Message-Passing Algorithms for Counting Short Cycles in a Graph

Mehdi Karimi, *Student Member, IEEE,* and Amir H. Banihashemi, *Senior Member, IEEE*

***Abstract*—A message-passing algorithm for counting short cycles in a graph is presented. For bipartite graphs, which are of particular interest in coding, the algorithm is capable of counting cycles of length $g, g+2, \ldots, 2g-2$, where g is the girth of the graph. For a general (non-bipartite) graph, cycles of length $g, g+1, \ldots, 2g-1$ can be counted. The algorithm is based on performing integer additions and subtractions in the nodes of the graph and passing extrinsic messages to adjacent nodes. The complexity of the proposed algorithm grows as $O(g|E|^2)$, where $|E|$ is the number of edges in the graph. For sparse graphs, the proposed algorithm significantly outperforms the existing algorithms, tailored for counting *short* cycles, in terms of computational complexity and memory requirements. We also discuss a more generic and basic approach of counting short cycles which is based on matrix multiplication, and provide a message-passing interpretation for such an approach. We then demonstrate that an efficient implementation of the matrix multiplication approach has essentially the same complexity as the proposed message-passing algorithm.**

***Index Terms*—Counting cycles in a graph, bipartite graph, girth, short cycles, closed walks, tailless backtrackless closed walks, low-density parity-check (LDPC) codes.**

I. Introduction

GRAPHICAL models are widely used in different branches of science and engineering to represent systems and facilitate the description of inference algorithms. The structure of the graphs consequently plays an important role in the dynamics of the system and the performance of the corresponding algorithms. One important example, which has many applications in areas such as artificial intelligence, signal processing and digital communications, is the *factor graph* representation of systems and the *sum-product* algorithm [20]. Factor graphs are bipartite graphs and the sum-product algorithm is a generic *message-passing* algorithm which operates in a factor graph. One notable application of factor graphs and message-passing algorithms is in channel coding, where widely popular schemes such as *turbo codes* [4] and *low-density parity-check (LDPC) codes* [11] can be considered as specific instances. In particular, a specific instance of a factor graph is a *Tanner graph* [32], which is used to represent an LDPC code. In fact, LDPC codes, which are famous for their capacity-approaching performance on many communication channels, owe their popularity to the good performance of the iterative message-passing algorithms that can decode these codes with relatively low complexity. The low complexity is a consequence of the sparsity of the Tanner graph.

In practical error correction schemes, finite-length codes have to be used. For such codes, the performance of the message-passing algorithms is closely related to the structure of the graph, in general, and its cycles, in particular. In [21], the girth distribution of the Tanner graph was related to the performance of an LDPC code. Numerous publications since have used the cycle structure of the Tanner graph as an important measure of the performance of LDPC codes, with the general belief that for good performance, short cycles should be avoided in the Tanner graph of the code. In [34], Xiao and Banihashemi devised message-passing schedules based on the cycle and closed-walk distributions of the Tanner graph that improved the error correction performance of iterative decoding algorithms compared to the conventional parallel (flooding) schedule. In [14], a code construction, known as progressive edge growth (PEG), was devised to maximize the local girth of the graph in a greedy fashion. Halford and Chugg [12] showed that in addition to the girth, the number and statistics of short cycles are also important performance metrics of the code. In [35], error rates of finite-length LDPC codes were accurately and efficiently estimated by enumerating and testing the subsets of short cycles as error patterns. More recently, it was demonstrated that the majority of the dominant trapping sets of an LDPC code, responsible for the error floor of the code, consist of short cycles [16], [17], [18]. Related to this, Asvadi *et al.* [2] devised cyclic liftings that improve the error floor performance of LDPC codes significantly by breaking the short cycles involved in the dominant trapping sets of the base code. The close relationship between the performance of graph-based coding schemes and the cycle structure of the graph, especially the number of short cycles, motivates the search for efficient algorithms that can count cycles of different length in the graph. In the context of coding, the graph is often bipartite. This includes the Tanner graph of LDPC codes.

Counting and enumerating (finding) cycles in a general graph are both known to be hard problems [10]. Much work has been dedicated to lower the complexity of solving these problems. Two well-known examples of algorithms for cycle enumeration are the Tarjan algorithm [33] and the Johnson algorithm [15]. These algorithms have complexities $O(n|E|(\mathcal{C}+1))$ and $O((n+|E|)(\mathcal{C}+1))$, respectively, where $|E|$ is the number of edges, n is number of nodes, and $\mathcal{C}$ is number of cycles in the graph. It is worth noting that the number of cycles, $\mathcal{C}$, may itself increase exponentially

Paper approved by E. Ayanoglu, the Editor for Communication Theory and Coding Applications of the IEEE Communications Society. Manuscript received July 11, 2012; revised August 7, 2012.

Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada (e-mail: {mkarimi, ahashemi}@sce.carleton.ca).

A preliminary version of this paper was presented at the 2010 IEEE Information Theory Workshop, Cairo, Egypt, Jan. 2010.

Digital Object Identifier 10.1109/TCOMM.2012.100912.120503

with n. Also noteworthy is that it is not straightforward to use the Tarjan and the Johnson algorithms for specifically finding cycles of a given length. Another algorithm for finding cycles is the Bax algorithm [3], which is of time complexity $O(2^n poly(n))$ and storage complexity $O(poly(n))$. This algorithm can be easily adopted to find cycles of a given length in a graph. More recently, Schott and Staples [26], [27], [28] studied the problems of counting and enumerating cycles of a given length k, also referred to as k-cycles, in a graph through *zeon algebra*. By using an adjacency matrix, called *nilpotent adjacency matrix* [24], [25], they demonstrated that k-cycles can be enumerated and counted using the k-th power of this adjacency matrix. Their method for enumerating k-cycles is of time complexity $\mathcal{O}(n^{\omega+k-1})$ and storage complexity $\mathcal{O}(n^2 2^n)$, where $\omega \leq 3$ is the exponent representing the complexity of matrix multiplication, and is shown to be faster than the algorithms of the Tarajan and the Bax [28]. The method proposed in [27] for counting k-cycles in a graph with n vertices has the worst case time complexity of $\mathcal{O}(n^{\omega+1}2^n)$. The average time complexity of the method for counting k-cycles in a random simple (i.e, without parallel edges) graph is $\mathcal{O}(n^{\omega+1}(1+p)^n)$, where p is the edge-existence probability. In the case of sparse graphs, the average complexity of the algorithm for counting k-cycles is $\mathcal{O}(n^4(1+q)^n)$, where q satisfies $e \leq q(\frac{n(n-1)}{2k} - 1)$. It should be however noted that, these average complexities can be very large even for small values of p (or q). For example, for a $(500, 250)$ regular LDPC code with left-degree 3, the average complexity of the algorithm for counting 8-cycles is $\approx 1.3 \times 10^{25}$. Still, the most prohibitive part of Schott and Staples' algorithm is its storage complexity of $\mathcal{O}(n^2 2^n)$, which increases very rapidly with n. In [5], Cash used the *immanants* of the adjacency matrix of the graph to count the k-cycles. Cash's method is prohibitively complex even for counting short cycles. The most computationally demanding part of his method is determining the irreducible character matrix of the symmetric group S_n. This matrix is a $P(n) \times P(n)$ matrix of integers, where $P(n) \approx \frac{\exp(\pi\sqrt{2n/3})}{4n\sqrt{3}}$ (e.g., for $n = 1000$, $P(n) \approx 2.4 \times 10^{31}$). Cash applied his method for counting cycles in a graph with 30 nodes, where it took three weeks to generate the matrix on a desktop computer operating at 1.4 GHz [5].

Another approach to count or to enumerate cycles or closed walks in a graph is by using different types of *zeta functions* [29], [13], [30]. In [29], it is shown that closed walks of length k and k-cycles of a graph can be counted or enumerated through finding the trace of the k-th power of a properly defined adjacency matrix (see also [30]). Horton [13] showed that the girth g of a graph and the number of cycles with length g can be determined from the *Ihara zeta function* of the graph.

