An Algorithm for Counting Short Cycles in Bipartite Graphs

Thomas R. Halford, Student Member, IEEE, and Keith M. Chugg, Member, IEEE

Abstract—Let \( G = (U \cup W, E) \) be a bipartite graph with disjoint vertex sets \( U \) and \( W \), edge set \( E \), and girth \( g \). This correspondence presents an algorithm for counting the number of cycles of length \( g, g+2, \) and \( g+4 \) incident upon every vertex in \( U \cup W \). The proposed cycle counting algorithm consists of integer matrix operations and its complexity grows as \( \mathcal{O}(gt^3) \) where \( t = \max(|U|, |W|) \).

Index Terms—Bipartite graphs, cycles, girth, graphical models of codes, loops.

I. INTRODUCTION


Section IV presents the proof technique used to solve said recursion. Section V presents the specific expressions required to count cycles of length \( g, g+2 \) and \( g+4 \) in bipartite graphs with girth \( g \). Section VI presents a number of examples of the counting algorithm. The complexity of the proposed algorithm is discussed in Section VII. A sample of the proofs of the expressions presented in Section V is appended. The cycle counting technique presented in this correspondence was described previously in [1].

II. REVIEW OF GRAPH THEORY AND NOTATION

Throughout this correspondence, the shorthand \([a, b]\) is used to abbreviate the set

\[
\{a, a+1, \ldots, b-1, b\} \subseteq \mathbb{Z}
\]

where \( a < b \in \mathbb{Z} \) (the set of integers). Similarly, \([c, d]\) and \([e, f]\) are used to abbreviate sets of consecutive even and odd integers, respectively, where \( c < d \in 2\mathbb{Z} \) and \( e < f \in 2\mathbb{Z} + 1 \).

A. Graph Theory

A graph \( G(V, E) \) consists of a finite nonempty set of vertices \( V \) and a set of edges \( E \), which is any subset of the pairs

\[
\{(u, v) : u, v \in V, u \neq v\}
\]

A walk of length \( n \) in \( G(V, E) \) is a sequence of vertices \( \alpha_1, \alpha_2, \ldots, \alpha_{n+1} \) such that \( \alpha_i \in V \) and \( \{\alpha_i, \alpha_{i+1}\} \in E \) for all \( i \in [1, n] \). A walk of length \( n \) is termed closed if \( \alpha_1 = \alpha_{n+1} \); that is, if \( \alpha_1 \) and \( \alpha_{n+1} \) correspond to the same vertex \( v \in V \). A walk is a trail if the edges that form the walk are distinct; a trail is a path if the vertices are distinct. A closed path of length \( n \) is a cycle if \( \alpha_1, \alpha_2, \ldots, \alpha_n \) are distinct; that is, if \( \alpha_1 \neq \alpha_j \) for all \( i \neq j \in [1, n] \). The lengths of the shortest and longest cycles of a graph are respectively denoted its girth \( g \) and circumference \( c \) [17]. A graph with no cycles is a tree and has \( g = c = \infty \). This work introduces the term \((m, n-m)\)-lollipop walk to describe length \( n \) walks where \( \alpha_1, \alpha_2, \ldots, \alpha_m \) are distinct and \( \alpha_{n+1} = \alpha_{m+1} \) for some \( m \in [1, n] \). Cycles of length \( 2m \) are thus \((0, 2m)\)-lollipop walks. Figs. 1 and 2 illustrate a \((4, 2)\)-lollipop walk and a \((2, 4)\)-lollipop walk, respectively.

A graph \( G(V, E) \) is bipartite with vertex classes \( U = \{u_i\}_{i=1}^{l} \) and \( W = \{w_j\}_{j=1}^{m} \) if \( V = U \cup W \) and \( U \cap W = \emptyset \) and each edge joins a vertex in \( U \) to one in \( W \). All cycles in bipartite graphs contain an even number of edges. Consequently, all \((m, n-m)\)-lollipop walks in a bipartite graph satisfy \( n - m \equiv 0 \bmod 2 \). Associated with the bipartite graph \( G(U \cup W, E) \) is the \([l \times m]\) edge matrix \( E = [e_{ij}] \) where \( e_{ij} = 1 \) if \( (u_i, w_j) \in E \) and 0 otherwise.

