

Designing LDPC Codes Using Bit-Filling

Jorge Campello[†], Dharmendra S. Modha[‡], Sridhar Rajagopalan[†]

[†]IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

emails: {campello,sridhar}@almaden.ibm.com

[‡]7074 Via Ramada, San Jose, CA 95139

email: modha@ieee.org

Abstract—Bipartite graphs of bit nodes and parity check nodes arise as Tanner graphs corresponding to low density parity check codes. Given graph parameters such as the number of check nodes, the maximum check-degree, the bit-degree, and the girth, we consider the problem of constructing bipartite graphs with the largest number of bit nodes, that is, the highest rate. We propose a simple-to-implement heuristic BIT-FILLING algorithm for this problem. As a benchmark, our algorithm yields codes better or comparable to those in MacKay [1].

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 parity check matrix H , which is an $m \times n$ binary matrix. 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 $V = \{1, 2, \dots, m, m+1, \dots, m+n\}$. 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]. In this paper, we are interested in the following design problem in constructive combinatorics. Suppose we are given positive integers a, b, g , and m ; where g is required to be even. We are interested in constructing a $m \times n$ parity check matrix H with the largest possible rate (that is, the largest possible number of columns n) such that H has exactly a ones in each column, at most b ones in each row, and $G(H)$ has girth g .

As our main contribution, we propose a simple-to-implement heuristic BIT-FILLING algorithm for the above problem. Our algorithm has a computational complexity of $O(bm^3)$ (a simpler version can be implemented using $O(bm^2)$ operations). This is in sharp contrast to various

ad-hoc, random constructions in use today that often have exponential time complexity, and, hence, can be made to work only for small column weights and for small values of g (for example, $g = 6$). As a benchmark, our algorithm finds better or comparable rates to some of the highest rate codes found by MacKay [1].

Our algorithm can easily be adapted to the following important cases.

fixed rate, high girth: Given the number of check nodes m , the number of bit nodes n , the check-degree b , and bit-degree a (such that $mb = na$), maximize the girth g .

fixed girth, fixed length, high rate: Given the length of the code n , the bit-degree a , and the girth g , minimize the number of check nodes m , that is, maximize the rate.

Finally, our algorithm can easily be modified to construct several irregular graphs.

We now briefly review various approaches to constructing LDPC codes. In his original monograph, Gallager [2, Appendix] gave an algorithm for the fixed rate construction. For various algebraic constructions, also in fixed rate setting, see Margulis[5] and Lubotzky, Phillips, and Sarnak [6]. For various recent designs of graphs with girth 6 using mainly random constructions, see, for example, MacKay, Simon, and Davey [7] and MacKay and Davey [8]. For designs of irregular graphs using random constructions and linear programming, see Luby et al. [9]. Recently, Fan [10] has considered design of LDPC codes using array codes, Mitelholzer [11] has considered design of LDPC codes using Steiner designs, and Yu, Lin, and Fossorier [12] have considered design of LDPC codes using finite geometries.

II. THE BIT-FILLING ALGORITHM

A. The Idea

Suppose we have already constructed a matrix H with n columns, $n \geq 0$, that satisfies all the constraints, that is, the weight of each of its columns is exactly a , no row has weight more than b , and the associated Tanner graph $G \equiv G(H)$ has girth at least 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 and is initially empty. The set U_1 is a set of check nodes and, hence, is a subset of $|m| \equiv \{1, 2, \dots, m\}$. Further suppose we have already added i check nodes, $0 \leq i < a$, to U_1 . The following procedure attempts to add $(i+1)$ -th check node to U_1 . It

```

1: set  $n = 0$ ,  $A = |m|$ , and  $U_1 = \phi$ 
2: for  $c \in |m|$ , set  $\text{weight}(c) = 0$  and  $\mathcal{N}_c = \phi$ 
3: do {
4:    $\forall c \in U_1$ , set  $H_{c,n} = 1$  and increment  $\text{deg}(c)$  by 1
5:   set  $i = 0$ ,  $U_1 = \phi$ , and  $U = \phi$ 
6:   do {
7:     compute  $F_0 = A \setminus U$ 
8:     if ( $F_0 \neq \phi$ ) {
9:       choose  $c^*$  from  $F_0$  according to some heuristic
10:       $\forall c \in U_1$ , update  $\mathcal{N}_c = \mathcal{N}_c \cup \{c^*\}$  and update  $\mathcal{N}_{c^*} = \mathcal{N}_{c^*} \cup U_1$ 
11:      update  $U_1 = U_1 \cup \{c^*\}$ ,  $U = U \cup V_{(g/2)-1}(c^*)$ , and  $A$ 
12:    }
13:     $i = i + 1$ 
14:  } while ( $(i < a)$  and ( $F_0 \neq \phi$ ))
15: } while ( $(i = a)$  and ( $F_0 \neq \phi$ ))

