A Branch-and-Cut Algorithm for a Bipartite Graph Construction Problem in Digital Communication Systems

Banu Kabakulak*[1], Z. Caner Taşkın[1], and Ali Emre Pusane[2]

[1]Department of Industrial Engineering, Boğaziçi University, İstanbul, Turkey

[2]Department of Electrical and Electronics Engineering, Boğaziçi University, İstanbul, Turkey

We study a bipartite graph (BG) construction problem that arises in digital communication systems. In a digital communication system, information is sent from one place to another over a noisy communication channel using binary symbols (bits). The original information is encoded by adding redundant bits, which are then used to detect and correct errors that may have been introduced during transmission. Harmful structures, such as small cycles, severely deteriorate the error correction capability of a BG. We introduce an integer programming formulation to generate a BG for a given smallest cycle length. We propose a branch-and-cut algorithm for its solution and investigate the structural properties of the problem to derive valid inequalities and variable fixing rules. We also introduce heuristics to obtain feasible solutions for the problem. The computational experiments show that our algorithm can generate BGs without small cycles in an acceptable amount of time for practically relevant dimensions.

Keywords: Telecommunications, bipartite graphs, integer programming, branch-and-cut algorithm.

*Corresponding author. Tel.: +90 2123596771; fax: +90 2122651800.
E-mail addresses: banu.kabakulak@boun.edu.tr (B. Kabakulak), caner.taskin@boun.edu.tr (Z. C. Taşkın), ali.pusane@boun.edu.tr (A. E. Pusane).

1 Introduction and Literature Review

Telecommunication is the transmission of messages from a transmitter to a receiver over a potentially unreliable communication environment. In a digital communication system, any information is represented by binary code symbols (*bits*). In parallel to rapid developments in technology, digital communication systems have several application areas such as messaging via digital cellular phones, fiber optic Internet, TV broadcasting, and agricultural monitoring through digital satellites. Information is often transmitted across noisy environments such as air or space, which may introduce transmission errors. In digital communications, the original information is encoded by adding redundant bits to enable error recovery. When the receiver obtains the information, the decoder estimates the original information by detecting and correcting errors in the received vector using the redundant bits.

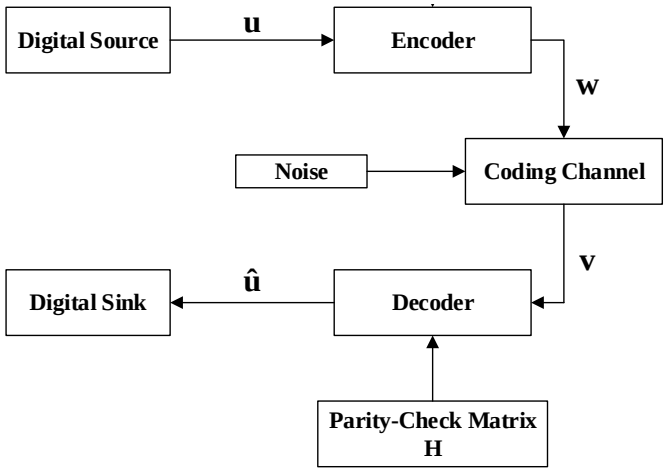

Figure 1: Digital communication system diagram

Figure 1 shows the information flow in a digital communication system using low-density parity-check (LDPC) codes. Let the original information be a binary vector $\mathbf{u}$ of k-bits. The encoder adds redundant parity-check bits to vector $\mathbf{u}$, generating codeword $\mathbf{w}$ of n-bits, where $n \geq k$. After the transmission of codeword $\mathbf{w}$ through a noisy channel, the receiver receives vector $\mathbf{v}$ of n-bits. The decoder detects whether the received vector $\mathbf{v}$ includes errors or not by checking whether the expression $\mathbf{vH}^{\mathrm{T}}$ is equal to vector $\mathbf{0}$ in (mod 2), where $\mathbf{H}$ is a binary matrix known as parity-check matrix (see Figure 2). In case where $\mathbf{v}$ contains errors, the decoder attempts to determine the error locations and tries to fix them [11]. As a result, the information $\mathbf{u}$ sent from the source is estimated as $\hat{\mathbf{u}}$ at the sink.