Finally, the number of cycles of length \( 2m \) in a bipartite graph is denoted \( N_{2m} \).

Fig. 1. A \((4, 2)\)-lollipop walk where \( \alpha_7 = \alpha_5 \) while \( \alpha_1, \alpha_2, \ldots, \alpha_6 \) are distinct.

\[ \begin{array}{cccccccc}
\alpha_1 & \rightarrow & \alpha_2 & \rightarrow & \alpha_3 & \rightarrow & \alpha_4 & \rightarrow & \alpha_5 \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
\alpha_7 & \rightarrow & \alpha_5 & \rightarrow & \alpha_4 & \rightarrow & \alpha_3 & \rightarrow & \alpha_1 \\
\end{array} \]

\[ V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \rightarrow V_7 \]

\[ V_1 \rightarrow V_7 \rightarrow V_6 \rightarrow V_5 \rightarrow V_4 \rightarrow V_3 \rightarrow V_2 \]

0018-9448/$20.00 © 2006 IEEE
Fig. 2. A (2, 4)-lollipop walk where θ = α3 while α1, α2, ..., α6 are distinct.

B. Matrix Notation

The following matrix notation is used throughout this correspondence. Let A and B be matrices. The matrix product is denoted AB while A^T is the transpose of A. If A = [a_{ij}] and B = [b_{ij}] have the same dimensions then the direct (element-wise) product is defined by

\[ C = A \circ B = [c_{ij}], \quad \text{where} \quad c_{ij} = a_{ij}b_{ij}. \]  

(1)

If A is a square matrix then Tr(A) is the trace of A and the \( \mathbb{Z}[a] \) operator is defined by

\[ \mathbb{Z}[A] = A - A \circ I \]  

(2)

where I is the identity. Thus, \( \mathbb{Z}[A] \) is A with the diagonal elements replaced by zeros. The following notation is also employed for arbitrary matrices \( A = [a_{ij}] \) where \( k \in \mathbb{Z} \):

\[ B = \begin{pmatrix} A_k \\ k \end{pmatrix} = [b_{ij}], \quad \text{where} \quad b_{ij} = \frac{a_{ij}}{k!} \\
B = A + k = [b_{ij}], \quad \text{where} \quad b_{ij} = a_{ij} + k \\
B = kA = [b_{ij}], \quad \text{where} \quad b_{ij} = ka_{ij} \\
B = \max(A, k) = [b_{ij}], \quad \text{where} \quad b_{ij} = \max(a_{ij}, k). \]  

(3-6)

III. THE LOLLIPOP RECURSION

This section presents a recursion that transforms the problem of counting paths and thus cycles in a bipartite graph into the problem of counting lollipop walks. Eight matrices that count the number of paths and lollipop walks in a bipartite graph \( G(U \cup \mathcal{W}, E) \) with edge matrix \( E \) must first be defined. Let \( \mathcal{P}_i^{V} \) be a \( |U| \times |W| \) matrix where the \((i, j)\)th element is the number of paths of length \( 2k \) from \( u_i \in U \) to \( w_j \in \mathcal{W} \); similarly define \( \mathcal{P}_i^{W} \). The diagonal elements of \( \mathcal{P}_i^{V} \) and \( \mathcal{P}_i^{W} \) are necessarily zero. Let \( P_{2k}^{V} \) be a \( |U| \times |W| \) matrix where the \((i, j)\)th element is the number of paths of length \( k \) from \( u_i \in U \) to \( w_j \in \mathcal{W} \); similarly define \( P_{2k}^{W} \). Let \( P_{2k}^{V, k, 2k} \) be a \( |U| \times |U| \) matrix where the \((i, j)\)th element is the number of \((2k, 2k-2k)\)-lollipop walks from \( u_i \in U \) to \( w_j \in \mathcal{W} \); similarly define \( P_{2k}^{W, k, 2k} \).