```

Fig. 1. The BIT-FILLING Algorithm.

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 ... while” loop in lines 3–15 attempts to add a column to the parity check matrix. If an iteration succeeds, then, in the beginning of the next iteration, line 4 adds updates the matrix H . Each iteration of the inner “do ... while” loop in lines 6–13 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 $(n + 1)$ -th column of H is like adding $(n + 1)$ -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^* to U_1 is like adding an edge from the $(n + 1)$ -th bit to c^* -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^* to U_1 will create a path of length 2 from c^* to every check node in U_1 . So, if c^* is in U_2 , then we are guaranteed a 4-cycle. Hence, to avoid 4-cycles, 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 check nodes in the set

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

to U_1 .

Let $\text{deg}(c)$ denote the degree of the check node c . Let

$$A = \{c \in |m| : \text{deg}(c) < b\}$$

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

$$F_0 = A \setminus U.$$

The procedure terminates if F_0 is empty. Before we take up the important issue of how to select a check node from F_0 , we 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

$$V_{(g/2)-1}(c) = \bigcup_{1 \leq j \leq (g/2)-1} U_j(c). \quad (4)$$

Finally, update U using $U = U \cup V_{(g/2)-1}(c)$.

B. Main Heuristic

The most crucial step of the BIT-FILLING algorithm is the choice of the check node $c^* \in F_0$ in line 9 of Figure 1.

While any choice of c^* from F_0 is valid, judicious selection of c^* is crucial to achieving high-rate codes. Technically, using back-tracking and recursion, we can try every possible choice, and then “conditioned” on that choice try every future choice, etc., thus searching through an extremely large tree of possibilities. Of course, we often wish we could live life in this way! Unfortunately, while optimal, such exhaustive search is computationally infeasible. Also, a large number of choices actually lead to isomorphic graphs and are thus equivalent from our perspective. The role of heuristics is to pick one seemingly effective, and yet computationally feasible, path through this gargantuan tree of possibilities.

We let our choice of c^* in line 9 be guided by the simple principle of keeping the Tanner graph G as *homogeneous* as possible, that is, from the set of all feasible check nodes pick the check node that is least used so far. This choice amounts to keeping all parts of the graph equally dense. Also, from the perspective of the $(n+1)$ -th bit node that is being added, this choice connects it to the check node that is least used. We now make our choice precise.

As a first try, we look at the following subset of F_0

$$F_1 = \{c_1 \in F_0 : \forall c_2 \in F_0, \deg(c_1) \leq \deg(c_2)\} \quad (5)$$

namely, the set of smallest degree check nodes in F_0 . As a FIRST-ORDER heuristic, we may simply choose c^* to be any element of F_1 . We shall refer to this heuristic as “1-h” meaning first-order homogeneity. We will show in Section III that this first-order heuristic already yields quite competitive codes.

Typically, while F_1 is smaller than F_0 , it does not uniquely determine a check node. To further narrow available choices, we appeal once more to homogeneity. The idea is to look at the degrees of check nodes that are two edges away from the check nodes in F_1 . We write

