Removing Loops from LDPC Codes

James A. McGowan and Robert C. Williamson

***Abstract*— Small loops in the graphical representations of low-density parity-check codes are hypothesised to be detrimental to the code's performance. In this paper we demonstrate a procedure for removing such loops from the graphs. An algorithm based on the code's adjacency matrix is used to locate any unwanted loops, and then certain edges are exchanged within the graph to eliminate those loops (without simultaneously creating any others). This process can be repeated until there are no loops remaining with a size less than a particular limit. A comparison of the theoretical and experimental values for this limit is given. New codes created in this way are shown to have significantly improved performance, especially at low bit-error rates, and avoid the error floor phenomenon found in the unmodified codes.**

***Index Terms*— Graph theory, low-density parity-check codes, belief propagation, graphical models.**

I. Introduction

Low-density parity-check codes (LDPCs), introduced by Gallager in 1963 [4], are being increasingly studied due to their high performance capabilities [3, 7, 8, 11, 13, 14]. An LDPC code uses a large binary parity-check matrix H to verify codewords. Each row and column of H will have a predetermined number of 1s, which is designed to be a very low fraction of the total number of elements. The columns of H correspond to the bits of a received message x, and the rows correspond to parity-checks on those bits. Thus a correctly received codeword will satisfy the equation

$$Hx \equiv \mathbf{0} \bmod 2. \tag{1}$$

The matrix H can also be conveniently represented by a graph. A node of the graph is associated with each row and each column of H, and an edge connects a row node with a column node if and only if they intersect in H with a 1. This graph is *bipartite* because the edges only connect between the two disjoint sets of check (i.e. row) nodes and variable (i.e. column) nodes. In a bipartite graph any even-length path of edges must finish in the same set it started, and an odd-length path will finish in the opposite set. A *loop* (or *cycle*) is a closed path with no repeated nodes, and must therefore be of even length.

There is at most one edge between any two nodes, and so the shortest length a loop can have is 4, the next largest is 6, and so on. We refer to such loops as 4-loops, 6-loops, etc., and in general a loop of length n is an n*-loop*. The *girth* of a graph is the length of the shortest loop.

The best techniques for LDPC decoding are based on Pearl's belief propagation algorithm [12], which passes probability messages around the graphical network of a code to find the most likely original message. If the graph contains no loops then this decoding is quickly computable. Unfortunately LDPCs have loopy graphs, and so the algorithm needs to be repeatedly iterated until it converges to a solution. The main problem is that in a loop the value of an incorrect bit will propagate back to itself, effectively reinforcing itself and resisting the efforts of the algorithm to correct it. With longer loops this effect is diluted, however, and not as critical to the decoder's performance. LDPC graphs contain many loops of different sizes, and in this paper we investigate a process for removing the shorter loops, which improves the effectiveness of the belief propagation decoder and thus ultimately results in improved performance for the code.

Section II introduces the adjacency matrix of a code, and shows how it can be used to find loops of a desired size. Section III shows a procedure that removes these loops once found, and verifies that no other unwanted loops are created at the same time. Section IV investigates how many loops can be removed from a given graph, and compares the theoretical and experimental results. Section V presents the simulation results, and compares the improved performances of the loop-removed LDPC codes. Conclusions are drawn in Section VI, and the Algorithm is included as an appendix.

II. Loop Detection

Most LDPC generating algorithms include some acknowledgment of the desirability of removing small loops [9, page 22], but none to date have been able to guarantee removal of all loops of a certain length. A standard approach [10] is to search the parity-check matrix H for columns with two 1s in identical positions (thus forming a rectangle of four 1s in the matrix). Then some elements are shuffled around, eliminating the rectangle while preserving the other relevant properties of the matrix. This is equivalent to removing a 4-loop from the graph.

This approach is difficult to expand to 6-loops though, and even more so for longer lengths. An answer has been to completely delete those nodes found to be in unwanted loops [10], but this is a largely ineffective approach that changes important properties of the code as well.

Our first requirement is an algorithm to easily locate loops of any length. We can do this using the adjacency matrix of the code, as follows.