Attempts are also made to find efficient algorithms for finding *short* cycles in a graph. Alon *et al.* [1] presented methods for counting short cycles in a general graph. The complexity of their algorithm however is prohibitively high for longer cycles, say beyond 7. Chang and Fu [6] derived an expression for the number of 6-cycles in a graph, by subtracting the number of closed walks that are not 6-cycles from the total number of closed walks of length 6 in the graph. Their expression, however, is limited to only the number of 6-cycles, and involves several summations over the elements of powers (up to 6) of the adjacency matrix. Fan and Xiao [9] presented a method for counting cycles of length $2k, 2 \leq k \leq 5$, in the Tanner graph of LDPC codes. The complexity of their method is $O(m^{k+1})$ where m is the number of the check nodes in the graph. Their method quickly becomes prohibitively complex even for counting cycles as short as 6, particularly in graphs with large m. An algorithm with similar complexity was proposed in [7] for counting only the shortest cycles of a Tanner graph. Halford and Chugg [12] presented a method for counting short cycles of length $g, g+2$ and $g+4$ in bipartite graphs with girth g. The complexity of their method is $O(gn^3)$, where n is the size of the larger set between the two node partitions.

In this paper, we present an algorithm that counts the cycles of length $g, g+2, \ldots, 2g-2$ in a bipartite graph. The algorithm is based on message-passing on the edges of the graph, where the messages are computed at the nodes with integer additions and subtractions. The algorithm can also be applied to general (non-bipartite) graphs to count cycles of length $g, g+1, \ldots, 2g-1$. The complexity of the proposed algorithm is $O(g|E|^2)$, where $|E|$ is the number of edges in the graph. For sparse bipartite graphs, the proposed algorithm can significantly outperform the algorithm of [12] in terms of both computational complexity and memory requirements. As an example, for a regular graph with node degrees 3 and 6 corresponding to an (8000,4000) LDPC code, the proposed algorithm is more than 30 times faster than the method of [12] and requires less memory by a factor of about 600. Conceptually also, the proposed algorithm is much simpler than the algorithm of [12], in which complex matrix equations are involved in the counting process. Noteworthy is also the fact that for graphs with $g \geq 6$, the proposed algorithm is capable of counting short cycles of lengths up to at least the same value as the algorithm of [12] does.

As part of this paper, we also present an interpretation of the matrix multiplication approach of [29] as a message-passing algorithm over a trellis diagram. Moreover, we provide an efficient implementation of the approach which has basically the same complexity as the proposed message-passing algorithm.

The remainder of this paper is organized as follows. Basic definitions and notations are provided in Section II. In Section III, we develop the proposed algorithm and give a simple example. In our presentation, we use bipartite graphs for the sake of simplicity and for the reason that the graphs involved in most coding applications are bipartite. The pseudo code for the algorithm is presented in Section IV. Discussions on complexity and memory requirements and comparisons with the algorithm of [12] and the matrix multiplication approach will follow in Section V. Section VI contains numerical results. Section VII concludes the paper.

II. Definitions and Notations

An undirected Graph $G = (V, E)$ is defined as a set of nodes V and a set of edges E, where E is some subset of the pairs $\{\{u, v\} : u, v \in V, u \neq v\}$. In this definition and without loss of generality in the context of this paper, we exclude loops using the condition $u \neq v$. Parallel edges are also indistinguishable by this definition and are excluded for simplicity. A *walk* of length k in G is a sequence of nodes $v_1, v_2, \dots, v_{k+1}$ in V such that $\{v_i, v_{i+1}\} \in E$ for all $i \in \{1, \dots, k\}$. Equivalently, a walk of length k can be described by the corresponding sequence of k edges. A walk is a *path* if all the nodes $v_1, v_2, \dots, v_k$ are distinct. A walk is called a *closed walk* or a *cycle* if the two end nodes are identical, i.e., if $v_1 = v_{k+1}$ in the previous description. It should be noted that in this definition of a cycle, the nodes v_i, $1 \leq i \leq k$, are not necessarily distinct. Consider a cycle c of length $\ell(c) = k$ represented by the sequence of edges $e_{i_1}, e_{i_2}, \dots, e_{i_k}$. The cycle c is *backtrackless* if $e_{i_s} \neq e_{i_{s+1}}$ for any $s \in \{1, \dots, k-1\}$. The cycle c is *tailless* if $e_{i_1} \neq e_{i_k}$. The cycle c is called *primitive* if c is not obtained by going $r > 1$ times around some other cycle b (i.e., $c \neq b^r$). Two cycles of the same length are *equivalent* if both have the same set of edges and nodes and one is obtained by changing the starting node of the other one. A cycle c is called *simple* if all the nodes v_i, $1 \leq i \leq k$, are distinct. Clearly, every simple cycle is tailless and backtrackless, but the reverse is not necessarily true. In this paper, we are mainly interested in simple cycles unless specified otherwise. We also refer to "simple cycles" as just "cycles" in the rest of the paper. We also use the term *tbc walk* to refer to a tailless backtrackless closed walk.

In a graph G, cycles of length k, also referred to as *k-cycles*, are denoted by $\mathcal{C}_k$. We use N_k for $|\mathcal{C}_k|$, where equivalent cycles are counted only once. To each undirected walk (cycle), we associate two directed walks (cycles), depending on which end node or edge is selected as the starting point. This concept is important in the description of the proposed algorithm since the direction of edges is of consequence in message-passing algorithms.

A graph $G(V, E)$ is called *bipartite* if the set V can be partitioned into two disjoint subsets U and W ($V = U \cup W$ and $U \cap W = \emptyset$) such that every edge in E connects a node from U to a node from W. We denote $|U|$ by n and $|W|$ by m. Tanner graphs of LDPC codes are bipartite graphs, in which U and W are referred to as *variable nodes* and *check nodes*, respectively. Parameters n and m in this case are the code block length and the number of parity check equations, respectively.[1]

The *girth* g of a graph is the length of a shortest cycle in the graph. For bipartite graphs, all cycles have even lengths and g is an even number. The number of edges connected to a node v is called the *degree* of v, and is denoted by d_v. We call a bipartite graph $G = (U \cup W, E)$ *regular* if all the nodes in U have the same degree d_u and all the nodes in W have the same degree d_w. Otherwise, the graph is called *irregular*. For a regular graph, it is easy to see $nd_u = md_w = |E|$.

The *adjacency matrix* of a graph G is the matrix $A = (a_{ij})$, where a_{ij} is the number of edges connecting node i to node j for all $i, j \in V$. Matrix A is symmetric and since we have assumed that G has no parallel edges or loops, $a_{ij} \in \{0, 1\}$ for all i, $j \in V$, and $a_{ii} = 0$ for all $i \in V$. One important property of the adjacency matrix is that the number of walks between any two nodes of the graph can be easily determined using the powers of this matrix. More precisely, the entry in the i^{th} row and the j^{th} column of A^k, $(A^k)_{ij}$, is the number of walks of length k between nodes i and j. In particular, $(A^k)_{ii}$ is the number of closed walks of length k containing node i, and $tr(A^k)$ is the total number of closed walks of length k, where $tr(\cdot)$ denotes the trace of a matrix.

[1]For LDPC codes, the biadjacency matrix of the Tanner graph is the parity check matrix H of the code. Using H, one can thus obtain the Tanner graph, and implement the message-passing algorithm proposed here.

For a graph $G = (V, E)$, the *(Hashimato) directed edge matrix*, denoted by A_e, is a $2|E| \times 2|E|$ matrix defined as follows. For every edge $e_i \in E$, consider two directed edges with opposite directions and denote them by f_i and $f_{|E|+i}$. The entry $(A_e)_{ij}$ is defined as [29]

$$(A_e)_{ij} = \begin{cases} 1 & \text{if edge } f_i \text{ feeds into edge } f_j \text{ with no backtracking,} \\ 0 & \text{otherwise.} \end{cases}$$

In particular, $(A_e)_{ij} = 0$, for $j = i + |E|, i = 1, \dots, |E|$, and for $j = i - |E|, i = |E| + 1, \dots, 2|E|$. In general, the matrix A_e can be represented as

$$A_e = \begin{pmatrix} A & B \\ C & D \end{pmatrix},$$

where A, B, C and D are $|E| \times |E|$ matrices with the following properties:

a. $B = B^T$, and $C = C^T$,
b. $D = A^T$,
c. $b_{ii} = c_{ii} = 0$, $\forall i \in \{1, \dots, |E|\}$.

One important property of the directed edge matrix is that $tr(A_e^k)$ is the number of tbc walks of length k in the graph.

III. Main Ideas of the Proposed Algorithm

A. Message Passing

A message-passing algorithm operates in a graph by computing messages at the nodes and passing them along the edges to the adjacent nodes. A well-known example is the sum-product algorithm operating in a factor graph [20]. Message passing algorithms often have the property that a message sent along an edge e is not a function of the message previously received along e. We refer to this property as *extrinsic* message-passing. An example is shown in Fig. 1, where the operation at node v_1 is multiplication. Extrinsic message-passing, for example, is known to be an important property of good iterative decoders [23]. The algorithm proposed in this paper also has this property.