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Figure 2: A parity-check matrix from $(3, 6)$-regular LDPC code family

An LDPC code is identified by its $\mathbf{H}$ matrix, and said to be *regular* if there is a constant number of ones at each column and row of the $\mathbf{H}$ matrix. The $\mathbf{H}$ matrix given in Figure 2 represents a $(3, 6)$-regular code since there are 3 ones at each column and 6 ones at each row. The code can alternatively be represented as a bipartite graph (BG) corresponding to the $\mathbf{H}$ matrix [28]. On one part of the BG, there is a variable node j (v_j), $j \in \{1, ..., n\}$, for each bit of the received vector $\mathbf{v}$. Each row of the $\mathbf{H}$ matrix corresponds to a check node i $(c_i), i \in \{1, ..., n-k\}$, on the other part of the BG. The *degree* of v_j (c_i) is the number of adjacent check nodes (variable nodes) on the BG. Hence, the $\mathbf{H}$ matrix is the bi-adjacency matrix of the BG. A check node c_i is *satisfied* if the sum of the adjacent bits is zero in (mod 2). Figure 3 shows the BG representation of the $\mathbf{H}$ matrix given in Figure 2.

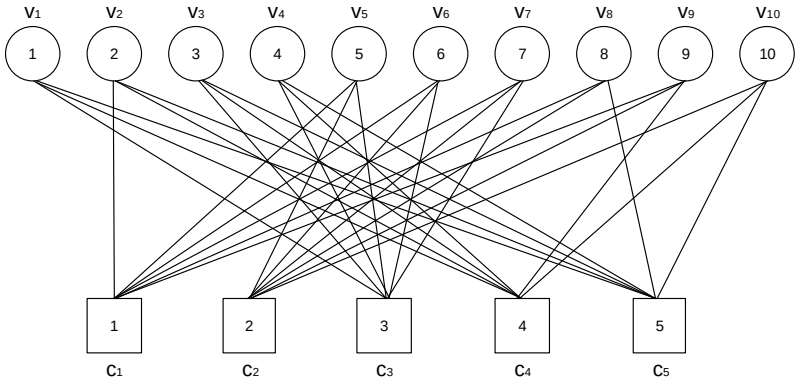

Figure 3: BG representation of the parity-check matrix given in Figure 2

The sparsity property of LDPC codes facilitates the usage of iterative decoding algorithms (such as Gallager A) with low complexity [10, 33]. We demonstrate the Gallager A algorithm under a binary symmetric channel (BSC) in Figure 4. In case of an error BSC flips the value of a bit from 0 to 1, and vice versa. Assuming that the decoder has received codeword $\mathbf{w} = (1\ 1\ 0\ 0\ 0)$ as vector $\mathbf{v} = (0\ 0\ 1\ 0\ 1)$, Gallager A sends these bits to the check nodes to evaluate the parity-check equations (Figure 4a). We observe that c_2 and c_3 are satisfied whereas c_1 is not. Then, each check node conveys the information of whether it is satisfied (S) or unsatisfied (U) to the adjacent variable nodes.

