Extended Bit-Filling and LDPC Code Design

Jorge Campello and Dharmendra S. Modha

Abstract— **Recently, we [1] proposed a simple-to-implement heuristic, namely, BIT-FILLING, for constructing high rate and high girth LDPC codes. In the present work, we extend BIT-FILLING, and demonstrate that the extended algorithm produces better codes, that is, codes with higher rate/girth and good bit error rate performance. We demonstrate the positive effect of girth on bit error rate performance.**

I. INTRODUCTION

Low density parity-check codes (LDPC) were introduced by Gallager [2]. Recently, it has been shown that LDPC codes can perform very close to the Shannon capacity limit when their associated Tanner graphs possess certain desirable properties, see, for example, [3], [4].

A LDPC code can be described by specifying its $m \times n$ binary parity check matrix H. The name "low density" refers to the fact that the matrix H is very sparse. We say that the *length* of the LDPC code is n and its *rate* is $(n-m)/n$. Given a parity check matrix H, we can define its associated Tanner graph $G(H) = (V, E)$ as a bipartite graph with $m+n$ vertices $V = \{1, 2, \ldots, m, m+1, \ldots, m+n\}$. The first m vertices correspond to m parity check equations, and are referred to as the *check nodes*. The last n vertices are referred to as the *bit nodes*. For $1 \leq i \leq m$ and $1 \leq j \leq n$, there is an edge $(i, m+j)$ in E iff $H_{i,j} = 1$. The length of the smallest cycle in a graph is known as its *girth*. The importance of large girth stems from the fact that when decoding LDPC codes using the sum-product decoding algorithm, the number of independent iterations of the algorithm is proportional to the girth of the Tanner graph corresponding to the code [2, Lemma C.1]. Furthermore, it is known that minimum distance increases with girth [5]. For pointers to various approaches to constructing LDPC codes, see [1].

Recently, we considered the following problem, and proposed a heuristic algorithm, namely, BIT-FILLING , for it.

Problem I.1 *Suppose that we are given positive integers m, b, $\{a_1, a_2, \ldots\}$, and g; where g is required to be even. We would like to construct a $m \times n$ parity check matrix H with the* **largest possible rate** *(that is, largest n) such that H has exactly a_j ones in the j-th column, $1 \leq j \leq n$, at most b ones in each row, and $G(H)$ has girth g.*

In this paper, we consider the following slightly different problem.

Problem I.2 *Suppose that we are given positive integers m, N, b, and $\{a_1, a_2, \ldots, a_N\}$. We would like to construct a $m \times N$ parity check matrix H with the* **largest possible girth** *$G(H)$ such that H has exactly a_j ones in the j-th column, $1 \leq j \leq n$, and at most b ones in each row.*

Suppose we have an algorithm for solving Problem I.2, then it can be easily leveraged to solve Problem I.1. Fix parameters m, $\{a_1, a_2, \ldots, a_N\}$, b, and g in Problem I.1. Now, apply the algorithm for Problem I.2 with parameters m, $\{a_1, a_2, \ldots\}$, b, and N. Denote the resulting $m \times N$ parity check matrix as H_N. Let n denote the largest N such that $G(H_N) \geq g$. Then, H_n constitutes an answer to Problem I.1.

It turns out, however, that Problem I.2 leads us to an extension of the BIT-FILLING algorithm that yields codes with better bit error rate performance, and, somewhat surprisingly, leads to better solutions of Problem I.1. The extended algorithm has essentially the same computational complexity as before, that is, $O(bm^3)$ (a simpler version can be implemented using $O(bm^2)$ operations). As a benchmark, our algorithm finds better or comparable rates to some of the highest rate codes found by MacKay [6].

II. THE MAIN IDEA

The basic design principle for the extended version remains the same as that for the original version–where the idea was to put the bits in the graph one-by-one while always maintaining the prescribed girth constraint. The central new idea in this version is that the girth constraint varies throughout the execution of the algorithm. The algorithm starts out by insisting on a large girth constraint, $\overline{g}$ until it can no longer add more bits without violating the girth constraint. At that point, the girth constraint is decreased (by 2) and the algorithm continues. The algorithm progresses in this fashion until either (a) all the needed bits, namely, N, are put into the graph or (b) the girth constraint falls below a specified minimum value, say, $\underline{g}$. Observe that in case (a) the algorithm would find a solution to Problem I.2, while in case (b), if we had set $N = \infty$ and $\underline{g} = g$, it would find a solution to Problem I.1.