For bipartite graphs $G(U \cup W, E)$, a natural message-passing schedule is for every node in U to send messages to adjacent nodes in W followed by every node in W to send messages to adjacent nodes in U. This is referred to as *parallel schedule* and is used often in iterative decoding algorithms. In this case, a complete cycle of message-passing from U to W and then from W to U is called one *iteration*. We assign discrete time t to message-passing, starting from time index zero followed by positive integer values. Corresponding to a time index $t \geq 0$, we associate an iteration number $\ell = \lfloor t/2 \rfloor + 1 \geq 1$. The time indices $t = 2\ell - 2$ and $t = 2\ell - 1$ correspond to the first and the second halves of the iteration ℓ. We also refer to messages passed at $t = 0$ as *initial messages*, and use the notation $m^{(\ell)}_{u \to w}$ for a message passed from node u to node w at iteration ℓ. The notations $m^{(\ell)}_{u \overset{e}{\leftarrow}}$ and $m^{(\ell)}_{u \overset{e}{\rightarrow}}$ are used for

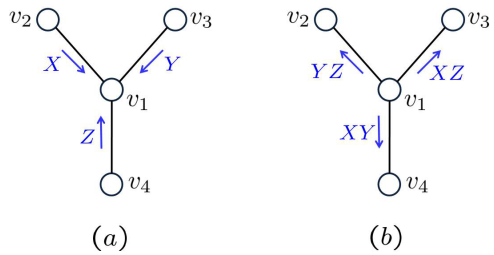

Fig. 1. An extrinsic message-passing algorithm: a) messages received by v_1 at t, b) messages sent by v_1 at $t+1$

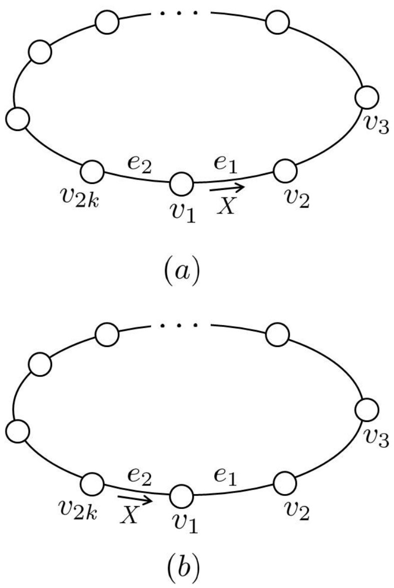

Fig. 2. Message passing for a cycle of length $2k$. a) initial message X is passed along e_1 , b) after k iterations, v_1 receives X along e_2.

the incoming and the outgoing messages to and from node u along edge e at iteration ℓ, respectively.

In the general context of iterative decoding, all nodes in the same partition (U or W) perform the same type of operation to generate their messages. The types of operation however are usually different for the two partitions and depend on the nature of the algorithm and the domain in which the messages are presented. In the algorithm developed in this paper, however, all the nodes perform the same type of operation. The messages are all monomials and the operation is multiplication. An example can be seen in Fig. 1. In this work, a monomial is the product of integer powers of variables. For example, a message $m = X_1^i X_2^j X_3^k$ is a monomial with variables X_1, X_2 and X_3. We say m contains i copies of X_1, j copies of X_2 and k copies of X_3. If the variables are ordered, we may use a simpler representation of m as a vector: $m = (i, j, k)$. Using the vector representation of messages, the multiplication of monomials is reduced to the addition of the corresponding vectors.

B. Algorithm Development

Consider an extrinsic message-passing algorithm in a graph with messages as monomials and node operations as monomial multiplication. In the following, we explain how such an algorithm can count short cycles of the graph. Consider a cycle C of length $2k$ as depicted in Fig. 2(a). Suppose that node v_1 of C passes the monomial X as the initial message at $t = 0$ to v_2. Due to the extrinsic property of message-passing, X will be passed to v_3 from v_2 at $t = 1$ and continues its journey around the cycle, one node at a time, until it reaches back to v_1 at $t = 2k - 1$ and at the end of iteration k, as shown in Fig. 2(b). Clearly, if node v_1 had also passed a monomial Y along the edge e_2 to v_{2k} at $t = 0$, it would have also received Y from v_2 along e_1 at the end of iteration k. So the iteration number at which node v_1 receives back the messages it passed at the first iteration is half the length of the cycle. The following lemma puts this basic idea in the context of the message-passing in a general graph.

Lemma 1: Suppose that C is a cycle of length $2k$ in a bipartite graph $G = (V, E)$, and $v \in V$ is in C. Denote the two adjacent edges of v in C by e_1 and e_2. Assume that the message-passing algorithm is initiated on the side of the graph which includes v by passing 1 along every edge in E, except e_1 and e_2. For e_1 and e_2, the initial messages are monomials X_1 and X_2, respectively. Then, at iteration k, node v will receive one copy of X_2 and one copy of X_1 along e_1 and e_2, respectively, where both copies have traveled through all the edges of C.

Proof: The proof is straightforward and follows directly from the definition of extrinsic message-passing. ■

It is easy to see that if the node v in Lemma 1 is in $N_{2k}^{v;e_1,e_2}$ cycles of length $2k$ which all include e_1 and e_2, then at iteration k, node v will receive $N_{2k}^{v;e_1,e_2}$ copies of X_2 and $N_{2k}^{v;e_1,e_2}$ copies of X_1 along e_1 and e_2, respectively, where each pair of copies has traveled through all the edges of one of the cycles, respectively. Assuming there are no additional copies of X_2 received by v along e_1 and no additional copies of X_1 received by v along e_2 at iteration k, the monomials received at iteration k by v along e_1 and e_2 are respectively $X_2^{N_{2k}^{v;e_1,e_2}}$ and $X_1^{N_{2k}^{v;e_1,e_2}}$.

We note that in addition to copies of X_2 which are received by node v along e_1 at iteration k, v may also receive copies of X_1 along e_1 at iteration k. These correspond to closed walks of length $2k$ which start and end at edge e_1 (and are thus with tails, and clearly not cycles). To eliminate these structures in the counting process of $N_{2k}^{v;e_1,e_2}$, one should consider the power of received variables along e_1 and e_2 excluding the initial message. To describe this, we use the notation $m^{(k)}_{E,v\xleftarrow{e_1}}$ to denote the incoming message to node v along e_1 at iteration k, excluding the variable of the initial message passed by v along e_1. In the above scenario, we have $m^{(k)}_{E,v\xleftarrow{e_1}} = X_2^{N_{2k}^{v;e_1,e_2}}$, and $m^{(k)}_{E,v\xleftarrow{e_2}} = X_1^{N_{2k}^{v;e_1,e_2}}$. This results in

$$N_{2k}^{v;e_1,e_2} = \{\mathrm{ex}(m^{(k)}_{E,v\xleftarrow{e_1}}) + \mathrm{ex}(m^{(k)}_{E,v\xleftarrow{e_2}})\}/2, \qquad (1)$$

where $\mathrm{ex}(\cdot)$ is the exponent of the monomial, defined as the sum of the powers of all its variables.

There is also a possibility that node v receives additional copies of X_2 along e_1 and additional copies of X_1 along e_2 at iteration k. These additional copies travel either through the same cycle multiple times or through non-cycle tbc walks of length $2k$ which start and end at e_1 and e_2, respectively.

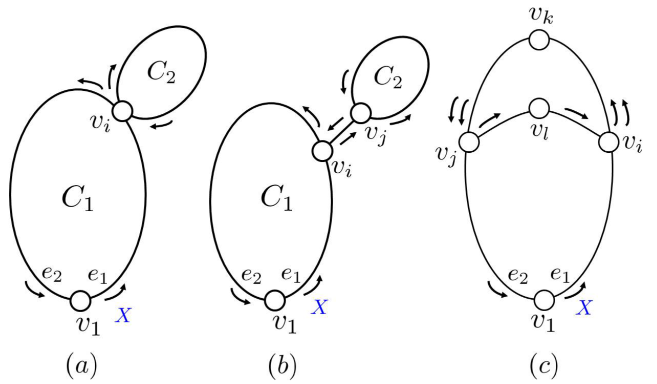

Fig. 3. Three problematic structures for which the incoming extrinsic messages do not represent cycles.