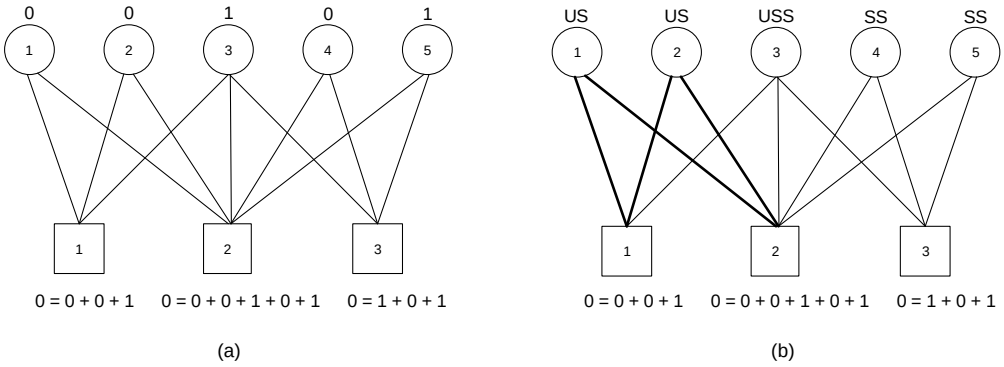

Figure 4: An iteration of Gallager A algorithm

Among the variable nodes that have more unsatisfied adjacent check nodes than satisfied nodes, Gallager A flips the value of the node v_j having the highest number of unsatisfied adjacent check nodes. The message passing between the variable and check nodes continues until all check nodes are satisfied (a codeword is found), or a termination criterion (such as iteration limit or no candidate bit to flip) is met. In Figure 4b, notice that all of the adjacent check nodes of erroneously received bits v_1 and v_2 are in the length-4 cycle $(v_1,\ c_1,\ v_2,\ c_2)$. Gallager A fails to recover the bits v_1 and v_2, since the number of satisfied check nodes equals the number of unsatisfied nodes. Hence, Gallager A terminates with the vector $\mathbf{v} = (0\ 0\ 1\ 0\ 1)$ due to the nonpresence of a candidate bit to flip. The resulting vector is not a codeword as c_1 is not satisfied.

As we illustrate in Figure 4, the error correction capability of iterative decoding algorithms (such as Gallager A) deteriorates due to cycles in the BG. If there is no iteration limit, an iterative decoding algorithm may loop indefinitely among vectors in the presence of small cycles [24]. The length of a smallest cycle in a graph is known as *girth* [13]. In this study, we focus on designing regular LDPC codes whose BGs do not contain small cycles. In particular, we aim to construct a BG having girth no smaller than a given target girth value g.

In practice, an LDPC code is printed on the board of an electronical device to carry out digital communication. Regular LDPC codes have the advantage of easy hardware implementation compared to irregular ones. As given in Figure 4, the information is decoded in packets of length n, and the complexity of an iterative decoding algorithm depends on the degrees J and K. In order to minimize the decoding latency, LDPC codes having small (m, n) dimensions and small (J, K) values are preferable. Hence, determining the minimum (m, n) dimensions when girth g and (J, K) degrees are given is one of the focal points of this work.

The literature includes studies on the girth of a graph and constructing high girth graphs. [32] investigates the theoretical properties of graphs with large girth. The linear time algorithm of [9] computes the girth of a planar graph. [25] develops an approximation algorithm for the minimum weight cycle in a weighted undirected graph. This algorithm is used to approximate the girth of a graph. Some families of small regular graphs of girth 5 are described in [1]. Small regular BGs of girth 6 can be constructed with the method given in [3]. [16] focuses on the r-regular graphs with large girth and provides upper and lower bounds on the size of maximum independent sets, maximum dominating sets, and minimum connected sets.

[8] describes a high girth graph construction algorithm that starts from the empty graph. The bit-filling heuristic starts with a large girth target and decreases its target as it inserts edges to BG one-by-one [7]. The heuristic terminates when a prescribed girth is met. The Progressive Edge Growth (PEG) heuristic is based on adding edges to the BG iteratively without constructing small cycles [18]. In [21] certain edges are exchanged within the BG to eliminate small cycles without creating any others. In the edge deletion algorithm an edge that commonly appears in the maximum number of cycles is selected [4]. In [15] the selection criteria of the PEG algorithm to locate an edge in a BG is modified in order to have a better girth value than PEG. There are other constructive heuristics in the literature that avoid small cycles [5, 19]. A genetic algorithm to design a BG with a small number of nodes is given in [6]. In [26] a modified shortest-path algorithm is used to construct a BG.

These methods are heuristic approaches and they change the degree distribution of the nodes in the BG. Furthermore, they may fail to generate a BG for a given dimension with a target girth value. In this study, we investigate the BG construction problem from an optimization point of view. Our main contributions to the literature can be listed as follows:

- We propose an integer programming (IP) formulation to generate BGs with a given girth value and develop a branch-and-cut (B&C) algorithm for its solution. To the best of our knowledge, our paper is the first work that implements optimization algorithms on an IP model for the BG (or LDPC code) design problem.

- We propose methods that are capable of finding a BG for a given girth value, or proving that there cannot be such a BG.

- We investigate the structural properties of the problem for (J, K)-regular BGs to improve our algorithm by applying a variable fixing scheme, adding valid inequalities and uti-

lizing initial solution generation heuristics. The computational results indicate that our proposed methods significantly improve the solvability of the problem.

- We discuss how our method can be used to find the smallest dimension n that one can generate a (J, K)-regular BG.

The remainder of the paper is organized as follows: we formally define the problem, and introduce the mathematical formulation and the proposed B&C algorithm in the next section. We propose a number of techniques to improve the performance of the B&C algorithm in Section 3. We test the efficacy of our methods via computational experiments in Section 4. Finally, some concluding remarks and comments on future work appear in Section 5.

