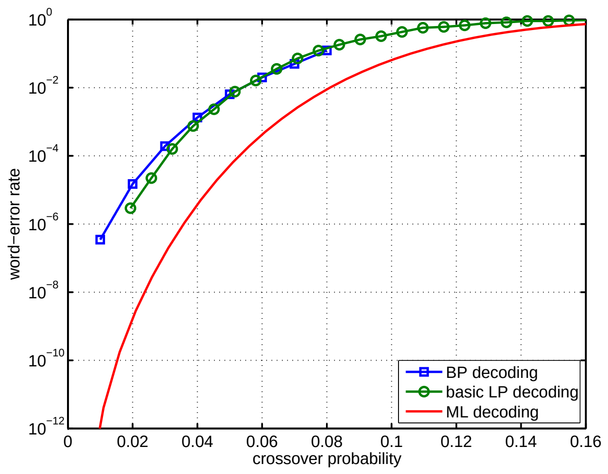

Fig. 2. BP, LP, and ML decoding word-error rates (WERs) as a function of channel crossover probability.

In contrast to the ML plot, since we don't have estimates of LP WERs at 6 or fewer bit flips, and yet errors still occur, we cannot get good estimates of the WER for low crossover probabilities. Therefore, for the LP results we simulated the channel stochastically, generating 200 LP decoding errors at crossover probabilities down to a crossover probability of 0.02. For comparison we also plot the performance of BP decoding (sum-product). As has been observed by others, the performance of BP and LP decoding are quite similar.

Another quantity of obvious interest is the computation requirement of mixed-integer ALP decoding. To give a feel of the computation time needed to produce our results, in Table II we provide statistics on decoding time. We tabulate the average, median, and maximum decoding times, in seconds, for 12, 14, 16, 18 and 20 bit flips. We give overall statistics, as well as the breakdown as categorized by whether or not a decoding error occurs. We note that we did not spend time

# bit flips	12	14	16	18	20
average time (all sims)	0.14	0.22	0.87	8.32	114
average time (correct dec)	0.14	0.22	0.85	7.17	75
average time (erroneous dec)	1.40	2.66	11.75	84.25	461
median time (all sims)	0.12	0.15	0.23	1.33	20.6
median time (correct dec)	0.12	0.15	0.23	1.29	16.4
median time (erroneous dec)	0.85	1.18	3.93	28.09	162
maximum time (correct dec)	822	824	499	881	3091
maximum time (erroneous dec)	7.1	72	290	2142	3878

TABLE II

ML DECODING TIME STATISTICS, IN SECONDS, FOR 12, 14, 16, 18, AND 20 BIT FLIPS, CATEGORIZED BY WHETHER A DECODING ERROR OCCURRED.

optimizing our code and, as has already been mentioned, used an off-the-shelf LP solver. All simulations were run on 3GHz Intel Xeon processors.

When a ML decoding error occurs, the received sequence is often quite far from a codeword – it must have moved at least halfway towards another codeword. Typically, more integer constraints are required to find the ML codeword in such cases. Since integer constraints are far more computationally costly than linear constraints, the decoding times when there are decoding errors are generally much larger than when decoding succeeds. However, this isn't true for the worst-case decoding times at 12, 14, and 16 bit flips. The explanation for this seems to be the following. When we gather the statistics for no errors at these bit flip levels, the worst case is calculated over hundreds of thousands of decoding trials. On the other hand, the statistics for when there are errors are calculated over only hundreds of trials. There are therefore many more chances to happen upon a noise sequence that is correctable, but that requires many integer constraints to correct. This reasoning is borne out by the empirical observation that across all erroneous decodings at 12, 14 and 16 bit flips fewer integer constraints were required than for the worst-case correct decoding (i.e., most number of integer constraints) at the same bit flip level.

We also collect statistics on the number of integer constraints required to decode. Although in our method it is easy to find such constraints, as discussed above, integer constraints slow the LP solver considerably as compared to regular linear constraints. In Fig. 3 we plot the number of integer constraints as a function of the number of bit flips. The top line is the worst case number of iterations, the next line depicts the 95th percentile–95% of the simulations at each bit flip level took at most the indicated number of integer constraints to find the ML codeword. We also indicate the 90th percentile and the 50th (the median). Note that the worst case is much worse than even the 95th percentile. These numbers combine all decodings (successes and failures). Recall that we imposed a cap of 200 ALP decoding iterations on our decoder. This cap kicks in only very rarely, and only at the highest bit flip levels. In our simulations it kicked in at least once only at 20, 22, and 23 bit flips. For all other numbers of bit flips it never kicked in.