Denote the nodes of the graph as $v_1, v_2, \ldots, v_p$, and define the *adjacency matrix* $A = [a_{ij}]$ to be the $p \times p$ symmetric binary matrix

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is connected to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

The authors are with the Department of Telecommunications Engineering, Research School of Information Sciences and Engineering, The Australian National University, Canberra ACT 0200, Australia, e-mail:{James.McGowan, Bob.Williamson}@anu.edu.au

For an LDPC graph A contains similar information to H. The difference being that every graph node is now represented by both a row and column. A natural ordering of the nodes will give the relationship:

$$A = \begin{pmatrix} \mathbf{0} & H \\ H^\top & \mathbf{0} \end{pmatrix}, \tag{2}$$

showing that A is easily constructable from H. Consider the square of A:

$$A^2 = \begin{pmatrix} HH^\top & \mathbf{0} \\ \mathbf{0} & H^\top H \end{pmatrix}. \tag{3}$$

The entries $a_{ij}^{(2)}$ of A^2 are calculated from the formula

$$a_{ij}^{(2)} = \sum_{k=1}^{p} a_{ik} a_{kj}. \tag{4}$$

Note that this is also the number of paths of length 2 between v_i and v_j, because each path contains those intermediate nodes v_k whose two edges correspond to $a_{ik} = 1$ and $a_{kj} = 1$. This observation can be extended using induction [2] to give

Theorem 1: The (i, j)-entry of A^n is the number of paths of length n from v_i to v_j.

Here the diagonal elements A_{ii}^n are the number of paths of length n starting and finishing at v_i. These include the n-loops through v_i, but also other degenerate loops that repeat nodes and/or backtrack along edges. The following theorem avoids these unwanted cases, and is effective for locating loops.

Theorem 2: In a graph with girth n, two nodes v_i and v_j are directly opposite each other in an n-loop iff

$$A_{ij}^{n/2} \geq 2 \tag{5}$$

$$\text{and} \quad A_{ij}^{(n/2)-2} = 0. \tag{6}$$

Proof: Consider a graph of girth 4, as most random graphs are. In a 4-loop every node v_i is opposite another node v_j, with two paths of length 2 between them. It is these pairs of nodes that we set about finding. Firstly, we require that there are at least two paths between v_i and v_j, and so

$$A_{ij}^2 \geq 2. \tag{7}$$

The only cases where this inequality is true and v_i and v_j are not in a 4-loop is when the paths backtrack along themselves, ie., when $v_i = v_j$ or $i = j$. Thus all non-diagonal elements of A^2 with value at least 2 will correspond to a 4-loop. This is equivalent to equation 6, because when $n = 4$ we have

$$A_{ij}^{4/2-2} = A_{ij}^0 = I_{ij} = 0 \tag{8}$$

Once found, these loops can be removed using the method explained below, and we then have a graph with girth 6.

By induction we can now assume that we have a graph with girth n, and wish to locate the n-loops in it. Again we need to find at least two paths of length $n/2$ between v_i and v_j, and so Theorem 1 immediately gives us

$$A_{ij}^{n/2} \geq 2. \tag{9}$$

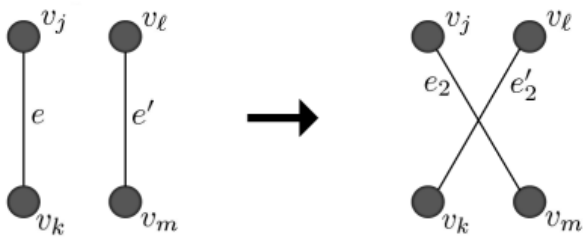

Fig. 1. Replacing the old edges e and e' with the new e_2 and e_2'.

In general any v_i and v_j satisfying (9) are either on a genuine n-loop or on smaller loops that intersect (which cannot be the case here as the graph has girth n), assuming the shortest paths between v_i and v_j have length $n/2$. The next shortest possible length is $(n/2) - 2$, and (6) ensures that none of these occur. ■

A loop located in this way can then be removed, and the process repeated until the girth increases, in which case we start again by removing the now larger loops. Obviously there are multiple pairs able to define each loop (an n-loop has $n/2$ opposite pairs), but any one is sufficient.

III. Loop Removal

Once an unwanted loop is found, the next task is to remove it from the graph. It is not difficult to destroy it by swapping some edges around, but we need to do this in such a way that no new loops are formed.

Firstly we need an edge of the loop. The above detection technique will give us two nodes v_i and v_j. A node v_k adjacent to v_j in the loop will be at distance $n/2 - 1$ from v_i, and so we can take any v_k (the two directions around the loop give two possibilities) that satisfies

$$A_{ik}^{((n/2)-1)} > 0 \quad \text{and} \quad A_{jk} = 1. \tag{10}$$

This gives the loop-edge $e = \overline{v_j v_k}$. We now calculate C_e, the set of all nodes at a distance greater than $n - 1$ from e, which is those nodes v_c for which $A_{cj}^{n-1} = A_{ck}^{n-1} = 0$. An edge e' with both end-nodes in C_e is randomly chosen. If there are no edges with this property then e is not removable, and a different loop-edge will need to be selected.