2 Solution Methods

In this section, we introduce our integer programming formulation and propose a B&C algorithm for the solution of the problem. We investigate additional methods to improve the performance of our B&C algorithm. We summarize the symbols used in this paper in Table 1.

Table 1: List of the symbols

Parameters	
$\mathbf{H}$	parity-check matrix
k	length of the original information
n	length of the encoded information, number of columns in $\mathbf{H}$
m	$n-k$, number of rows in $\mathbf{H}$
g	target girth
v_j	variable node j
c_i	check node i
$d(v_j)$	target degree of v_j
$d(c_i)$	target degree of c_i
$\rho(i,j)$	local girth of edge $\{c_i, v_j\}$
Decision Variables	
x_{ij}	(i,j) entry of the $\mathbf{H}$ matrix
s_j	slack for degree of v_j
t_i	slack for degree of c_i

2.1 Mathematical Formulation

In our Minimum Degree Deviation (MDD) model, our aim is to generate an $\mathbf{H}$ matrix (a BG) of dimensions (m, n), where $m = n - k$, having girth no smaller than a given value g. In the MDD model, x_{ij} represents the (i, j) entry of the $\mathbf{H}$ matrix. We are given the target degree distributions $d(v_j)$ and $d(c_i)$ in the BG for the nodes v_j and c_i, respectively. The degree

deviation s_j of the node v_j from the target $d(v_j)$ is modeled with the constraints (2). Similarly, the constraints (3) introduce the deviation t_i of the node c_i from the target $d(c_i)$. In the constraints (4), $K_{m,n}$ represents a complete bipartite graph with bipartitions $C = \{c_1, ..., c_m\}$ and $V = \{v_1, ..., v_n\}$. These constraints eliminate the cycles with length less than the target girth g from the solution space. The objective function (1) minimizes the total weighted degree deviations from the given degree distributions.

Minimum Degree Deviation (MDD) Model:

$$\min \quad \sum_{j=1}^{n} \frac{s_j}{d(v_j)} + \sum_{i=1}^{m} \frac{t_i}{d(c_i)} \tag{1}$$

$$\text{s.t.:} \quad \sum_{i=1}^{m} x_{ij} + s_j = d(v_j),\ j = 1, ..., n \tag{2}$$

$$\sum_{j=1}^{n} x_{ij} + t_i = d(c_i),\ i = 1, ..., m \tag{3}$$

$$\sum_{(i,j):\{c_i,v_j\}\in S} x_{ij} \leq |S| - 1, \quad \text{for each cycle } S \subseteq E(K_{m,n}) : |S| < g \tag{4}$$

$$x_{ij} \in \{0, 1\}, \quad i = 1, ..., m,\ j = 1, ..., n. \tag{5}$$

$$s_j, t_i \geq 0, \quad i = 1, ..., m,\ j = 1, ..., n. \tag{6}$$

As we discussed in Section 1, an iterative decoder (such as Gallager A) decides on the value of a bit using the messages among the check and variable nodes. The decoder's decision is more reliable if it collects messages from as many nodes as possible. In other words, the degree deviation of a node having a smaller target degree is more important in terms of the error correction capability of a code. Hence, the weight of a degree deviation (s_j or t_i) is inversely proportional to the target degree ($d(v_j)$ or $d(c_i)$) in the objective function (1) of our MDD model.

Note that if all deviations s_j and t_i are zero in MDD, i.e., the value of the objective function is zero, one obtains a BG with the target degree distributions and the girth is no smaller than g. As a special case, one can generate a (J, K)-regular $\mathbf{H}$ matrix by picking $d(v_j) = J$ for all j, and $d(c_i) = K$ for all i. Since there can be an exponential number of cycles in a BG, we can have exponential number of the constraints (4) in the MDD model. In order to obtain a solution in an acceptable amount of time, we can add the constraints (4) in a cutting-plane fashion to MDD. This is the main idea behind our B&C algorithm explained in the next section.