We now outline the modified algorithm.

The algorithm in itself makes no assumptions about the relative ordering of $a_1, a_2, \ldots$. However, as discussed in Section III-C, in practice, it turns out better to assume that $a_1 \leq a_2 \leq \ldots$, that is, allocating bits with low degrees first, and then progressively moving to higher degree bits leads to codes with better performance.

Now, suppose that we have already constructed a matrix H with n columns, $n \geq 0$, that satisfies all the constraints, that is, the weight of the k-th column is exactly a_k for $k = 1, 2, \ldots, n$, each row has weight at most b, and the associated Tanner graph $G \equiv G(H)$ has girth at least $g' \geq \underline{g}$. We now show how to add the $(n+1)$-th column to H. We think of the new column to be added as a set U_1 which has size at most a_{n+1} and is initially empty. The set U_1 is a set of check nodes and, hence, is a subset of $\{1, 2, \cdots, m\}$. Further suppose we have already added i check nodes, $0 \leq i < a_{n+1}$,to U_1. The following procedure attempts to add the $(i+1)$-th check node to U_1. It may fail, and, in that case, the whole procedure stops.

In Figure 1, we present the algorithm based on the outline sketched above. Each iteration of the outer "do $\cdots$ while" loop

J. Campello and D. S. Modha are with the IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120. E-mails: {campello,dmodha}@us.ibm.com

0-7803-7206-9/01/$17.00 © 2001 IEEE

```
Inputs: m, N, b, {a_1, a_2, ..., a_N}, \overline{g}, and \underline{g}

1:  set n = 0, A = {1, 2, ..., m}, and U_1 = φ, g' = \overline{g}
2:  for c ∈ {1, 2, ..., m}, set deg(c) = 0 and N_c = φ
3:  do {
4:      ∀c ∈ U_1, set H_{c,n} = 1 and increment deg(c) by 1
5:      set i = 0, U_1 = φ, and U = φ
6:      do {
7:          compute F_0 = A \ U
8:          if (F_0 ≠ φ) {
9:              choose c* from F_0 according to the heuristic in [1, Section II.B]
10:             ∀c ∈ U_1, update N_c = N_c ∪ {c*} and update N_{c*} = N_{c*} ∪ U_1
11:             update U_1 = U_1 ∪ {c*}, U = U ∪ V_{(g'/2)-1}(c*), and A
12:             i = i + 1
13:         } else{
14:             g' = g' - 2
15:             recalculate U = ∪_{1≤j≤(g'/2)-1} U_j
16:         }
17:     } while ((i < a_n) and (g' ≥ \underline{g}))
18:     n = n + 1
19: } while ((n < N) and (g' ≥ \underline{g}))

Outputs: n, g', H_n ≡ H
```

Fig. 1. The BIT-FILLING Algorithm.

in lines 3–19 attempts to add a column to the parity check matrix. If an iteration succeeds, then, in the beginning of the next iteration, line 4 updates the matrix H. Each iteration of the inner "do $\cdots$ while" loop in lines 6–17 attempts to add a check node to the set U_1.

To describe the procedure, it helps to think of the Tanner graph G; G has m check nodes and n bit nodes. The process of adding the $(n+1)$-th column of H is like adding the $(n+1)$-th bit node to G. In this context, U_1 can be thought of as the set of check nodes that have been already connected to the $(n+1)$-th bit node that is being added. The process of adding a check node $c^\star$ to U_1 is like adding an edge from the $(n+1)$-th bit to $c^\star$-th check node. This new edge must not create any cycles of length $(g'-2)$ or smaller; we now describe a test to enforce this constraint.