$$F_2 = \{c_1 \in F_1 : \forall c_2 \in F_1, W_2(c_1) \leq W_2(c_2)\},$$

where

$$W_2(c) = \sum_{c' \in V_2(c)} \deg(c')$$

and $V_2(\cdot)$ is as in (4). Now, we may try selecting a check node from F_2 . However, once again, it is possible that F_2 has more than one element. Thus, to further narrow the set of choices, we may look at the degrees of check nodes that are four edges away from the check nodes in F_1 , and can continue in this fashion. The basic idea can now be described algorithmically, see Figure 2. The idea is to progressively look at larger and larger neighborhoods of the feasible check nodes in order to distinguish them; we refer to this heuristic as “c-h” for complete-homogeneity. The “while ... endwhile” loop in lines 9C–9I terminates when the set of choices reduces to a set of cardinality one or till the set of

```

9A: set  $j = 0, E_0 = F_0$ 
9B:  $\forall c \in F_0$ , set  $U_1(c) = \{c\}$ 
9C: while  $(|F_j| > 1)$  and  $(E_j = F_j)$  {
9D:    $j = j + 1$ 
9E:    $\forall c \in F_{j-1}$ ,
       compute  $W_j(c) = \sum_{c' \in V_j(c)} \deg(c')$ 
9F:   compute  $F_j =$ 
        $\{c_1 \in F_{j-1} : \forall c_2 \in F_{j-1}, W_j(c_1) \leq W_j(c_2)\}$ 
9G:    $\forall c \in F_j$ , compute  $V_{j+1}(c)$  using (4)
9H:   compute  $E_j = \{c \in F_j : V_{j+1}(c) \neq V_j(c)\}$ 
9I: } endwhile
9J: if  $(|F_j| = 1)$ 
9K:   select  $c^*$  from  $F_j$ 
9L: else
9M:   select  $c^*$  from  $F_j \setminus E_j$ 
9N: endif

```

Fig. 2. The “c-h” heuristic for selecting the check node c^* from F_0 . This heuristic is meant to replace line 9 in Figure 1.

choices does not further reduce in cardinality. The set E_j in line 9H contains all check nodes in F_j such that their neighborhood *can* be further enlarged. The loop terminates if $E_j \neq F_j$, since, in this case, the check nodes in E_j are guaranteed to be less homogeneous than those in $F_j \setminus E_j$. Furthermore, since, using homogeneity as our guide, we cannot further distinguish between the elements of $F_j \setminus E_j$, we simply make some choice from this set. Specifically, we select the lexicographically smallest check node from this set.

III. RESULTS AND DISCUSSION

Remark III.1 (xhd girth, high rate) MacKay has collected a large number of LDPC codes at [1]. Amongst the codes listed there, 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		
		MacKay	1-h	c-h
3	60	492	437	485
3	62	495	464	483
3	90	998	970	1087
3	100	900	1229	1339
3	111	999	1515	1636
4	222	1998	2628	2752
4	282	4376	4293	4821
4	300	4096	4807	5499
4	444	3584	10839	12360

The “1-h” and “c-h” columns refer to selecting c^* in line 9

of Figure 1, respectively, using the first-order heuristic in (5) and using the complete-homogeneity argument in Figure 2. Also, for all our codes, we required $g = 6$ and b was unrestricted. It is evident that in 7 out of 9 cases our algorithm yields higher rate codes, and in 2 cases, while not better, it yields virtually identical rates.

In Figure 3, we compare the performance of two codes, say, A and B, in [1] with parameters $m = 222$, $n = 1998$, $a = 4$, $b = 36$, and $g = 6$ to a code with the same parameters generated by the 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 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.

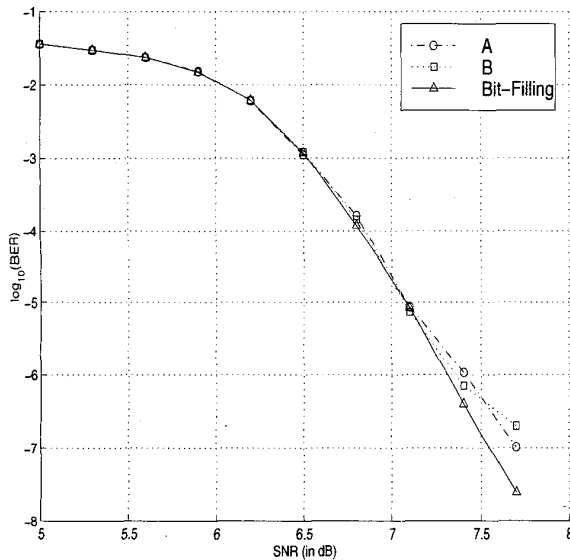


Fig. 3. A comparison of three LDPC codes with parameters $m = 222$, $n = 1998$, $a = 4$, $b = 36$, and $g = 6$.