2.2 Branch-and-Cut (B&C) Algorithm

In our B&C algorithm, we are given a target girth value g, and the dimensions (m, n) of $\mathbf{H}$ matrix. We initialize our algorithm by relaxing the constraints (4) from MDD, to obtain the relaxed model MDD^r. We can find either an integral or a fractional solution after solving MDD^r. We test the feasibility of an integral solution x of MDD^r with respect to the relaxed constraints (4) by our integral solution separation (*IntSep*) algorithm. Let $G_x = (V \cup C, E_x)$ be the induced BG by the nonzero x_{ij} values. *IntSep* first constructs the graph G_x of an integral solution x, and implements a depth-first-search (DFS) up to depth $(g-2)$ to detect all cycles smaller than g. The complexity of the *IntSep* algorithm is determined by the number of nodes in the DFS tree, and can be given as $\mathcal{O}(\bar{J}[(\bar{J}-1)(\bar{K}-1)]^{g/2})$, where $\bar{J} = \max_{v_j \in V}\{d(v_j)\}$ and $\bar{K} = \max_{c_i \in C}\{d(c_i)\}$.

We can strengthen the linear relaxation of MDD by separating a fractional solution x of MDD^r that violates the constraints (4). We can detect all cycles that violate the constraints (4) by implementing DFS up to depth $(g-2)$ in the induced graph G_x by the nonzero x_{ij} values as in *IntSep*. However, the number of nonzero x_{ij} values can be mn for a fractional x, whereas it is at most $\bar{J}\bar{K}$ in *IntSep*. Hence, determining all violating cycles for a fractional solution has complexity $\mathcal{O}(m[(m-1)(n-1)]^{g/2})$. Instead, our *FracSep* algorithm finds the maximum mean cost cycle on G_x to find some of the cycles that violate the constraints (4) with lower complexity.

The minimum mean cost cycle problem, which is a special case of the minimum cost-to-time ratio cycle problem, is a well known network problem in the literature and there is a polynomial time solution algorithm that solves it in directed graphs [2, 12, 22]. The problem aims to find a directed cycle S having the smallest mean cost $\sum_{(c_i,v_j)\in S} x_{ij}/|S|$ in a directed graph. However, we cannot implement this algorithm directly since we seek an undirected BG. Instead, we update the best known mean cost by utilizing a negative cycle detection algorithm repeatedly. Recall that the Bellman-Ford algorithm can detect negative cycles while searching for one-to-many shortest paths in directed graphs. The Bellman-Ford algorithm is also applicable for the undirected induced G_x in $\mathcal{O}((m+n)|E_x|)$ time if for an edge $\{c_i, v_j\}$ the algorithm updates the distance label of node v_j when it is not the predecessor of node c_i [2]. If the algorithm detects a negative cycle, we can track the predecessor list to identify the cycle.

In our *FracSep* algorithm we use the undirected Bellman-Ford algorithm to detect negative cycles within a mean cost update method. We first set edge costs as $-x_{ij}$ to convert our maximization problem to a minimization problem. Let μ represent an estimation on the minimum mean cost and μ^* denote the (unknown) optimal value of μ. Then, given a μ value, we update the edge costs to $(-x_{ij} - \mu)$ and check for the existence of a negative cycle. If we start with a μ that is an upper bound for μ^*, we can encounter one of the following cases for μ^*.

Case 1: G has a negative cycle S. In this case, $\sum_{\{c_i,v_j\}\in S}(-x_{ij} - \mu) < 0$. Therefore,

$$\mu > -\frac{\sum_{\{c_i,v_j\}\in S} x_{ij}}{|S|} > \mu^*. \tag{7}$$

Hence, μ is a strict upper bound on μ^*. We can update μ as $\mu = -\frac{\sum_{\{c_i,v_j\}\in S} x_{ij}}{|S|}$ in the next iteration.

Case 2: G has a zero-cost cycle S^*. In this case, $\sum_{\{c_i,v_j\}\in S^*}(-x_{ij} - \mu) = 0$. This implies,

$$\mu = -\frac{\sum_{\{c_i,v_j\}\in S^*} x_{ij}}{|S^*|} = \mu^*. \tag{8}$$

Hence, $\mu = \mu^*$, and S^* is a minimum mean cost cycle. *FracSep* concludes that $\mu = \mu^*$ if $|\mu - \mu^*| < \epsilon$, where ϵ is a given very small positive real number.