Examples of the latter structures are given in Fig. 3, where the message X is initiated at node v_1. In Fig. 3(a), $2k$ is in fact the sum of the lengths of the two cycles C_1 and C_2, while in Fig. 3(b), it is the sum of the lengths of the two cycles plus twice the length of the path between v_i and v_j. In Fig. 3(c), message X travels from v_1 to v_i first, and then from v_i to v_j through v_k. It then travels back from v_j to v_i through v_l followed by a trip from v_i to v_j through v_k for the second time. The journey finally ends when X is passed back from v_j to v_1. In this case, the total length of the walk is $2k$.

A careful inspection of the problematic structures, as described above, reveals that they all include at least two cycles. This implies that the shortest length of such structures is $2g$, where g is the girth of the graph. We thus have the following results.

Lemma 2: Every tbc walk of length less than $2g$ in a graph of girth g is a (simple) cycle.

Lemma 3: Consider a bipartite graph $G = (V, E)$ with girth g. Select a node $v \in V$ with two adjacent edges e_1 and e_2. Assume that the message-passing algorithm is initiated at $t = 0$ by passing 1 along every edge in E, except e_1 and e_2. For e_1 and e_2 the initial messages are set to monomials X_1 and X_2, respectively. Then, at iteration $k, k < g/2$, node v will only receive 1 along all its edges including e_1 and e_2. At iteration $k, g/2 \le k \le g-1$, node v will receive monomials $X_1^i X_2^{N_{2k}^{v;e_1,e_2}}$ and $X_1^{N_{2k}^{v;e_1,e_2}} X_2^j$ along e_1 and e_2, respectively, where i and j are non-negative integers. Equation (1) is thus valid for $k \le g-1$.

Proof: Node v will receive messages other than 1 only if a copy of X_1 or X_2 is passed back to it. Due to the extrinsic nature of message-passing, such a copy must travel through a backtrackless closed walk with both ends at v. Since the length of a backtrackless closed walk is at least g, no messages other than 1 will be received by v at iterations $k, k < g/2$. At iterations $k, g/2 \le k \le g-1$, node v can receive copies of X_1 and X_2 that have traveled through backtrackless closed walks with both ends at v. In particular, the number of copies of X_1 and X_2 that v receives at iteration $k \ge g/2$, along e_2 and e_1, respectively, is equal to the number of tbc walks of length $2k$ that start and end at e_1 and e_2. For k in the range $g/2 \le k \le g-1$, based on Lemma 2, such tbc walks are limited to cycles of length $2k$ that include e_1 and e_2. (For $k \ge g$, in addition to cycles, they can include multiple trips over the same cycle or cases such as those in Fig 3.) ∎

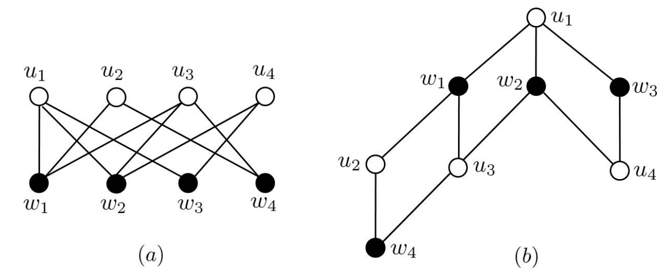

Fig. 4. Bipartite graph of the example in Section III.C: a) G, b) G unwound from node u_1.

Let us now focus on the problem of counting *all* the cycles of a certain length $2k$, $k < g/2$, which pass through a certain node v in a bipartite graph $G = (U \cup W, E)$. Without loss of generality, we assume $v \in U$. One approach to count all the $2k$-cycles containing v is to use Lemma 2 and count the cycles involving different adjacent edges, two at a time, and then add up the results. The following lemma however suggests a more efficient approach.

Lemma 4: Consider a bipartite graph $G = (U \cup W, E)$ with girth g, and a node $v \in U$. Initiate the message-passing algorithm by passing 1 on all the edges connected to nodes $u \in U$, $u \ne v$, while passing d_v different monomials, say $X_1, X_2, \ldots, X_{d_v}$, along the edges connected to $v : e_1, \ldots, e_{d_v}$, respectively. For $k \le g-1$, we then have

$$N_{2k}^v = \sum_{j=1}^{d_v} \text{ex}(m^{(k)}_{E,v \overset{e_j}{\leftarrow}})/2 \,, \tag{2}$$

where N_{2k}^v is the number of $2k$-cycles containing v.

Proof: At iteration $k \le g-1$, consider the message received by v along $e_j, j = 1, \ldots, d_v$, excluding the variable X_j. In this extrinsic message $m^{(k)}_{E,v \overset{e_j}{\leftarrow}}$, the power of variable $X_i, i \ne j$, is $N_{2k}^{v;e_i,e_j}$. We therefore have

$$\text{ex}(m^{(k)}_{E,v \overset{e_j}{\leftarrow}}) = \sum_{\substack{i=1 \\ i \ne j}}^{d_v} N_{2k}^{v;e_i,e_j} \,.$$

This combined with

$$N_{2k}^v = \frac{1}{2} \sum_{j=1}^{d_v} \sum_{\substack{i=1 \\ i \ne j}}^{d_v} N_{2k}^{v;e_i,e_j} \,,$$

completes the proof. ∎

In Lemma 4, at iteration $k, k < g/2$, node v will only receive 1 along all its edges, indicating there are no cycles of length $g-2$ or smaller containing v.

It is worth noting that the message-passing algorithm can be simplified by allowing node v to always pass 1 after the first iteration. This is demonstrated in the following example.

C. A Simple Example

Here, we illustrate the proposed method by a simple example. Consider the bipartite graph G shown in Fig. 4(a), where the nodes in U and W are represented by hollow and full circles, respectively. Suppose that we are interested in

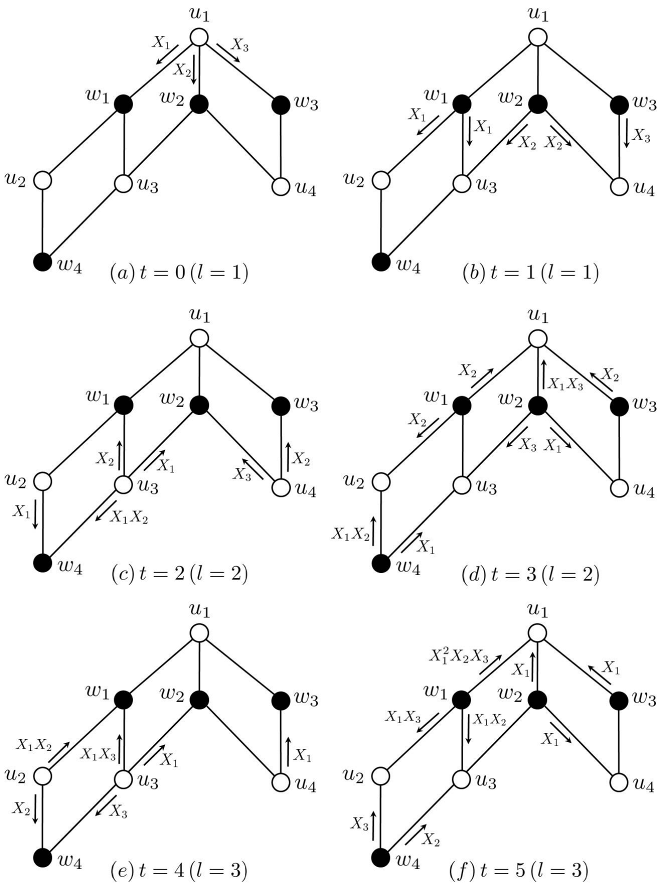

Fig. 5. Message passing of the proposed algorithm for three iterations in the graph of Fig. 4.

counting short cycles containing node u_1. For the simplicity of presentation, as shown in Fig. 4(b), we can unwind the graph G from node u_1. It is easy to see from Fig. 4(b) that the girth of G is 4. Using the proposed method, we can thus count cycles of length up to $2g-2=6$. The message-passing algorithm is illustrated in Figures 5(a)-(f):

(a) At $t=0$, the algorithm is initiated by node u_1 passing messages X_1, X_2, and X_3 along its 3 edges. All the other messages sent by nodes u_2, u_3 and u_4 along their edges are equal to 1, and not shown. [Equivalently, in the vector representation, the initial messages of node u_1 are vectors $(1,0,0), (0,1,0)$ and $(0,0,1)$, while all the other messages are $(0,0,0)$.]

(b) At $t=1$, only the nodes in W are active. The corresponding (non-one) messages are shown in Fig. 5(b). Note that in this iteration ($\ell=1$), all the incoming messages to node u_1 are equal to one.