Remark III.2 (fixed rate, high girth) So far we focussed on maximizing the rate given constraints on the number of check nodes, the check-degree, the bit-degree, and the girth. Another interesting problem, first considered in [2], is given the number of check nodes m , the check-degree b , and bit-degree a , maximize the girth g . For various algebraic constructions of fixed rate, high girth graphs, see [5], [6].

In this case, the number of bit nodes is $n = (mb)/a$. Our algorithm can be adapted to this case as follows. We simply apply the algorithm with given parameters m , a , b , and some girth g . If the algorithm finds exactly (resp. less than) n bit nodes, then the girth g is achievable (resp. not achievable) with our algorithm. By trying a few girth values (formally, doing a binary search on girth values), one can quickly determine the largest girth that is attainable using our algorithm.

Once again, we use the encyclopedia of codes compiled by MacKay [1] as our benchmark. All the Gallager codes listed in [1] have girth 6, that is, avoid 4 cycles. For all of these codes, our algorithm matches the girth. More interestingly, for codes with larger block lengths, our algorithm is able to improve the girth parameter. For example, for rate $1/2$ codes with $a = 3$, $b = 6$, and $m = 408$ our algorithm is able to improve the girth from 6 to 8. In this case, the algorithm achieves $n = 815$ —just one column less than the desired 816.

Remark III.3 (fixed girth, fixed length, high rate) The following observation is due to Ajay Dholakia. In practice, we are often interested in highest rate codes with fixed girth g , fixed length n , and fixed column weight a ; that is, given g , n , and a , we would like to minimize m . The BIT-FILLING algorithm can easily be used to construct codes that achieve as high a rate as possible by trying different values of m , in a binary search mode, and selecting the smallest m that either exceeds or achieves the desired length; and, by deleting the excess columns, if any. For comparison purposes, we reuse three of the codes from [1] that were used in Remark III.1, but this time we fix n and minimize m . The results are given in the table below.

a	n	m		
		MacKay	BIT-FILLING 1-h	BIT-FILLING c-h
3	900	100	86	82
3	999	111	91	86
4	1998	222	195	186

Remark III.4 (irregular graphs) We now discuss two extensions to irregular graphs.

The BIT-FILLING algorithm guarantees that the bit nodes are all regular. However, the check nodes are not necessarily regular, since, in general, the upper bound b may not be achieved by all of them. We now consider an extension where the bit nodes are also allowed to be irregular. In particular, we are given an upper bound a_u and a lower bound $a_l \geq 2$ on the degree of the bit nodes. In this case, our objective is to find the largest integer n such that we first maximize the number of columns with weight a_u , then maximize the number of columns with weight $a_u - 1$, etc. The extension to achieve this goal is simple: start the BIT-FILLING

algorithm with $a = a_u$ and add as many columns as possible. At some point the algorithm will fail in line 13. Now, if $i < a_l$, then the procedure stops. However, if $i \geq a_l$, then reset $a = i$, and continue.

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. The BIT-FILLING algorithm can be easily extended to construct LDPC codes with a prescribed bit-degree distribution with a large girth. Suppose that we want to construct a code with bit-degrees $a_1 > a_2 > \dots > a_r$, where there are n_i bit nodes with degree a_i . We proceed as follows. We fix the girth g and we start allocating the columns with the largest degree, a_1 . After allocating n_1 of these columns, we allocate n_2 columns of degree a_2 , and so on. If we succeed in allocating the $n = \sum_{i=1}^r n_i$ columns, we increase the target girth; otherwise, we decrease it. In this fashion, we search for the largest girth graph that attains the prescribed bit-degree sequence.

Remark III.5 (computational hardness) The formalization that follows was posed to us by Brian Marcus. To analyze the computational complexity of the problem of designing high rate codes satisfying given parameters, we consider the following decision problem.

GIRTH&BI-DEGREE

Instance: (a, b, g, m, n) where a, b, g, m , and n are positive integers and g is even.