Let \( L_{2k}^{V, k, 2k} \) be \( |U| \times |W| \) matrix where the \((i, j)\)th element is the number of \((2k+1, 2k-2k)\)-lollipop walks from \( u_i \in U \) to \( w_j \in \mathcal{W} \); similarly define \( L_{2k}^{W, k, 2k} \).

Based on the preceding definitions, there is a simple recursion for the path matrices with the initial conditions \( P_0^V = I_1 \), \( P_0^W = I_1 \) and \( L_0^{V, 0, 0} = L_0^{W, 0, 0} = 0 \)

\[ P_{2k}^{V, k, 2k} = P_{2k-1}^{V} - \sum_{i=0}^{k-1} L_{2i}^{V, i, 2i} \]  

(7)

\[ P_{2k}^{W, k, 2k} = P_{2k-1}^{W} - \sum_{i=0}^{k-1} L_{2i}^{W, i, 2i} \]  

(8)

\[ P_{2k}^{V} = P_{2k-1}^{V} E^T - \sum_{i=0}^{k-1} L_{2i}^{V, i, 2i-2} \]  

(9)

\[ P_{2k}^{W} = P_{2k-1}^{W} E^T - \sum_{i=0}^{k-1} L_{2i}^{W, i, 2i-2} \]  

(10)

To prove (7), note that a path of length \( 2k+1, \alpha_1, \alpha_2, ..., \alpha_{2k+2} \) is formed by augmenting a single edge to a path of length \( 2k, \alpha_1, \alpha_2, ..., \alpha_{2k+1} \) such that \( \alpha_{2k+2} \neq \alpha_i \forall i \in [1, 2k + 1] \).

All walks \( \alpha_1, \alpha_2, ..., \alpha_{2k+1}, \alpha_{2k+2} \) where the first \( 2k + 1 \) vertices form a path are counted by \( P_{2k}^{V, k, 2k} \). The product \( P_{2k}^{V, k, 2k} \) thus counts both length-\( 2k + 1 \) paths and all length-\( 2k + 1 \) lollipop walks, hence (7). Equations (8)–(10) are proved in a similar manner. The recursion defined by (7)–(10) transforms the problem of counting paths and cycles to that of counting lollipop walks and is thus denoted the lollipop recursion.

Recall that cycles of length \( 2k \) are \((0, 2k)\)-lollipop walks. Since \((0, 2k)\)-lollipop walks are paths of length \( 2k - 1 \) with an added edge such that \( \alpha_{2k+1} = \alpha_1 \), there is a simple expression for counting the number of cycles of length \( 2k \) passing through each vertex

\[ L_{2k}^{V, 0, 2k} = \left( P_{2k}^{V} E^T \right) \circ I \]  

(11)

\[ L_{2k}^{W, 0, 2k} = \left( P_{2k}^{W} E^T \right) \circ I. \]  

(12)

The total number of cycles of length \( 2k \) is

\[ N_{2k} = \frac{1}{2k} \text{Tr} \left( L_{2k}^{V, 0, 2k} \right) = \frac{1}{2k} \text{Tr} \left( L_{2k}^{W, 0, 2k} \right). \]  

(13)

It is clear from (11) and (12) that in order to count cycles of length \( g, g + 2, \) and \( g + 4 \) in a bipartite graph of girth \( g \), the lollipop recursion must be solved up to and including \( 2k + 1 = g + 3 \). In a bipartite graph of girth \( g \), there are no \((m, n - m)\)-lollipop walks for which \( n - m < g \). Expanding (7)-(10) yields, for \( 0 < k < \frac{g}{2} \)

\[ P_{2k}^{V} = P_{2k-1}^{V} E^T - L_{2k-2}^{V} \]  