(c) At $t=2$, nodes in U are active. They all pass extrinsic messages using multiplication. For example, $m^{(2)}_{u_3\to w_4} = m^{(1)}_{w_1\to u_3} \times m^{(1)}_{w_2\to u_3} = X_1X_2$. [In the vector representation, $m^{(2)}_{u_3\to w_4} = (1,0,0)+(0,1,0)=(1,1,0)$.]

(d) At $t=3$ ($\ell=2$), for the first time node u_1 receives non-one messages, an indication that there is at least one cycle of length $2\ell=4$ containing u_1. Using (2), we obtain $N_4^{u_1} = (1+2+1)/2 = 2$.

(e) At $t=4$, the nodes in U are active and pass messages.

(f) At $t=5$ ($\ell=3$), nodes in W are active. Again in this iteration, node u_1 receives non-one messages, an indication that it belongs to at least one 6-cycle. Using (2), we have $N_6^{u_1} = (2+1+1)/2 = 2$.

IV. Proposed Message-Passing Algorithm

A. Pseudo Code

To count the short cycles of a certain length $2k$ in the whole graph $G=(U\cup W, E)$, one can apply the proposed algorithm described in the previous section to every node in one of the node partitions, U or W, and then add up the results for each cycle length. In this case, for each cycle length, the result should be divided by k as every cycle is counted k times:

$$N_{2k} = (\sum_{u\in U} N^u_{2k})/k = (\sum_{w\in W} N^w_{2k})/k\ , \frac{g}{2} \le k \le g-1\ . \quad (3)$$

To simplify the algorithm and to avoid the k-fold counting repetition, we can deactivate a node as soon as its cycles are counted. This would be equivalent to removing the node and all its adjacent edges from the graph. Moreover, the algorithm can be further simplified by only activating nodes that have at least one non-one incoming message. Based on these simplifications, the proposed algorithm has the pseudo code provided in Algorithm 1.

Algorithm 1 is initiated from U. Similarly, it can be initiated from W. Nodes in U are indexed by $i=1,\ldots,n$, and notation $m_{E,(w_j\to u_i)}$ is used to denote the incoming message from node w_j to node u_i excluding the initial variable passed from u_i to w_j. Notation $N(u)$ is used for the nodes adjacent to u (neighbors of u).

Here we have implicitly assumed that the girth g of the graph is known. In the following subsection, we discuss a modification of the algorithm that can compute g and N_g.

B. Parallel Implementation

The algorithm presented in the previous subsection is based on sequentially going through the nodes in one of the two partitions in the graph. To speed up the counting process and at the expense of larger memory usage, one can run a parallel version of the algorithm in which all the nodes in one partition are initialized simultaneously. This is explained in Fig. 6(a) for the graph of Fig. 4.

The parallel implementation, just described, can also be used to compute g and N_g. To see this, note that in the parallel implementation, none of the nodes in the initiating partition will receive a copy of its initial messages before iteration $g/2$. At iteration $g/2$, the nodes which are contained in the shortest cycles will receive copies of their initial messages and all such copies are received along the edges whose initial messages differ from the received messages. This means that all the received copies represent true g-cycles. Therefore to compute g and N_g, one does not need to distinguish among the initial messages of a node. The initialization in this case is explained in Fig. 6(b) for the graph of Fig. 4. In this setup, if the first iteration in which at least one of the nodes receives a non-one message is iteration k, then $g=2k$, and the number of g-cycles is equal to the total number of received non-one messages by all the nodes divided by $2k$.

Algorithm 1 Proposed Message-Passing Algorithm for Counting Short Cycles

for $k = 1 : g-1$ **do**
  $counter(k) = 0$
end for
for $i = 1 : n$ **do**
  Initialization
  $l = 1$
  for $w_j \in N(u_i)$ **do**
    $m^{(0)}_{u_i \to w_j} = X_l$
    $l = l + 1$
  end for
  for $i' = i+1 : n$ **do**
    for $w_j \in N(u_{i'})$ **do**
      $m^{(0)}_{u_{i'} \to w_j} = 1$
    end for
  end for

  for $k = 1 : g-1$ **do**
    Message Passing from W
    for $j = 1 : m$ **do**
      for $u_{i'} \in N(w_j)$ **do**
        $m^{(2k-1)}_{w_j \to u_{i'}} = \prod_{u_h \in N(w_j),\, h \geq i,\, h \neq i'} m^{(2k-2)}_{u_h \to w_j}$
      end for
    end for

    Counting Cycles
    $local\text{-}counter(k) = \sum_{w_j \in N(u_i)} \mathbf{ex}(m^{(2k-1)}_{E,(w_j \to u_i)})$

    Message Passing from U
    for $i' = i+1 : n$ **do**
      for $w_j \in N(u_{i'})$ **do**
        $m^{(2k)}_{u_{i'} \to w_j} = \prod_{w_h \in N(u_{i'}),\, h \neq j} m^{(2k-1)}_{w_h \to u_{i'}}$
      end for
    end for

  end for

  for $k = 1 : g-1$ **do**
    $counter(k) = counter(k) + local\text{-}counter(k)/2$
  end for
end for

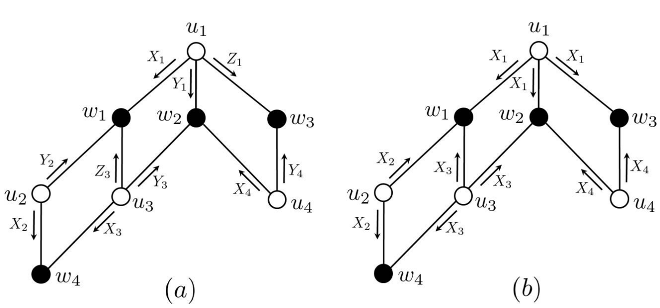

Fig. 6. Initial message-passing in parallel implementations: a) counting short cycles of length $2k$, $g/2 \leq k \leq g-1$, b) calculating g and N_g.

V. Complexity of the Proposed Algorithm

A. Computational Complexity

In the following, we arbitrarily assume that the algorithm is initiated from the node set U. We consider a sequential implementation, where the nodes in U are processed one at a time. We also consider the vector representation of messages and first derive the complexity for a regular graph. We then generalize the results to irregular graphs. For a regular graph $G = (U \cup W, E)$, starting from a node $u \in U$, there are d_u initial messages, each represented by a unit vector of length d_u. All the subsequent messages are also vectors of length d_u. To calculate the messages at an active node $w \in W$, we first add all the incoming vectors to w, and then subtract from this, the incoming message along each adjacent edge to obtain the outgoing message along that edge. This requires $(2d_w - 1)d_u$ integer additions and subtractions. Similarly, for each active node $u \in U$, we need $(2d_u - 1)d_u$ integer additions and subtractions to obtain the outgoing messages. Considering that in even and odd time instances, the number of active nodes are upper bounded by n and m, respectively, the number of operations per iteration is $O(nd_u^2 + md_ud_v) = O(|E|d_u)$. Since the algorithm needs to perform $g-1$ iterations, the complexity of the algorithm for each node $u \in U$ is $O(gnd_u^2 + gmd_ud_v) = O(g|E|d_u)$. The total complexity is thus

$$O(gn^2d_u^2 + gnmd_ud_v) = O(gn^2d_u^2) = O(g|E|^2) \, .$$

It is easy to see that the same complexity order also applies to irregular bipartite graphs.

In the above discussions, it is implicitly assumed that the girth of the graph is known a priori. Since the computational complexity of finding the girth is at most $O(n^2)$, e.g., based on the algorithm of [21],[2] the extra complexity for computing the girth is negligible compared to the rest of the computations.

B. Memory Requirements

For each edge of the bipartite graph, we need two memory locations to store the message vectors in both directions. For a regular graph, since each vector has d_u elements, the total number of memory locations, each storing an integer number, is $2d_u|E|$ or $O(d_u|E|) = O(nd_u^2)$. For an irregular graph, the storage complexity is $O(d_{max}|E|)$, where d_{max} is the maximum node degree in U or W, depending on which side initiates the algorithm.

VI. Comparison with Existing Literature

A. Comparison with the Algorithm of [12]

First, it is important to note that while the algorithm of [12] is limited to bipartite graphs, the proposed algorithm is capable of counting short cycles in a general (non-bipartite) graph. For bipartite graphs, the algorithm of [12] counts cycles of length $g, g+2, g+4$, while the proposed algorithm counts cycles of length $g, g+2, \ldots, 2g-2$. The coverage of the proposed algorithm is thus at least as much as the algorithm of [12] for graphs with $g \geq 6$. It should be noted that the Tanner graphs of almost all good LDPC codes have $g \geq 6$.