Input: A solution of MDD^r with fractional x_{ij} values, g target girth

1. Let $\mu = 0$, set cost of the edge $\{c_i, v_j\}$ as $(-x_{ij} - \mu)$.
2. **While** we can detect a negative cycle S with the undirected Bellman-Ford,
3. **If** $|S| < g$ and S violates (4), **Then** add the corresponding cut (4).
4. Update $\mu \leftarrow -\frac{\sum_{\{c_i,v_j\}\in S} x_{ij}}{|S|}$
5. **End While**

Output: Cuts added to MDD^r model

Figure 5: Fractional solution separation (*FracSep*) algorithm

FracSep in Figure 5 summarizes our fractional solution separation algorithm. We initially set $\mu = 0$, since it is an upper bound on μ^*. If we can find a violating negative cycle with length $|S| < g$, we can add a cut to MDD^r. This means that S is a cycle with $\sum_{\{c_i,v_j\}\in S} x_{ij} > |S| - 1$. We continue updating μ values until we find a minimum mean cycle. The time complexity of *FracSep* is $\mathcal{O}(log(1/\epsilon)(m+n)|E_x|)$.

3 Modeling and Algorithmic Improvements

In this section, we propose some improvements to the B&C algorithm given in the previous section. We first observe that the solution space of MDD includes symmetric solutions. Hence, we consider a variable fixing approach to decrease the adverse effect of symmetry. Secondly, we introduce some valid inequalities to improve the linear relaxation of MDD. Finally, we develop heuristic approaches to provide an initial feasible solution to the B&C algorithm.

3.1 Symmetry in the MDD Solution Space

In combinatorial optimization problems, such as scheduling, symmetry among the solutions is an important issue, which directly affects the performance of the applied solution methods [14, 27]. We observe that the feasible region of MDD contains symmetric solutions. That is, there can be isomorphic representations of a BG obtained by permuting the variable and check nodes. As an example, the variable nodes are in the sequence $\{v_1, v_2, v_3, v_4\}$ in Figure 6a, and the names of v_2 and v_4 are swapped in Figure 6b.

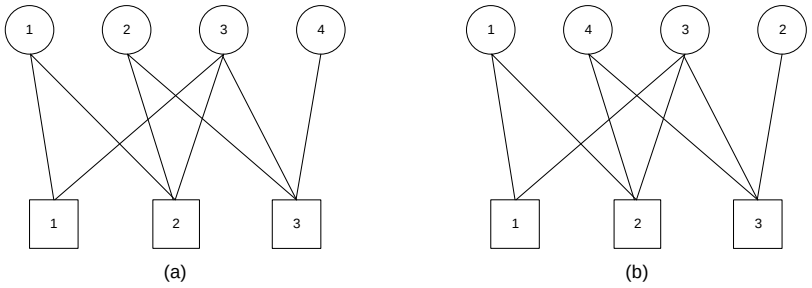

Figure 6: Symmetry in the MDD solution space

In Figure 7, $\mathbf{H_1}$ and $\mathbf{H_2}$ are the parity-check matrices for the BGs in Figures 6a and 6b, respectively. We see that although the BGs are isomorphic, their $\mathbf{H}$ matrix representations are not identical. In the MDD solution space, $\mathbf{H_1}$ and $\mathbf{H_2}$ are considered as two different solutions, which increases the complexity of the solution algorithm.

$$\mathbf{H_1} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \qquad \mathbf{H_2} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Figure 7: Parity-check matrices for the BGs in Figure 6

We can calculate the number of symmetric solutions for a BG as $(n!)(m!)$, since we can permute n variable nodes as $(n!)$ and m check nodes in $(m!)$ different ways.

3.2 Symmetry Breaking with Variable Fixing

In the literature ordering the decision variables, adding symmetry-breaking cuts to the formulation, and reformulating the problem are some of the suggested techniques to eliminate symmetric solutions from the feasible region [27, 20]. In our case, we propose a variable fixing scheme (given as *VarFix* in Figure 8) for the nonzero x_{ij} entries of a (J, K)-regular $\mathbf{H}$ matrix, which breaks symmetry and does not affect the girth g of the BG.

Input: (m, n) dimensions, (J, K) values

0. Let $r_{cr} = \lfloor (n-1)/(K-1) \rfloor$, and $c_{cr} = \lfloor (m-1)/(J-1) \rfloor$.
   For $i = 1, ..., r_{cr}, j = 1, ..., n$, set $x_{ij} = 0$.
   For $i = r_{cr} + 1, ..., m, j = 1, ..., c_{cr}$, set $x_{ij} = 0$.
   Set $x_{11} = 1$.
1. **For** $i = 1, ..., r_{cr} + 1, j = 1, ..., K - 1$,
2. **If** $1 + (i-1)(K-1) + j \leq n$, **Then** set $x_{i,1+(i-1)(K-1)+j} = 1$.
3. **End For**
4. **For** $i = 1, ..., J - 1, j = 1, ..., c_{cr} + 1$,
5. **If** $1 + (j-1)(J-1) + i \leq m$, **Then** set $x_{1+(j-1)(J-1)+i,j} = 1$.
6. **End For**