(14)

\[ P_{2k}^{W} = P_{2k-1}^{W} E^T - L_{2k-2}^{W} \]  

(15)

\[ P_{2k+1}^{V} = P_{2k}^{V} E^T - L_{2k}^{V} \]  

(16)

\[ P_{2k+1}^{W} = P_{2k}^{W} E^T - L_{2k}^{W} \]  

(17)

and for \( k \geq \frac{g}{2} \)

\[ P_{2k}^{V} = P_{2k-1}^{V} E^T - L_{2k-2}^{V} \]  

(18)

\[ P_{2k}^{W} = P_{2k-1}^{W} E^T - L_{2k-2}^{W} \]  

(19)

\[ P_{2k+1}^{V} = P_{2k}^{V} E^T - L_{2k}^{V} \]  

(20)

\[ P_{2k+1}^{W} = P_{2k}^{W} E^T - L_{2k}^{W} \]  

(21)

\[ P_{2k+2}^{V} = P_{2k+1}^{V} E^T - L_{2k+2}^{V} \]  

(22)

\[ P_{2k+2}^{W} = P_{2k+1}^{W} E^T - L_{2k+2}^{W} \]  

(23)

\[ P_{2k+3}^{V} = P_{2k+2}^{V} E^T - L_{2k+3}^{V} \]  

(24)

\[ P_{2k+3}^{W} = P_{2k+2}^{W} E^T - L_{2k+3}^{W} \]  

(25)
Expressions for the \((m,n-m)\)-lollipop walk matrices in (14)–(25) for which \(m > 0\) are given in Section V. In the following section, a general proof structure for the lollipop matrix equations is detailed.

IV. LOLLIPOP MATRIX PROOF STRUCTURE

Fig. 3 illustrates an \((m,n-m)\)-lollipop walk where \(m\) is even.\(^2\) Throughout the remainder of this correspondence, lollipop walks are illustrated with vertices from class \(U\) represented by white circles and vertices from class \(W\) represented by black circles. The lollipop matrix \(L^W_{(m,n-m)}\) thus counts walks of the form shown in Fig. 3.

The product \(E L^W_{(m,n-m)}\) counts \((m,n-m)\)-lollipop walks but also walks where \(\alpha_1 = \alpha_i\) for \(i \in [3,n-1]\). In order to count \((m,n-m)\)-lollipop walks, the inclusion/exclusion principle that was used to counts paths in (7)–(10) is applied: expressions for walks where \(\alpha_1 = \alpha_i\) for \(i \in [3,n-1]\), are found and subtracted (or excluded) from the product \(E L^W_{(m,n-m)}\).

As an example, consider the case where \(m = 2\) and \(n = 8\) illustrated in Fig. 4. The product \(E L^W_{(1,6)}\) counts both \((2,6)\)-lollipop walks and \((m,n-m)\)-lollipop walks where \(\alpha_1 = \alpha_{25}\) and \(\alpha_1 = \alpha_7\). In order to count \((2,6)\)-lollipop walks, expressions for walks where \(\alpha_1 = \alpha_2\) and \(\alpha_1 = \alpha_5\) are found and then subtracted (or excluded) from the product \(E L^W_{(1,6)}\).

As \(m\) and \(n\) get large, the number of terms that must be excluded from the product \(E L^W_{(m,n-m)}\) also gets large. However, many of these terms are zero as they violate the girth constraint. Returning to the \(m = 2, n = 8\) example illustrated in Fig. 4, if \(g = 6\) then neither walks where \(\alpha_1 = \alpha_5\) nor \(\alpha_1 = \alpha_7\) are possible since these cases require the existence of a length-4 cycle.