Let the end-nodes of e' be v_ℓ and v_m. Delete edges e and e' from the graph, and replace them with $e_2 = \overline{v_j v_m}$ and $e_2' = \overline{v_\ell v_k}$, as in Fig. 1.

Theorem 3: Replacing e and e' with e_2 and e_2' will remove the loop that e was part of, and create no new loops of size n or less.

Proof: We know from the definition of C_e that the two new edges did not exist before, so the exchange is possible to do. The old loop is definitely removed, as one of its edges has gone.

In order to confirm that no new n-loops could have been produced, we examine the three cases:

1) A new loop comprising of e_2 and other pre-existing edges would require a path from v_j to v_m of length $n - 1$ or less. But $v_m \in C_e$, and by the definition of C_e this is not possible.

2) Similarly $v_\ell \in C_e$, so there can be no loop containing just e_2'.
3) The other potential way to form a new loop would be if both e_2 and e_2' were included. There is nothing in our selection criteria preventing v_ℓ and v_m being connected by a path of length 3. We know that v_i and v_j were previously on an n-loop however, and so with e removed the shortest path between them must have length $n-1$. Therefore no new loop can be created in this way with a length less than $(n-1)+1+3+1 = n+4$.

■

An explicit statement of the Algorithm suggested by the above is given in the appendix.

IV. Loop Size Limit

The method described in the previous Seciton can be used to remove any undesirable loop from a graph, providing sufficiently distant edges exist. For relatively small loops this is always possible. For larger loops, however, there is a length above which none can be removed.

To find this length theoretically, let the average degree of the variable nodes be λ, and the average check node degree be μ. In this paper we use regular graphs with $\lambda = 3$ and $\mu = 6$, but irregular graphs could equally well have been chosen. By counting outwards from an arbitrary node [1], we obtain

$$N_v \geq \sum_{i=0}^{r-1} (\mu-1)^{\lceil i/2 \rceil} (\lambda-1)^{\lfloor i/2 \rfloor} \tag{11}$$

$$N_c \geq \sum_{i=0}^{r-1} (\lambda-1)^{\lceil i/2 \rceil} (\mu-1)^{\lfloor i/2 \rfloor}, \tag{12}$$

where N_v and N_c are the number of variable and check nodes respectively, and the girth $g = 2r$ [5]. With fixed N_v, N_c, λ and μ it follows that r (and g) must be bounded above.

When $r = 2s$ is even (odd r reaches the same conclusion via a different calculation) we can simplify (11) to get:

$$N_v \geq \sum_{i=0}^{2s-1} (\mu-1)^{\lceil i/2 \rceil} (\lambda-1)^{\lfloor i/2 \rfloor} \tag{13}$$

$$= \sum_{i=0}^{s-1} (\mu-1)^{i} (\lambda-1)^{i} + (\mu-1)^{(i+1)} (\lambda-1)^{i} \tag{14}$$

$$= \sum_{i=0}^{s-1} \mu \left[(\mu-1)(\lambda-1)\right]^{i} \tag{15}$$

$$= \frac{\mu\left(\left[(\mu-1)(\lambda-1)\right]^{s} - 1\right)}{\left[(\mu-1)(\lambda-1)\right] - 1} \tag{16}$$

and rearranging gives

$$s \leq \frac{\log\left(N_v/\mu\left(\left[(\mu-1)(\lambda-1)\right]-1\right)+1\right)}{\log\left[(\mu-1)(\lambda-1)\right]} \tag{17}$$