The computational complexity of the algorithm of [12] is $O(gn^3)$, where $n = \max(|U|, |W|)$. The complexity of the proposed algorithm is $O(g|E|^2)$. One can thus see that

[2] It is easy to see that if we use the algorithm proposed in Section IV.B to compute g, the complexity is $O(gn^2d_u)$, which is in general larger than that of [21]. The algorithm of [21] however only finds g, while the proposed algorithm also computes N_g.

for sparse graphs with $|E|$ growing slower than $n^{3/2}$, the complexity of the proposed algorithm is less than that of the algorithm in [12]. Moreover the computations in the algorithm presented here are simple integer additions and subtractions, while in [12] the operations are mainly high-precision multiplications.

In terms of memory requirements, the algorithm of [12] requires at most $11(n^2 + m^2) + 21nm$ high bit-width (64-bit integer) storage locations, which is of order $O(n^2)$. The proposed algorithm on the other hand requires $2d_u|E|$ memory locations, i.e., $O(d_{max}|E|)$, which for sparse graphs can be much smaller than what is needed for the algorithm of [12]. Moreover, the maximum size of memory locations for the proposed algorithm, which is proportional to the number of cycles, is usually much less than 64 bits.

B. Relationship and Comparison with Matrix Multiplication Techniques

A natural approach to counting the number of k-cycles in a graph G is to use the k-th power of the directed edge matrix A_e of G. In general, $tr(A_e^k)$ is equal to the total number of tbc walks of length k in G. Now considering that each tbc walk of length k appears k times in the counting process (equivalent tbc walks are counted multiple times), and that each tbc walk is counted twice for the two directions, one can see that the total number of undirected non-equivalent tbc walks of length k is $tr(A_e^k)/2k$. Based on Lemma 2, all the tbc walks of length $k < 2g$ are cycles. We thus have

$$N_k = tr(A_e^k)/2k\ , \text{ for } \ k < 2g\ . \tag{4}$$

1) Complexity of calculating A_e^k: There are a number of methods for calculating the powers of a matrix. These methods have different complexities and their relative performance may depend on the sparsity level of the matrix. For the Tanner graph of LDPC codes, for example, the matrix A_e has only $\sum_{i=1}^{n} d_{u_i}^2 + \sum_{i=1}^{m} d_{w_i}^2 - 2|E|$ nonzero (one) elements. For the case of a regular LDPC code, this simplifies to $|E|(d_u + d_w - 2)$ nonzero elements. This implies that for the Tanner graph of LDPC codes, the matrix A_e is sparse.

To compute A_e^k, one can use the *iterative method*: $A_e^k = A_e^{k-1} A_e$. Another approach is to use *successive squares method*: $A_e^k = \prod_l A_e^{2^l}$, where $\sum_l 2^l$ represents the binary expansion of k. While the successive squares method is generally more efficient than the iterative method, the latter can be more efficient in the case that A_e is sparse. The reason is that the powers of a sparse matrix may not be sparse, and thus the general complexity of the successive squares method is determined by the complexity of dense matrix multiplication. This is while, in the case of the iterative method, in each iteration, at least one of the matrices (i.e., A_e) is sparse. This property can be used to reduce the computational complexity of the matrix multiplication.

Consider two integer $N \times N$ matrices A and B. The complexity of the straightforward calculation of $C = A \times B$ is $O(N^3)$. One of the first algorithms proposed for faster matrix multiplication is by Strassen [31] and has a complexity of $O(N^{2.81})$. Currently, the fastest known algorithm for matrix multiplication is the Coppersmith-Winograd algorithm [8] which has an asymptotic complexity of $O(N^{2.376})$. This algorithm however, cannot utilize the sparsity (even if any or both of the two matrices is sparse), and its complexity remains unchanged even if the multiplied matrices are extremely sparse [36]. Moreover, in the Coppersmith-Winograd algorithm, the complexity $O(N^{2.376})$ is only achieved when the size of the matrices tends to infinity. More recently, Yuster and Zwick [36] presented a method for fast sparse matrix multiplication with the asymptotic complexity of $O(M^{0.7}N^{1.2} + N^{2+o(1)})$, where M is the number of nonzero elements in the less sparse matrix among the two. This complexity result not only is asymptotic and applies only to very large matrices, but also requires both matrices to be sparse. One however should note that even the product of two very sparse matrices can be very dense, and thus the algorithm of [36] would not be applicable to the case of finding the k-th power of a sparse matrix.

For the case where only one of A or B is sparse, one can use the straightforward method of calculating the product $C = A \times B$ through: $c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$. It is easy to show that in this case, the complexity of multiplication is bounded by $O(MN)$, where M is the number of nonzero elements in the sparser matrix between A and B. Applying this result to the calculation of A_e^k by the iterative method, one can see that the computational complexity is $O(kMN)$. For the Tanner graph of regular LDPC codes, this reduces to $O(k|E|^2(d_u + d_w))$, which implies that the complexity of counting the cycles of length k, $g \leq k \leq 2g - 2$, using (4) is $O(g|E|^2(d_u + d_w))$ or $O(gn^2d_u^3 + gn^2d_u^2d_w))$. It is worth noting that since all the nonzero elements of A_e are ones, all the operations in computing A_e^k are simple additions. The storage complexity of this method is dominated by the required memory to store A_e^{k-1}. Since A_e^{k-1} can be dense, the storage complexity of the method is of $O(|E|^2)$ or $O(n^2d_u^2)$.

The comparison of the computational complexity and memory requirements of the proposed algorithm, which are $O(g|E|^2)$ and $O(d_{max}|E|)$, respectively, with those of the above matrix multiplication method (naive approach), which are $O(g|E|^2(d_v + d_w))$ and $O(|E|^2)$, respectively, shows the superiority of the proposed method.

2) Efficient calculation of N_k using matrix multiplication: Consider the directed edge matrix A_e corresponding to a graph $G = (V, E)$. We construct a $|V| \times 2|E|$ matrix B from A_e by adding the rows of A_e that correspond to the edges emanating from each node v of G for every $v \in V$, and removing all the other rows. Note that these rows have no overlap in the nonzero locations. We then multiply B by A_e from the right $k - 1$ times to obtain

$$B' = (\cdots((B \times A_e) \times A_e) \times \cdots \times A_e)\ . \tag{5}$$

We then derive a $|V| \times |V|$ matrix B'' from B' by adding up the columns of B' that correspond to the edges emanating from each node v of G for every $v \in V$ (and removing all the other columns), and then dividing the ith diagonal element of the resulting matrix by $d_i - 1$, where d_i is the degree of the ith node in G. The complexity of obtaining B'' from B' is $O(|V||E|)$. Based on (4), it is then easy to see that

$$N_k = tr(B'')/2k\ , \text{ for } \ k < 2g\ . \tag{6}$$

The complexity of calculating N_k using (5) and (6) is $O(k|V|M)$, where M is the number of nonzero elements in A_e. For the Tanner graph of regular LDPC codes, this reduces to $O(k|E|^2(d_u+d_w)^2/(d_ud_w))$, which implies that the complexity of counting the cycles of length k, $g \le k \le 2g-2$, for the Tanner graph of a regular LDPC code of a given rate $r = 1 - d_u/d_w$ using (6) is $O(g|E|^2)$, which is similar to that of the proposed message-passing algorithm. However, it should be noted that the memory requirement of the matrix multiplication approach is $O(|V||E|)$, which for example for a regular LDPC code reduces to $O(|E|^2/d_u)$ which is larger than that of the proposed algorithm.

3) Message-passing interpretation of matrix multiplication algorithms: Consider a bipartite graph representation of A_e, where one set of nodes, say, those on the left, represent the rows of A_e, and the other set of nodes (those on the right) represent the columns of A_e. An edge is connected between node i on the left and node j on the right iff the ijth element of A_e is nonzero. Suppose that we connect k such graphs together in sequence, such that the right nodes of the ith graph coincides with the left nodes of the $(i+1)$th graph, for $i = 1, \ldots, k-1$. We call the resulting graph a *trellis diagram*, or a *trellis* in brief, following the nomenclature used in coding. Both the naive and the efficient calculations of N_k can be described by a message-passing algorithm over this trellis.