V. LOLLIPOP MATRIX EQUATIONS

In the following, only expressions for the \(L^U_{(m,n,m)}\) lollipop matrices are given explicitly. In order to obtain an expression for

\[^2\text{The case where } m \text{ is even is presented without loss of generality; the proposed proof technique also applies to the case where } m \text{ is odd.}\]
VI. EXAMPLES

In this section, the proposed cycle counting algorithm is used to count the short cycles in a number of Tanner graphs.

A. A Golay Code Tanner Graph

Consider the (23, 12) Golay code with systematic parity check matrix shown in (39) at the bottom of the page. The cycle counting algorithm computes $N_4 = 452, N_6 = 10319,$ and $N_8 = 199994$ in the corresponding Tanner graph (which has incidence matrix $E = H_C$).

B. A Small LDPC Tanner Graph

Consider the small LDPC code due to [19] with parity check matrix shown in (40) at the bottom of the page. This code was designed to have a Tanner graph with girth 12. Accordingly, the cycle counting algorithm computes $N_4 = N_6 = N_8 = N_{10} = N_{14} = 0, N_{12} = 28,$ and $N_{10} = 21$ in the corresponding Tanner graph.

C. LDPC Tanner Graphs

In this example, the relationship between cycle structure and performance for two sets of LDPCs is examined. First consider two rate-1/2 Gallager codes with block length 816. Definitions for both codes are due to MacKay [20]. Code A has column weight 3 and corresponds to MacKay’s code 816,3,174 while Code B has column weight 5 and corresponds to MacKay’s code 816,5,5,178. The short cycle structure of the Tanner graphs corresponding to these codes is tabulated in Table I. Codes A and B have the same girth $g = 6$ but Code B has many more short cycles. The effect of these cycles on code performance is dramatic: Code A outperforms Code B by approximately 0.75 dB at a bit-error rate of $10^{-5}$ [20] (on the additive white Gaussian noise (AWGN) channel with binary antipodal signaling).

Next consider two rate-1/2, block length 504 LDPCs constructed using the progressive-edge growth method due to Hu et al. [21]. Codes C and D correspond to MacKay’s codes $PEG_{525}{x_{504}}$ and $PEG_{525}{x_{504}}$, respectively. The short cycle structure of the Tanner graphs corresponding to these codes is tabulated in Table II. Given that the Tanner graph corresponding to Code C has larger girth, one may initially expect it to outperform Code D. However, Code D outperforms Code C by approximately 0.5 dB at a bit-error rate of $10^{-5}$ [21] (on the AWGN channel with binary antipodal signaling). This seeming contradiction is explained by examining the regularity of the respective Tanner graph cycle structures. Table III tabulates the mean $\mu_k$ and standard deviation $\sigma_k$ of the number of cycles of length $k$ incident on each code bit vertex in the Tanner graphs corresponding to Codes C and D. The small $\sigma_k$ values for Code C indicate an extremely regular cycle distribution whereas the $\sigma_k$ values are very large for Code D indicating a much more irregular, or random, cycle distribution. In this example, the negative impact of cycle regularity...
5.03, 4.72, 1.94, and 1.43 s, respectively. Note that the Altivec-enabled
of RAM), the short cycles in Codes A, B, C, and D were counted in
5.03, 4.72, 1.94, and 1.43 s, respectively. Note that the Altivec-enabled
BLAS library was used for fast matrix multiplication [22].

VII. COMPLEXITY

The time complexity of the proposed algorithm is dominated by matrix
multiplication and may be estimated by counting the number of matrix
operations required to compute the number of cycles of length
$g, g + 2,$ and $g + 4$ in a bipartite graph with girth $g > 6$. Carefully
counting the total number of matrix multiplications yields an expres-
sion that is linear in $g$:

$$6g + 37.$$ 

The time complexity of each matrix multiplication grows as $O(n^3)$
where $n = \max \{ |V|, |W| \}$. The time complexity of counting short
cycles thus grows as $O((gn)^3)$. 