Substituting our values of $\lambda = 3$ and $\mu = 6$ gives

$$s \leq \log_{10}\left(3N_v/2 + 1\right) \tag{18}$$

$$g \leq 4\log_{10}\left(3N_v/2\right), \tag{19}$$

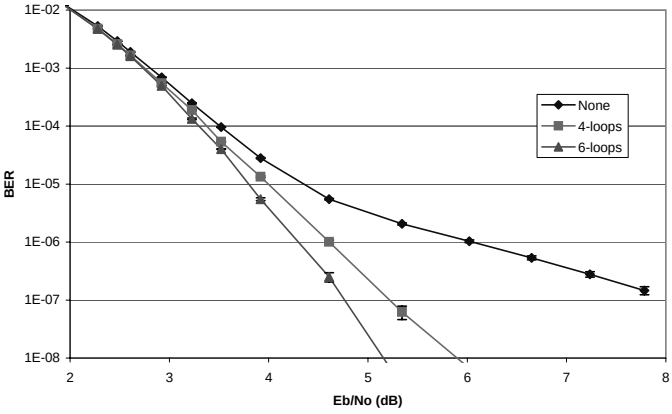

Fig. 2. The average coding performances for random LDPC codes when decoded by belief propagation with $N_v = 300$, $R = 1/2$ for the BIAWGNC. The codes are unmodified for the top line, have 4-loops removed for the middle line, and 4- and 6-loops removed for the bottom line. The error bars, only visible for low BERs, represent 95% confidence intervals.

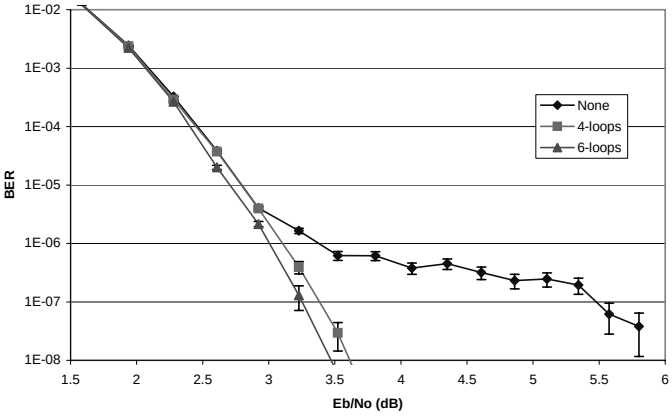

Fig. 3. As per Fig. 2 except with $N_v = 1000$, $R = 1/2$.

This upper bound for g applies to all graphs, no matter how extreme their properties. In practice the girths are lower than this (as shown by experiment). The following table shows the minimum N_v required for the loop removal algorithm to return a graph with girth g (an approximate value observed over numerous random trials), and the theoretical maximum girth for a graph this size given by (19):

g	N_v	$g_{\max}(N_v)$
6	40	7.1
8	250	10.3
10	1600	13.5
12	11000	16.9

V. Performance

The algorithm given in the Appendix was coded in Matlab, which is optimized to deal with matrical approaches such as this. Firstly a random LDPC of a certain size was constructed. A simulated message was sent through the binary input white gaussian noise channel (BIAWGNC) at a specific noise level (E_b/N_0) and the message decoded using belief propagation with a maximum of 25 iterations. The simulation was repeated

with 4-loops removed from the code, and then with 6-loops removed as well. This whole procedure was repeated many times (for many different random codes) The resulting average (over all codes) bit-error rates are plotted in Figs. 2 and 3.

Removing loops from LDPCs gave significant improvements in the performance of the codes. Fig. 2 shows the results for a (3,6)-regular LDPC code with $N_v = 300$ message bits. For low signal-to-noise ratio (SNR) values the loop removal had little effect on performance. The subtleties of the loops' effects on belief propagation are irrelevant when the noise level is so high. As the noise level decreased a noticeable difference between the 3 lines emerged. Even at the relatively large bit-error rate (BER) of 10^{-4} a difference of almost 3dB between the modified and unmodifed codes is apparent. And this grows considerably as the BER drops further. To a lesser extent one can also see that removing 6-loops is also superior to only removing 4-loops, and has a gain of around 1dB at the lower BERs.

In Fig. 3 the loop-removal is even more conclusive. Here we used longer codes, with $N_v = 1000$ bits. Again there is little difference between the lines for low SNR values, but once the E_b/N_0 passes 3dB the performances start to diverge. The unmodified codes have a similar-shaped line to before, albeit at a much lower BER. The major difference is the modified codes have a more rapid drop in BER. The relatively flat line for the unmodified codes shows the existence of an error floor. Belief propagation finds certain formations of 4-loops very difficult to deal with, no matter how large the code, and that is why the removal of 4-loops in particular has such a large effect [9].