For a check node $1 \leq c \leq m$, let $\mathcal{N}_c$ denote the set of all check nodes that share a bit node with it. In other words, $\mathcal{N}_c$ is the set of all check nodes that are exactly two distinct edges away from c. For $j \geq 2$, define

$$U_j = \bigcup_{c \in U_{j-1}} \mathcal{N}_c. \quad (1)$$

Intuitively, there is a path of length 2 from every check node in U_2 to some check node in U_1. Now adding a check node $c^\star$ to U_1 will create a path of length 2 from $c^\star$ to every check node in U_1. So, if $c^\star$ is in U_2, then we are guaranteed a cycle of length 4. Hence, to avoid cycles of length 4, we should avoid the check nodes in U_2. Continuing in this fashion, there is a path of length 2 from every check node in U_j to some check node in U_{j-1}, and, hence, there is a path of length at most $2j-2$ from every check node in U_j to some check node in U_1. Thus, adding a check node in U_j to U_1 will create a cycle of length $2j$ or smaller. Hence, to satisfy the girth constraint, we should avoid adding to check nodes in the following set, U, to U_1:

$$U = \bigcup_{1 \leq j \leq (g'/2)-1} U_j. \quad (2)$$

Let deg(c) denote the degree of the check node c. Let

$$A = \{c \in \{1, 2, \ldots, m\} : \text{deg}(c) < b\}$$

denote set of check nodes that are connected to fewer than the maximum number of bits nodes allowed. Then, the set of feasible check nodes that can be added to U_1 without violating the girth or the check-degree constraint is

$$F_0 = A \setminus U.$$

If F_0 is empty, the current girth is decreased, if g' falls below $\underline{g}$ the procedure stops. For the important issue of how to select a check node from F_0, please see [1, Section II.B]. We now briefly discuss an efficient implementation of (2).

Observe that, as check nodes are added to U_1, it is computationally more efficient to incrementally update U in line 10 than recomputing it afresh using (1)–(2). Specifically, if we have added a check node c to U_1, then we set $U_1(c) = \{c\}$ and, for $2 \leq j \leq (g'/2)-1$, compute

$$U_j(c) = \bigcup_{c' \in U_{j-1}(c)} \mathcal{N}_{c'} \quad (3)$$

and compute

$$V_{(g'/2)-1}(c) = \bigcup_{1 \le j \le (g'/2)-1} U_j(c). \tag{4}$$

Finally, update U using $U = U \cup V_{(g'/2)-1}(c)$. The only time we have to recalculate U from scratch is when the girth constraint is decreased, which happens at most $(\overline{g} - \underline{g})/2$ times.

III. RESULTS AND DISCUSSION

A. *Constructing High Rate Codes*

As with the original BIT-FILLING algorithm, we can use the extended algorithm to generate codes with high rate and a given girth. As pointed out in the Section II, the idea is to set $N = \infty$ and $\underline{g} = g$. In words, we let the algorithm put as many bits as it can without violating the girth constraint. This procedure is suitable for generating Gallager (regular) codes. In [1] we compared the rates of the codes constructed using BIT-FILLING with those collected by MacKay in [6]. Amongst these codes, he singles out 9 codes as "the following numbers give the highest rate codes I have made with column weights 3 and 4." In the following table, we compare these codes to those generated by our algorithm.

a	m	n		
			BIT-FILLING	
		MacKay	I	II
3	60	**492**	485	489
3	62	495	483	**508**
3	90	998	1087	**1120**
3	100	900	1339	**1353**
3	111	999	1636	**1717**
4	222	1998	2752	**2967**
4	282	4376	4821	**4867**
4	300	4096	5499	**5499**
4	444	3584	12360	**12370**

The I and II columns refer, respectively, to using the original or the extended version of the BIT-FILLING algorithm. Also, for all our codes, we set $\overline{g} = 200$ and $\underline{g} = g = 6$. It is evident that in 8 out of 9 cases our algorithm yields higher rate codes, and in 1 case, while not better, it yields a code with virtually identical rate.

In Figure 2, we compare the performance of two codes, say, A and B, in [6] with parameters $m = 222$, $n = 1998$, $a = 4$, $b = 36$, and $g = 6$ to a code with the same parameters generated by the extended BIT-FILLING algorithm. The latter was obtained by deleting the excess columns of the code corresponding to the sixth row in the above table. The comparison was made in terms of bit error rate as a function of the signal-to-noise ratio (SNR) on a simulated binary-input AWGN channel. It can be seen that these three codes have virtually identical performance. To summarize, on identical parameters, the BIT-FILLING codes perform on par with some of the best known codes. However, BIT-FILLING offers more flexibility in the code design parameters. For example, one may choose to trade-off the bit error rate performance against the code rate.