The message-passing algorithm is started by assigning initial values to the left-most nodes of the trellis. These nodes then pass their initial messages along their outgoing edges to the adjacent set of nodes. The algorithm is then continued by the messages being passed forward (left to right) in the trellis, one trellis section at a time, with each node adding up the messages it receives along its incoming edges on the left and passing the result along all its outgoing edges on the right. The algorithm will end by all the right-most nodes receiving messages along their incoming edges and passing out the sum of those messages as their output.

If the message-passing algorithm is initiated by zero values on all the left-most nodes in the trellis except for node i, which is assigned the value 1, then the set of outputs will be the ith row of matrix A_e^k. If the values of the input trellis nodes corresponding to the rows of A_e that represent the outgoing edges of node v of G are set to one with the rest of the input values equal to zero, then the set of outputs will be equal to the row of B' in (5) which corresponds to v.

As a final note, one should realize that although the message-passing algorithms over the trellis diagram and the original graph (for counting the short cycles) have both essentially the same complexity, the graph representation of the latter (original graph) is simpler than that of the former (a trellis section). (Note that for example, for a regular (d_u, d_w) LDPC code, one section of the trellis diagram has $|E|(d_u + d_w - 2)$ edges, while the original Tanner graph has only $|E|$ edges.)

VII. Numerical Results

In this section, we present numerical results obtained by applying the proposed algorithm to Tanner graphs of LDPC codes. We consider four rate-$1/2$ codes from [37]. Codes A and B are listed in [37] as $PEGirReg504x1008$ and $PEGReg504x1008$, respectively. Both codes are constructed using the Progressive Edge Growth (PEG) method of [14], and have $n = 1008$ and $m = 504$. Code A is irregular while Code B is regular. Codes C and D are MacKay's codes $8000.4000.3.483$ and $10000.10000.3.631$, respectively. They are both regular with $d_u = 3$ and $d_w = 6$. For Code C, $n = 8000$ and $m = 4000$, while these parameters for Code D are $20,000$ and $10,000$, respectively. The number of short cycles in the Tanner graphs of these codes is listed in Table I. Codes A, C and D have girth 6 and the proposed algorithm, similar to the algorithm of [12], can compute N_6, N_8 and N_{10}. Code B however has girth 8, and while the algorithm of [12] can only compute N_8, N_{10} and N_{12}, the proposed algorithm can also compute N_{14}.

TABLE I
NUMBER OF SHORT CYCLES IN THE TANNER GRAPHS OF FOUR RATE-$1/2$ LDPC CODES

	Code A	**Code B**	**Code C**	**Code D**
N_6	11538	0	179	161
N_8	408657	2	1218	1260
N_{10}	13110235	11238	9989	10051
N_{12}	-	91101	-	-
N_{14}	-	748343	-	-

TABLE II
CPU TIME AND MEMORY REQUIREMENTS FOR THE PROPOSED ALGORITHM

	CPU Time (S)	**Max Memory (MB)**	**Max Swap (MB)**
Code A	5.3	0.36	3.3
Code B	3	0.36	2.8
Code C	155	13	157
Code D	1127	13	157

TABLE III
CPU TIME AND MEMORY REQUIREMENTS FOR THE ALGORITHM OF [12]

	CPU Time (S)	**Max Memory (MB)**	**Max Swap (MB)**
Code A	10.3	1.5	35
Code B	16.6	1.5	35
Code C	4965	7839	14195
Code D	-	-	-

Tables II and III show the running time and memory requirements of the proposed algorithm and the algorithm of [12],[3] respectively. Both algorithms were run on the same machine with a 2.2-GHz CPU and 8 GB of RAM. As can be seen, the proposed algorithm is consistently faster than the algorithm of [12] and requires significantly less memory for larger graphs. In fact, for Code D, the algorithm of [12] ran out of memory and was not able to find the results.

As another experiment, we randomly generate six parity-check matrices for each of the following three rate-1/2 LDPC code ensembles: $(d_u, d_w) = (3,6), (4,8), (5,10)$.

[3]To implement the algorithm of [12], we used the authors' code in [38].

TABLE IV
DISTRIBUTION OF SHORT CYCLES IN THE TANNER GRAPHS OF RATE-$1/2$ RANDOM REGULAR LDPC CODES WITH DIFFERENT DEGREE DISTRIBUTIONS AND DIFFERENT BLOCK LENGTHS

Degree Distribution	Short Cycle Distribution	Code Lengths					
		200	**500**	**1000**	**5000**	**10000**	**20000**
$(3, 6)$	N_6	171	167	181	156	166	148
	N_8	1265	1239	1226	1235	1253	1285
	N_{10}	10069	10110	9939	9982	9858	9974
$(4, 8)$	N_6	1636	1611	1584	1562	1537	1572
	N_8	25005	24419	24379	24363	24529	24557
	N_{10}	409335	409373	408595	407958	408246	409051
$(5, 10)$	N_6	8626	8064	8055	7978	7858	7926
	N_8	213639	212484	210767	210153	209614	210159
	N_{10}	6052158	6054661	6049148	6043400	6049583	6043704

The lengths for each degree distribution are: $n = 200, 500, 1000, 5000, 10,000$ and $20,000$. In the generation of the parity-check matrices, 4-cycles are avoided. The proposed algorithm is then used to count the short cycles of each parity-check matrix. The results, which are reported in Table IV, show that while there is a large difference between the short cycle distribution of different degree distributions, the changes with respect to the block length for the same degree distribution are negligible. This would imply that the complexity of the algorithms which are based on the enumeration of short cycles in a Tanner graph is rather independent of the block length.[4]

VIII. CONCLUSIONS AND DISCUSSIONS

In this paper, we proposed a distributed message-passing algorithm to count short cycles in a graph. For bipartite graphs, the proposed algorithm counts short cycles of length $g, g+2, \ldots, 2g-2$, where g is the girth of the graph. For non-bipartite graphs, the algorithm counts cycles of length $g, g+1, \ldots, 2g-1$. The operations performed by the algorithm are integer additions and subtractions, and the computational and storage complexities of the algorithm are $O(g|E|^2)$ and $O(d_{max}|E|)$, respectively, where $|E|$ and d_{max} are the number of edges and the maximum node degree in the graph, respectively. For sparse graphs, the proposed algorithm is significantly faster and requires substantially less memory compared to the existing algorithms, such as that of [12], which are tailored for counting short cycles. This is particularly the case for larger graphs.

Interestingly, the more generic and basic approach of matrix multiplication, when properly implemented, has a complexity similar to the proposed message-passing algorithm, and is thus less complex than the approaches specifically tailored for counting *short* cycles, in particular for large sparse graphs. We demonstrated that the efficient implementation of matrix multiplication approach can also be described as a forward message-passing algorithm over a trellis diagram. Both the forward message-passing algorithm and the one proposed in this paper in fact count the tailless backtrackless closed (tbc) walks in the graph, which happen to coincide with the simple cycles as long as their length is less than twice the girth of the graph. The difference however is that to limit the closed walks to tbc walks, the former operates on a properly defined graph (matrix), i.e., the trellis associated with the directed edge matrix, while the latter employs the extrinsic property of the message-passing over the original graph.

While in this paper, our main focus was on counting short simple cycles, there may be applications where one is interested in counting the tbc walks in a graph. One example is the characterization of pseudo-codewords of iterative coding schemes [19]. In such applications, the proposed message-passing algorithm can be applied with no limitation on the length of the closed walks.

[4]It is worth mentioning that it is proved in [22] that for random regular bipartite graphs with $d_u = d_w = d$, as the number of nodes tends to infinity, the distribution of short cycles of different length c_i tends to independent Poisson distributions with average $\mu_i = (d-1)^{c_i}/c_i$. To the best of our knowledge, however, no generalization of this result is available for bipartite graphs with $d_u \neq d_w$.

IX. ACKNOWLEDGEMENT

The authors wish to thank Pascal Vontobel for his comments on the first version of this paper which significantly improved the presentation and the scope of this work. In particular, the material on the matrix multiplication approach was added to the paper as a result of such comments.