The space complexity of the proposed algorithm is dominated by
matrix storage which grows as $O(n^2)$. More specifically, computing
the number of cycles of length $g, g + 2,$ and $g + 4$ in a bipartite graph
with girth $g$ and with vertex class sizes $|V|$ and $|W|$ requires at most

$$11|V|^2 + 11|W|^2 + 21|V||W|$$

storage locations. Note that high bit-width storage locations (e.g.,
64-bit integer or double-precision floating point) must be used in order
to accommodate large dynamic ranges in the matrix entries. Also note
that the space complexity of the proposed algorithm does not grow with
girth.

VIII. NOTES

A technical note containing detailed proofs for (26)–(38) and a
C++ implementation of the proposed algorithm are available at:
http://csi.usc.edu/chugg/tools.

APPENDIX

A. Proof of (33)

$$L_{(g;2)}^U = 2 \left[ EL_{(g-1;2)}^W \right] - \max \left( L_{(g;2)}^W - 1, 0 \right) L_{(g;2)}^U + P_{g-2}^U \circ P_2^U - \delta_{g,1} P_2^U.$$ 

Walks of the form shown in Fig. 5 from $\alpha_1$ to $\alpha_{g+3} = \alpha_{g+1}$
are counted by $L_{(g;2)}^U$. The product $EL_{(g-1;2)}^W$ counts $(g, 2)$-lollipop
walks, walks where $\alpha_1 = \alpha_3$, and walks where $\alpha_1 = \alpha_{g+1}$. Walks
where $\alpha_1 = \alpha_i$ for $i \in [3, g - 1]$, require the existence of a cycle of
length less than $g$ and thus violate the girth constraint.

1) Walks Where $\alpha_1 = \alpha_3$: The product $P_{g-2}^U \circ P_2^U$ counts
walks where $\alpha_1 = \alpha_3$ along with walks where $\alpha_1 = \alpha_3$ and $\alpha_2 = \alpha_4$
and walks where $\alpha_1 = \alpha_3$ and $\alpha_2 = \alpha_{g+2}$. Note that walks where $\alpha_1 = \alpha_3$ and $\alpha_2 = \alpha_j$ for $j \in [3, g - 1]$ violate the girth
constraint. Walks where $\alpha_2 = \alpha_4$ are excluded by instead considering the product
$\max (L_{(g-2;2)}^U - 1, 0) L_{(g-2;2)}^U$. A walk where $\alpha_2 = \alpha_{g+2}$ requires that $\alpha_1$ and $\alpha_{g+1}$ be connected by both a path of length $g - 2$ and a path
of length $2$. The direct product $P_{g-2}^U \circ P_2^U$ counts such walks.

2) Walks Where $\alpha_1 = \alpha_{g+1}$: Eliminating the diagonal elements of
$EL_{(g-1;2)}^W$ with the $\mathbb{Z}_2$ operator removes the $\alpha_1 = \alpha_{g+1}$ walks.

3) The Special Case of $g = 4$: When $g = 4$, the direct product
$P_2^U \circ P_2^U$ includes pairs of identical length-2 paths, This double
counting is remedied by replacing $P_2^U \circ P_2^U$ with $2 \left( \frac{P_2^U}{2} \right)$ or, equivalently, $P_2^U \circ P_2^U = P_2^U$.

B. Proof of (36)

$$L_{(2,g)}^U = 2 \left[ EL_{(g-1;2)}^W \right] - 6 \delta_{g,4} \left( \frac{P_2^U}{3} \right).$$ 

Walks of the form shown in Fig. 6 from $\alpha_1$ to $\alpha_{g+3} = \alpha_3$ are
counted by $L_{(2,g)}^U$. When $g > 4$, the product $EL_{(g;2)}^W$ counts $(2, g)$-lollipop
walks and walks where $\alpha_1 = \alpha_3$. Note that walks where $\alpha_1 = \alpha_i$
for $i \in [3, g - 1]$, violate the girth constraint when $g > 4$. Eliminating the
diagonal elements of $EL_{(g-1;2)}^W$ with the $\mathbb{Z}_2$ operator removes the
$\alpha_1 = \alpha_3$ walks.