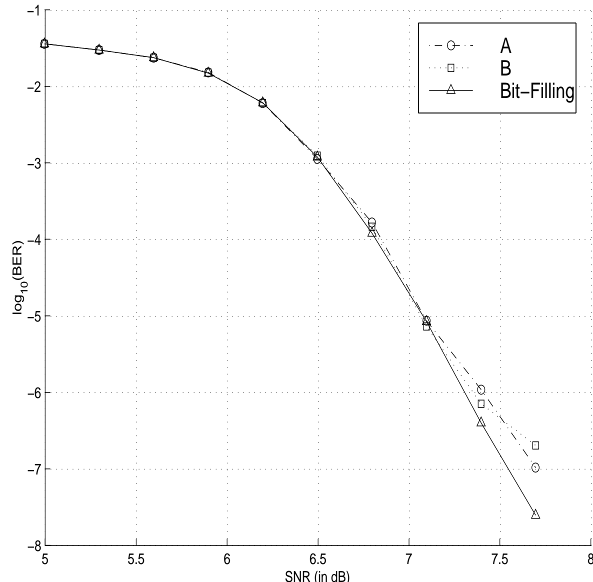

Fig. 2. A comparison of three LDPC codes with parameters $m = 222$, $n = 1998$, $a_1 = a_2 = \ldots = a_n = 4$, $b = 36$, and $g = 6$.

B. *Maximizing the Girth*

All the Gallager codes listed in [6] have girth 6, that is, avoid cycles of length 4. As shown above, for virtually all of the high rate codes our algorithm matches the girth, and, in most cases, is even able to generate higher rate codes. More interestingly, for codes with lower rates, our algorithm is able to improve the girth parameter–sometimes considerably. For example, in the following table we show some of the codes listed in [6] whose girth we are able to improve.

a	m	N	$\overline{g}$	g	
				MacKay	BIT-FILLING
3	408	816	10	6	8
3	504	1008	$\{82, 22\}$	6	8
4	544	816	18	6	8
3	272	408	22	6	10
3	544	816	$\{82, 22\}$	6	10
3	1280	1920	22	6	12

C. *Irregular Graphs*

In a recent work, Richardson, Shokrollahi, and Urbanke [4] have shown that the performance of LDPC codes can be improved by using irregular constructions with a carefully chosen bit-degree sequence, that is, by using an optimized distribution of the bit-degrees. BIT-FILLING can be used to easily construct LDPC codes with large girth having a prescribed bit-degree distribution. Suppose that we want to construct a code with bit-degree distribution given by

degree	fraction of bits
d_1	n_1/N
d_2	n_2/N
$\vdots$	$\vdots$
d_r	n_r/N

where $d_1 < d_2 < \ldots < d_r$ and $N = \sum_{i=1}^{r} n_i$. That means that for a code of length N, there would be n_i bit nodes with degree d_i, $1 \leq i \leq r$. Let $N_l = \sum_{i=1}^{l} n_i$. Then, to use BIT-FILLING, we set

$$a_j = \begin{cases} d_1 & \text{for} \quad 0 < j \leq N_1 \\ d_2 & \text{for} \quad N_1 < j \leq N_2 \\ & \vdots \\ d_r & \text{for} \quad N_{r-1} < j \leq N_r = N \end{cases}$$

for all $1 \leq j \leq N$.

The ordering of the bit degrees is important. In the assignment above, the first bits to be inserted into the graph have the lowest degree. Since BIT-FILLING starts with a large girth constraint $\overline{g}$, it means that the subgraph formed by considering only the bits that have low degree will have a high girth. Another consequence is that there will be no cycles of low weight (where the weight of the cycle is given by the sum of the degree of its bit nodes). It was pointed out to us (JC) by Thomas Richardson that avoiding cycles of small weight may be more important for performance than simply increasing the girth.

As an example, we constructed four irregular LDPC codes with the same degree distribution, three of them using a randomized construction and one using the BIT-FILLING algorithm. The four codes have parity-check matrices with parameters $N = 1000$ and $m = 300$, and they all have the same bit degree distribution. The degree distribution, shown below, was obtained by using the optimization procedure in [7].

degree	franction of bits
2	$246/1000$
3	$295/1000$
5	$8/1000$
8	$336/1000$
9	$115/1000$

The randomized construction was performed as follows. We start with an empty matrix and a random assignment of degrees to columns of the matrix so as to satisfy the bit degree distribution. Then we randomly traverse the entries of the matrix and put a 1 in that entry if all of the following conditions are met

- The degree constraint for the corresponding check has not been met.
- The degree constraint for the corresponding bit has not been met.
- The resulting matrix does not contain cycles of length four in its associated Tanner graph.