REFERENCES

[1] N. Alon, R. Yuster, and U. Zwick, “Finding and counting given length cycles,” *Algorithmica*, vol. 17, no. 3, pp. 209–223, 1997.
[2] R. Asvadi, A. H. Banihashemi, and M. Ahmadian-Attari, “Lowering the error floor of LDPC codes using cyclic liftings,” *IEEE Trans. Inf. Theory*, vol. 57, no. 4, pp. 2213–2224, Apr. 2011.
[3] E. T. Bax, “Algorithms to count paths and cycles,” *Inf. Process. Lett.*, vol. 52, pp. 249–252, 1994.
[4] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit errorcorrecting coding and decoding: turbo-codes,” in *Proc. 1993 IEEE Int. Conf. Commun.*, pp. 1064-1070.
[5] G. G. Cash, “The number of n-cycles in a graph,” *Applied Mathematics and Computation*, vol. 184, pp. 1080–1083, 2007.
[6] Y. C. Chang and H. L. Fu, “The number of 6-cycles in a graph,” *Bull. Inst. Combin. Appl.*, vol. 39, pp. 27–30, 2003.
[7] R. Chen, H. Huang, and G. Xiao, “Relation between parity-check matrices and cycles of associated Tanner graphs,” *IEEE Commun. Lett.*, vol. 11, no. 8, pp. 674–676, Aug. 2007.
[8] D. Coppersmith and S. Winograd, “Matrix multiplication via arithmetic progressions,” *J. Symbolic Comput.*, vol. 9, pp. 251–280, 1990.
[9] J. Fan and Y. Xiao, “A method of counting the number of cycles in LDPC codes,” in *Proc. 2006 Int. Conf. Signal Procss.*, vol. 3, pp. 2183–2186.

[10] J. Flum and M. Grohe, “The parameterized complexity of counting problems,” in *Proc. 2002 IEEE Symp. Foundations Comput. Science*, pp. 538–547.

[11] R. G. Gallager, “Low-density parity-check codes,” *IEEE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.

[12] T. R. Halford and K. M. Chugg, “An algorithm for counting short cycles in bipartite graphs,” *IEEE Trans. Inf. Theory*, vol. 52, no. 1, pp. 287–292, Jan. 2006.

[13] M. Horton, “Ihara zeta functions of irregular graphs,” Ph.D. thesis, U.C.S.D., 2006.

[14] X.-Y. Hu, E. Eleftheriou, and D. M. Arnold, “Regular and irregular progressive edge-growth Tanner graphs,” *IEEE Trans. Inf. Theory*, vol. 51, no. 1, pp. 386–398, Jan. 2005.

[15] D. B. Johnson, “Find all the elementary circuits of a directed graph,” *J. SIAM*, vol. 4, pp. 77–84, 1975.

[16] M. Karimi and A. H. Banihashemi, “An efficient algorithm for finding dominant trapping sets of LDPC codes,” in *Proc. 2010 International Symp. Turbo Codes Iterative Inf. Process.*

[17] M. Karimi and A. H. Banihashemi, “An efficient algorithm for finding dominant trapping sets of irregular LDPC codes,” in *Proc. 2011 IEEE ISIT*.

[18] M. Karimi and A. H. Banihashemi, “An efficient algorithm for finding dominant trapping sets of LDPC codes,” to appear in *IEEE Trans. Inf. Theory*. Available: CoRR,arxiv.org/abs/1108.4478.

[19] R. Koetter, W. C. W. Li, P. O. Vontobel, and J. L. Walker, “Characterizations of pseudo-codewords of (low-density) parity-check codes,” *Adv. Mathematics*, vol. 213, no. l, pp. 205–229, 2007.

[20] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, “Factor graphs and the sum-product algorithm,” *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 498–519, Feb. 2001.

[21] Y. Mao and A. H. Banihashemi, “A heuristic search for good low-density parity-check codes at short block lengths,” in *Proc. 2001 IEEE Int. Conf. Commun.*, vol. 1, pp. 41–44.

[22] B. D. McKay, N. C. Wormald, and B. Wysocka, “Short cycles in random regular graphs,” *Electron. J. Combinatorics*, vol. 11, no. R66, 2004.

[23] T. J. Richardson and R. Urbanke, “The capacity of low-density parity-check codes under message-passing decoding,” *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 599–618, Feb. 2001.

[24] G. S. Staples, “A new adjacency matrix for finite graphs,” in *Proc. 2005 International Conf. Cliord Algebras Their Appl.*

[25] R. Schott and G. S. Staples, “Nilpotent adjacency matrices, random graphs, and quantum random variables,” *J. Phys. A: Math. Theor.*, vol. 41, pp. 155–205, 2008.

[26] R. Schott and G. S. Staples, “On the complexity of counting cycles in sparse graphs using nilpotent adjacency matrices,” in *Proc. 2010 Midwest Conf. Combinatorics, Cryptography, Comput.*

[27] R. Schott and G. S. Staples, “On the complexity of cycle enumeration using zeons,” in *Proc. 2010 Appl. Geometric Algebras Comput. Science Eng.*

[28] R. Schott and G. S. Staples, “Cycle enumeration using nilpotent adjacency matrices with algorithm runtime comparisons.” Available: www.loria.fr/~schott/CVRENE3html.html.

[29] H. M. Stark and A. A. Terras, “Zeta functions of finite graphs and coverings,” *Adv. Math.*, vol. 121, pp. 124–165, 1996.

[30] C. Storm, “Some properties of graphs determined by edge zeta functions,” *Linear Algebra Appl.*, vol. 434, no. 5, pp. 1285–1294, Mar. 2011.

[31] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson, “Implementation of Strassen’s algorithm for matrix multiplication,” in *Proc. 1996 ACM/IEEE Conf. Supercomput.*, pp. 32.

[32] R. M. Tanner, “A recursive approach to low-complexity codes,” *IEEE Trans. Inf. Theory*, vol. 27, pp. 533–547, 1981.

[33] R. Tarjan, “Enumeration of the elementary circuits of a directed graph,” *J. SIAM*, vol. 2, pp. 211–216, 1973.

[34] H. Xiao and A. H. Banihashemi, “Graph-based message-passing schedules for decoding LDPC codes,” *IEEE Trans. Commun.*, vol. 52, no. 12, pp. 2098–2105, Dec. 2004.

[35] H. Xiao and A. H. Banihashemi, “Error rate estimation of low-density parity-check codes on binary symmetric channels using cycle enumeration,” *IEEE Trans. Commun.*, vol. 57, no. 6, pp. 1550–1555, June 2009.

[36] R. Yuster and U. Zwick, “Fast sparse matrix multiplication,” *ACM Trans. Algorithms*, vol. 1, pp. 2–13, 2004.

[37] Available: http://www.inference.phy.cam.ac.uk/mackay/codes/ data.html

[38] Available: http://csi.usc.edu/chugg/tools

Mehdi Karimi (S’10) received the B.A.Sc. degree and M.A.Sc degrees both in electrical engineering from Isfahan University of Technology, Isfahan,Iran in 2003 and 2005, respectively. From 2005 to 2008, he was with the Information and Communication Technology Institute, Isfahan University of Technology. Since 2008, he has been pursuing his Ph.D at Carleton University, Canada. His research interests include graphical modeling, iterative coding, statistical signal processing and wireless communications.

Amir H. Banihashemi (S’90-A’98-M’03-SM’04) received the B.A.Sc. degree in electrical engineering from Isfahan University of Technology, Isfahan, Iran in 1988, and the M.A.Sc. degree in communications engineering from the University of Tehran, Tehran, Iran, in 1991, with the highest academic rank in both classes.

From 1991 to 1994, he was with the Electrical Engineering Research Center and the Department of Electrical and Computer Engineering, Isfahan University of Technology. During 1994-1997, he was with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada, for his Ph.D. degree. He held two Ontario Graduate Scholarships for international students during this period. In 1997, he joined the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada, where he worked as a Natural Sciences and Engineering Research Council of Canada (NSERC) Postdoctoral Fellow. He joined the Faculty of Engineering at Carleton University in 1998, where at present he is a Professor in the Department of Systems and Computer Engineering. His research interests include coding and information theory, digital and wireless communications, network coding, theory and implementation of communication algorithms, and compressed sensing and sampling. He has published more than 150 papers in refereed journals and conferences.

Dr. Banihashemi served as an Associate Editor for the IEEE TRANSACTIONS ON COMMUNICATIONS from 2003 to 2009. He is the Director of the Broadband Communications and Wireless Systems (BCWS) Centre at Carleton University. He has been involved in many international conferences as chair, member of technical program committee, and member of organizing committee. These include co-chair for the Communication Theory Symposium of Globecom’07, TPC co-chair of the Information Theory Workshop (ITW) 2007, member of the organizing committee for the International Symposium on Information Theory (ISIT) 2008, and co-chair of the Canadian Workshop on Information Theory (CWIT) 2009. Dr. Banihashemi is a recipient of Carleton’s Research Achievement Award in 2006 and 2012, and Faculty Graduate Mentoring Award in 2011. He is also awarded one of the NSERC’s one hundred 2008 Discovery Accelerator Supplements (DAS).