Question: Does there exist a $m \times n$ binary matrix H such that H has exactly a ones in each column, at most b ones in each row, and $G(H)$ has girth g .

We take size of the instance to be $a + b + g + m + n$, and not $\log(abgmn)$. Now, given a $m \times n$ binary matrix H , the constraints on the weights of the rows and the columns are easy to check. Furthermore, using roughly $O((n+m)^2)$ operations it is easy to determine the girth of $G(H)$. Hence, the problem GIRTH&BI-DEGREE is clearly in NP. However, the input consists of simply five integers, and does not contain any additional structure. Hence, assuming that $P \neq NP$, it follows from Berman [13] that this problem cannot be NP-hard.

Remark III.6 (computational complexity) We will now roughly analyze the computational complexity of the BIT-FILLING algorithm in Figure 1. Line 7 implements a set difference, and can be implemented using $O(m)$ operations. Line 9 finds the check node c^* using the algorithm in Figure 2, and can be implemented, in the worst case, using $O(m^2)$ operations. Updating U_1 and \mathcal{N}_c on line 10 can both be implemented using $O(a)$ operations, updating A can

be implemented using $O(m)$ operations, and, using (3)–(4), updating U can be implemented using $O((ab)^{g-1}) = O(m)$ operations. Hence, each iteration of the inner “do ... while” loop in lines 6-14 can be implemented in $O(m^2)$ operations. Since, the loop is iterated at most a times, the entire loop in lines 6-14 can be implemented in $O(am^2)$ operations. Finally, the outer “do ... while” loop in lines 3-16 is iterated at most $(mb)/a$ times, and, hence, the entire algorithm requires $O(bm^3)$ operations. Observe that if, in line 9, we use the first-order heuristic algorithm in (5) instead of the entire algorithm in Figure 2, the entire resulting algorithm requires only $O(bm^2)$ operations. Thus, computational complexity of BIT-FILLING is quite favorable, especially, when compared to various exponential-time random construction methods that are often employed currently.

ACKNOWLEDGMENTS

We are grateful to Ajay Dholakia and Brian Marcus for a number of valuable discussions.

REFERENCES

- [1] D. J. C. MacKay, “Encyclopedia of sparse graph codes,” 1999. <http://wol.ra.phy.cam.ac.uk/mackay/codes/data.html>.
- [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. Urbanke, “Design of provably good low-density parity check codes,” *Submitted to IEEE Trans. Inform. Theory*, 1999.
- [5] G. A. Margulis, “Explicit group-theoretical construction of combinatorial schemes and their application to the design of expanders and concentrators,” *Problems of Inform. Transmission*, vol. 24, no. 1, pp. 39–46, 1988.
- [6] A. Lubotzky, R. Phillips, and P. Sarnak, “Explicit expanders and the Ramanujan conjectures,” *Combinatorica*, vol. 8, no. 3, pp. 261–277, 1988.
- [7] D. J. C. MacKay, S. T. Wilson, and M. C. Davey, “Comparison of constructions of irregular Gallager codes,” *IEEE Trans. Communications*, vol. 10, pp. 1449–1454, October 1999.
- [8] D. J. C. MacKay and M. C. Davey, “Evaluation of Gallager codes for short block length and high rate applications,” in *Proc. IMA Workshop Codes, Systems, & Graphical Models*, 1999.
- [9] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman, “Analysis of low-density codes and improved designs using irregular graphs,” in *Proc. 30th Ann. ACM Symp. Theory Computing*, pp. 249–258, 1998.
- [10] J. L. Fan, “Array codes as ldpc codes,” in *Proc. 2nd Int. Symp. Turbo Codes and Related Topics, Brest, France, September 4–7, 2000*.
- [11] T. Mittelholzer, 2000. personal communication.
- [12] Y. Kou, S. Lin, and M. P. Fossorier, “Low density parity check codes based on finite geometries: A rediscovery,” in *Proc. Int. Symp. Inform. Theory, Sorrento, Italy, June 25–29, 2000*.
- [13] P. Berman, “Relationship between density and deterministic complexity of NP-complete languages,” in *Fifth Intl. Colloq. on Automata, Languages, and Programming, Lecture Notes in Computer Science*, vol. 62, pp. 63–71, 1978.