VI. Conclusion

This paper introduced a new method for removing loops from LDPC graphs. We used properties of the adjacency matrix to locate unwanted loops. We then found a pair of edges to exchange, essentially removing the loop from the graph. This was repeated until the loops reached a certain size, at which point they were no longer removable. A theoretical analysis of this limit was shown to be a little higher than the results achieved in practice. Simulations showed that removing 4-loops from an LDPC gives a substantial increase in performance. Further improvements are seen after removing 6-loops as well. The size of removable loops increases with the code length, and we expect that longer codes will have even better potential gains from utilizing this algorithm. A similar approach could also be used to remove small loops from the interleavers in turbocodes, thereby improving the belief propagation decoding of them too.

Acknowledgement

The authors would like to thank Prof. Brendan McKay for a helpful discussion on aspects of graph theory.

Appendix

There are various ways to optimize this algorithm. By calculating and storing the powers of A in advance, they don't need to be recalculated every time. The highest power is A^{n-1}, but this lengthy calculation is not required as only two lines of it are used. In fact these lines can be initially added together to give a single vector, and then repeatedly multiplied with A to significantly reduce the complexity.

Also, after two edges are swapped only minor alterations are needed to A, and consequently to the higher powers too.

Algorithm 1 Removing all possible loops from a graph G

$n \Leftarrow 4$
repeat
  $A \Leftarrow$ the Adjacency Matrix of G (from eqn (2))
  for all elements (i, j) of A **do**
    if $A_{ij}^{n/2} \geq 2$ and $A_{ij}^{(n/2)-2} = 0$ **then** {*}
      Choose a k such that $A_{ik}^{(n/2)-1} > 0$ and $A_{kj} = 1$
      $C_e \Leftarrow [j$th row of A^{n-1} + kth row of $A^{n-1} = 0]$
      $E_e \Leftarrow$ all edges of G with endpoints in C_e
      if $E_e \neq \phi$ **then**
        Choose an $e' \in E_e$
        Swap the check nodes of e and e' in G
        Return to **repeat**
      end if
    end if
  end for
  if no elements (i, j) satisfied {*} **then**
    $n \Leftarrow n + 2$
  end if
until no edges were swapped since last repeat

References

[1] B. Bollobas and E. Szemeredi, "Girth of Sparse Graphs," *Journal of Graph Theory*, vol. 39, pp. 194–200, 2002.
[2] F. Buckley and F. Harary, *Distance in Graphs.* Addison-Wesley, Redwood, California, 1990.
[3] S.-Y. Chung, T. J. Richardson, and R. L. Urbanke, Analysis of Sum-Product Decoding of Low-Density Parity-Check Codes using a Gaussian Approximation, *IEEE Trans. Inform. Theory*, vol. 47, pp. 657-670, Feb 2001.
[4] R. G. Gallager, *Low Density Parity Check Codes*. Cambridge, MA: MIT Press, 1963.
[5] S. Hoory, "The Size of Bipartite Graphs with a Given Girth," to appear in *Journal of Combinatorial Theory - Series B.*
[6] R. Kannan, P. Tetali, and S. Vempala, "Simple Markov chain algorithms for generating bipartite graphs and tournaments," in *8th Annual Symposium on Discrete Algorithms*, ACM–SIAM, pp. 193–200, San Francisco, California, 1997.
[7] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Analysis of Low Density Codes and Improved Designs Using Irregular Graphs," *Proceedings of the 30th ACM Symposium on Theory of Computing*, pp. 249–258, 1998.
[8] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Improved Low-Density Parity-Check Codes Using Irregular Graphs," *IEEE Trans. Inform. Theory*, vol. 47, pp. 585–598, Feb 2001.
[9] D. J. C. MacKay, "Good Error Correcting Codes based on Very Sparse Matrices," *IEEE Trans. Inform. Theory*, vol. 45, pp. 399–431, March 1999.
[10] D. J. C. MacKay and R. M. Neal, "Near shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 33, pp. 457–458, March 1997.
[11] G. Miller and D. Burshtein, Bounds on the maximum-likelihood decoding error probability of low-density parity-check codes, *IEEE Trans. Inform. Theory*, vol. 47, pp. 2696–2710, Nov 2001.
[12] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* San Francisco, CA: Morgan Kaufmann, 1988.
[13] T. Richardson and R. Urbanke, The capacity of low-density parity check codes under message-passing decoding, *IEEE Trans. Inform. Theory*, vol. 47, pp. 599-618, Feb 2001.
[14] T. Richardson, A. Shokrollahi, and R. Urbanke, Design of capacity-approaching irregular low-density parity-check codes, *IEEE Trans. Inform. Theory*, vol. 47, pp. 619-637, Feb 2001.