We continue until either (a) the desired distribution has been allocated or (b) all the entries have been traversed. In case (a), we declare "success", and in case (b) we declare a "failure" and restart with a different seed for the pseudorandom number generator.

Finally, we generated a code using the BIT-FILLING algorithm with the same parameters as those for the randomly constructed codes. In the BIT-FILLING algorithm, we set $\overline{g} = 2m + 2$. Simulation results comparing the three randomly constructed codes and the code constructed using BIT-FILLING are given in Figure 3. As can be seen, for low values of SNR, the four codes are equivalent. But, as the SNR increases, the randomly constructed codes start to suffer performance degradation, whereas the bitt-filling code maintains good performance. From this, we conclude that at least for high SNR it is important to avoid cycles of small weight.

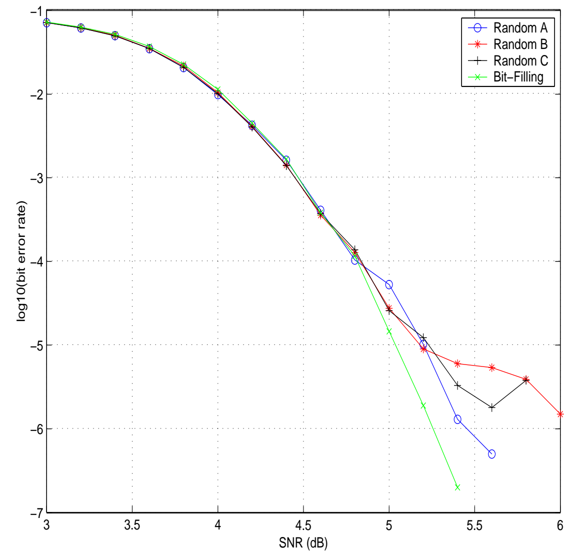

Fig. 3. A comparison of four irregular LDPC codes with the same degree distribution and with parameters $m = 300$, $N = 1000$, and $g = 6$. For the code generated by the BIT-FILLING algorithm, we set $\overline{g} = 2m + 2$.

D. *Impact of Girth*

To underscore the positive effects of increasing girth, we compared several regular and irregular codes of the same size but with increasing girths.

D.1 Regular Codes

We constructed regular codes with $m = 2000$, $N = 3502$, $a_j = 3, 1 \leq j \leq N$, $b = 6$, and $\overline{g} = \underline{g} \in \{6, 8, 10, 12\}$. The experimental results comparing these codes are given in Figure 4. As can be seen, quite substantial performance improvements can be obtained from increasing the girth for regular LDPC codes.

D.2 Irregular Codes

We also performed the experiment with irregular LDPC codes of same size but with increasing girths. All four codes had $m = 700$, $N = 1000$, and the same optimized bit degree distribution obtained from [7]. The four codes differed in their girths which were $6, 8, 10$, and 12, respectivaly. The results of this

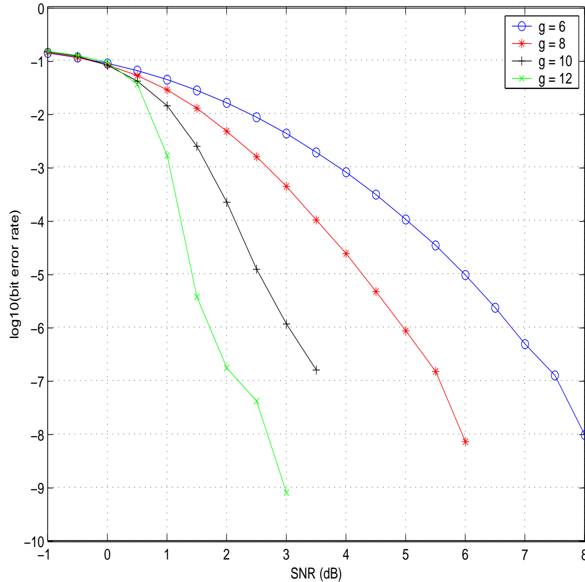

Fig. 4. A comparison of four **regular** LDPC codes with different girths. All the codes are regular with parameters $m = 2000$, $N = 3502$, $a_j = 3, 1 \leq j \leq N$, $b = 6$, and $\overline{g} = \underline{g} = g \in \{6, 8, 10, 12\}$.