When $g = 4$, walks where $\alpha_1 = \alpha_5$ do not violate the girth con-
straint and must also be excluded from the product $EL_{(1;2)}^W$. A walk
where $\alpha_1 = \alpha_5$ requires that $\alpha_1$ and $\alpha_3$ be connected by three different
length-2 paths. The expression $6 \left( \frac{P_2^U}{3} \right)$ counts such walks.

REFERENCES

Bipartite Graphs,” Communication Sciences Institute, USC, Los An-
Communication Theory Workshop, Capri, Italy.
In this paper our aim is to study codes that do not show error floor for the BERs higher than $10^{-12}$ and, at the same time, have a threshold near the Shannon limit. One method to solve the error floor problem is to use an outer code. In this method we use the outer code to reduce the BER. This method slightly increases the complexity of the system. This is specifically undesirable in page-oriented memories where simple and fast decoding algorithms are required. Moreover, using an outer code results in a rate loss; however, the rate loss is usually small. There are also methods for decreasing the error-floor effect for the capacity-approaching codes [2]; however, these methods are sometimes not effective for the BERs required by the storage systems. Depending on the application, the above methods may or may not be suitable. As it will be described in more detail, an alternative option is to use codes with linear minimum distance. These codes also have some desirable properties other than good error floor performance. Thus, in this paper our aim is to study codes with linear minimum distance and to find bounds on their achievable rates.

Performance of Low-Density Parity-Check Codes With Linear Minimum Distance

Hossein Pishro-Nik, Member, IEEE, and Faramarz Fekri, Senior Member, IEEE

Abstract—This correspondence studies the performance of the iterative decoding of low-density parity-check (LDPC) code ensembles that have linear typical minimum distance and stopping set size. We first obtain a lower bound on the achievable rates of these ensembles over memoryless binary-input output-symmetric channels. We improve this bound for the binary erasure channel. We also introduce a method to construct the codes meeting the lower bound for the binary erasure channel. Then, we give upper bounds on the rate of LDPC codes with linear minimum distance when their right degree distribution is fixed. We compare these bounds to the previously derived upper bounds on the rate when there is no restriction on the code ensemble.

Index Terms—Bipartite graphs, erasure channel, error floor, iterative decoding, low-density parity-check (LDPC) codes, minimum distance, performance bound.

I. INTRODUCTION

In some applications, it is necessary to design codes that do not suffer from the error floor problem at the desired bit error rates (BERs), while their rates are close to the channel capacity. For example, in some page-oriented memories, low-density parity-check (LDPC) codes can result in very efficient coding schemes [1]. In these memory systems, we can use large block lengths and thus we get performance close to the Shannon limit. However, BER’s less than $10^{-12}$ are required. Since the storage capacity of the system is directly proportional to the code rate, it is very important that the code rate be close to the capacity of the channel. Thus, we need to design LDPC codes that do not show error floor for the BERs higher than $10^{-12}$, and at the same time, have a threshold near the Shannon limit.

One method to solve the error floor problem is to use an outer code. In this method we use the outer code to reduce the BER. This method slightly increases the complexity of the system. This is specifically undesirable in page-oriented memories where simple and fast decoding algorithms are required. Moreover, using an outer code results in a rate loss; however, the rate loss is usually small. There are also methods for decreasing the error-floor effect for the capacity-approaching codes [2]; however, these methods are sometimes not effective for the BERs required by the storage systems. Depending on the application, the above methods may or may not be suitable. As it will be described in more detail, an alternative option is to use codes with linear minimum distance. These codes also have some desirable properties other than good error floor performance. Thus, in this paper our aim is to study codes with linear minimum distance and to find bounds on their achievable rates.