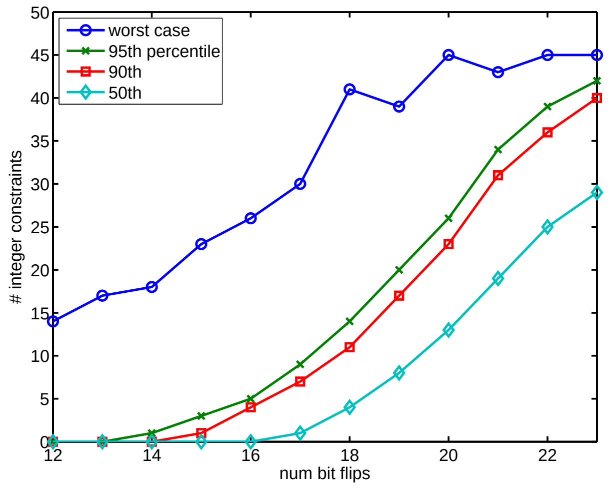

Fig. 3. Number of integer constraints required for ML decoding, plotted as a function of the number of bit flips.

By comparing Fig. 3 and Table II one can again see that the number of integer constraints has a large impact on the decoding time. Fig. 3 shows that the median case for 12, 14 and 16 bit flips is zero integer constraints. Compare this with the results in Table II. In the table we see that the median decoding times for all simulations and for correct decoding are identical, and are of the same order for 12, 14 and 16 bit flips, 0.12 , 0.15 and 0.23 seconds, respectively. These are representative decoding times for LP decoding via ALP without any integer constraints. On the other hand, when there are 18 or 20 bit flips, the median case uses a positive number of integer constraints, and the median decoding time increases sharply, to 1.33 and then 20.6 seconds for 18 and 20 bit flips, respectively.

V. Conclusions

In this paper we have introduced a simple way to add integer constraints to the ALP decoder to find the ML codeword. We exploit the computational efficiency of adaptive LP decoding to speed up our computations. We demonstrate the decoder on a (155,64) LDPC code, which is a surprisingly large code for which to obtain ML decoding performance.

We would like to note that even when the resulting ML decoder is too slow for a particular practical application, it may still be useful as a benchmark that gives the optimal decoding performance for a code.

There are a number of directions that build out of these initial ideas. We are most interested in exploring alternate ways of choosing the integer constraints. The criteria for choosing the "least reliable bits" made herein is more relevant for BP decoding since the value of each bit is an estimate of its log-likelihood ratio. The relationship to a tightening of the LP is not clear, though experimentally we observe that choosing only a small number of integer constraints in this manner leads to the ML solution. Perhaps with some small search we can do a better job of selecting the constraint most sensitive to the integer constraint, or determining if we can introduce multiple constraints at once. In addition we want to compare this approach to alternate ways of constraining the original LP, such as the redundant parity-check approach proposed in [4] and explored in [5], and "lift and project" [3].

Beyond the algorithmic specifics we are interested in building a hybrid BP/LP approach. At longer block-lengths (on the order of a few thousand) we observe that adaptive LP decoding slows considerably. One candidate system, similar to the "augmented BP" approach [7] uses BP to do its initial decoding. Only if BP gets stuck does the algorithm switch to LP. Hopefully when BP has gotten stuck it yields a good starting point for LP. Such an approach may allow us to get particularly good performance in the error floor region.

Finally, a caveat is in order. Although ALP decoders also work on *high* density parity check codes, we found that our mixed-integer LP decoder becomes intolerably slow on such codes so that, for example, it appears to be useless for BCH codes. Apparently the decoder will require a small number of integer constraints only when the pseudo-codewords that cause decoding to fail are relatively close to the transmitted codeword. Fortunately, that is often the case for the pseudo-codewords that cause LDPC error floors.

References

[1] GNU Linear Programming Kit. At http://www.gnu.org/software/glpk.

[2] M. Chertkov and M. Stepanov. An efficient pseudo-codeword search algorithm for linear programming decoding of LDPC codes. *Submitted to IEEE Trans. on Inform. Theory*, January 2006.

[3] J. Feldman. *Decoding Error-Correcting Codes via Linear Programming*. PhD thesis, Massachusetts Institute of Technology, 2003.

[4] J. Feldman, M. J. Wainwright, and D. Karger. Using linear programming to decoding binary linear codes. *IEEE Trans. Inform. Theory*, 51:954–972, March 2005.

[5] M. H. Taghavi N. and P. H. Siegel. Adaptive linear programming decoding. In *Proc. Int. Symp. Inform. Theory*, pages 1374–1378, Seattle, USA, July 2006.

[6] R. M. Tanner, D. Sridhara, and T. Fuja. A class of group-structured LDPC codes. In *Proc. ICSTA*, Ambleside, UK, 2001.

[7] N. Varnica, M. Fossorier, and A. Kavcic. Augmented belief-propagation decoding of low-density parity check codes. *Forthcoming, IEEE Trans. Commun.*

[8] K. Yang, J. Feldman, and X. Wang. Nonlinear programming approaches to decoding low-density parity-check codes. *IEEE J. Select. Areas Commun.*, 24:1603–1613, August 2006.