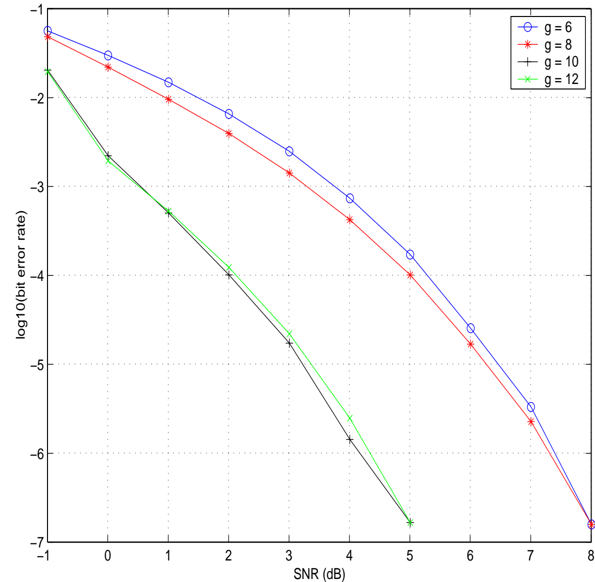

Fig. 5. A comparison of four **irregular** LDPC codes with different girths. All of the codes are irregular with parameters $m = 700$, $N = 1000$, and $\overline{g} = \underline{g} = g = \{6, 8, 10, 12\}$. All four codes use the same optimized bit-degree sequence obtained from [7].

experiment are given in Figure 5. As can be seen, while the relationship between girth and perfomance is not as direct as for regular codes, there is still a general trend of improvement with increasing girth.

E. Effect of $\overline{g}$

Observe that essentially all values of $\overline{g}$ larger than $2m$ are equivalent in that they lead to the same code. Furthemore, if we set $\overline{g} = \underline{g} = 2m + 2$, then we are essentially demanding a cycle free graph. We note that the extended BIT-FILLING algorithm seems to be mildly affected by the choice of $\overline{g}$. In other words, setting $\overline{g} = 2m + 2$ does not always lead to the best codes; in fact, sometimes a smaller $\overline{g}$ leads to better codes. For example, in Section III-B, we experimented with various values for $\overline{g}$.

F. Computational Complexity

The computational complexity of the extended BIT-FILLING algorithm in Figure 1 is the same as that for the original BIT-FILLING. That is, the complexity of the algorithm is $O(m^3)$, and if the first-order heuristic in [1, Section II.B] is used the complexity is only $O(m^2)$. To see this, note that the only extra work performed by the extended algorithm is the recalculation of the set U performed in line 15. This recalculation can be done in $O(m)$ time, and since it can only occur $(\overline{g} - \underline{g})/2$ times during the execution of the algorithm, it is clear that it does not affect the overall running time.

REFERENCES

[1] J. Campello, D. S. Modha, and S. Rajagopalan, "Designing ldpc codes using bit-filling," in *Proc. Int. Conf. Communications (ICC), Helsinki, Finland*, 2001.

[2] R. G. Gallager, *Low-density parity-check codes*. MIT Press, Cambridge, MA, 1963.

[3] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inform. Theory*, vol. 45, pp. 399–431, March 1999.

[4] T. J. Richardson, A. Shokrollahi, and R. L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inform. Theory*, no. 2, pp. 619–637, 2001.

[5] R. M. Tanner, "A recursive approach to low-complexity codes," *IEEE Trans. Inform. Theory*, pp. 533–547, 1981.

[6] D. J. C. MacKay, "Encyclopedia of sparse graph codes," 1999. http://wol.ra.phy.cam.ac.uk/mackay/codes/data.html.

[7] S. Chung, T. J. Richardson, and R. L. Urbanke, 1999. http://lthcwww.epfl.ch/ldpc/gaopt.html.