Output: Some x_{ij} values are fixed

Figure 8: Variable fixing (*VarFix*) algorithm

VarFix algorithm starts with a matrix whose entries are initially zero. In the first iteration, *VarFix* constructs the $\mathbf{H}^1$ matrix by fixing the first K entries in the first row and the first J entries in the first column to 1. The remaining entries in the first row and column are set to 0, since the constraints (2) for $j = 1$, and the constraints (3) for $i = 1$ are satisfied without deviation, i.e., $s_1 = t_1 = 0$. For a nonzero entry (i, j) of an $\mathbf{H}$ matrix, the *local girth* $\rho(i, j)$ is the size of the smallest cycle including the edge $\{c_i, v_j\}$ in the BG [18]. In the second iteration, we temporarily assume that $x_{ij} = 1$ for each unfixed (i, j) entry, and determine $\rho(i, j)$ by carrying out a breadth-first-search (BFS) on $\mathbf{H}^1$ starting from node v_j (see Figure 12 in Section 3.3 for $\rho(i, j)$ values when $J = 3$ and $K = 6$). We obtain $\mathbf{H}^2$ by permanently fixing the first $(K - 1)$ entries in the second row, and the first $(J - 1)$ entires in the second column that have $\rho(i, j) = \infty$ (no cycle is formed) to 1. *VarFix* generates $\mathbf{H}^\gamma$ in the γth iteration, and alternates between the rows and columns as allowed by the dimensions (m, n).

We illustrate *VarFix* in Figure 9 for a $(3, 6)$-regular BG of dimensions $(20, 40)$. In Figure 9, the labels on the rows and columns show the sum of the values in that row and column, respectively. We observe that for $r_{cr} = \lfloor (n-1)/(K-1) \rfloor$ many rows, the sum is equal to 6, and $c_{cr} = \lfloor (m-1)/(J-1) \rfloor$ many columns, the sum is equal to 3. Hence, for c_{cr}-columns the

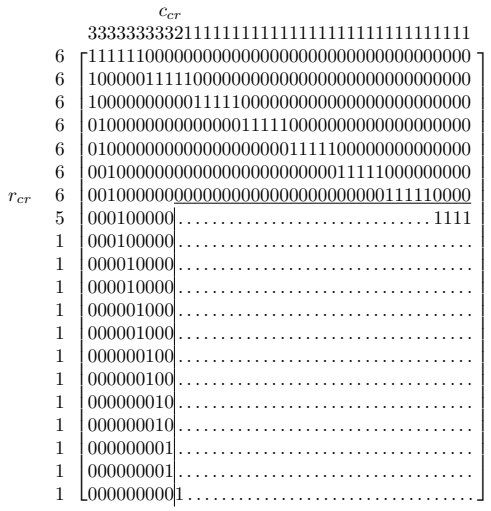

Figure 9: *VarFix* on a $(3, 6)$-regular $\mathbf{H}$ matrix of dimensions (20, 40)

constraints (2), and for r_{cr}-rows the constraints (3) are satisfied without deviation. We are left with a reduced rectangle of size $(m - r_{cr}) \times (n - c_{cr})$, which includes the unfixed x_{ij} variables shown as dots. *VarFix* in Figure 8 runs in $\mathcal{O}(nc_{cr})$ time. *VarFix* also helps to improve the error correction capability of a decoder since there is at least one neighboring check node for each variable node to control the correctness of the bit.

Proposition 1. *For given (m, n) dimensions, let F be the set of nonzero entries fixed by VarFix. As in Figure 10, let R be the reduced rectangle of size $(m - r_{cr}) \times (n - c_{cr})$ and $R \bigcup Q$ be the region between the entries in F. Let $\rho(i, j)$ be the local girth of the entry (i, j) when $x_{ij} = 1$, and $\tau = \max_{(i,j)\in Q}\{\rho(i, j)\}$. Then, for any (J, K)-regular $\mathbf{H}$ of dimensions (m, n) with girth $g > \tau$, the rows and columns can be reordered such that all nonzero entries are in F and R (see Appendix for the proof).*

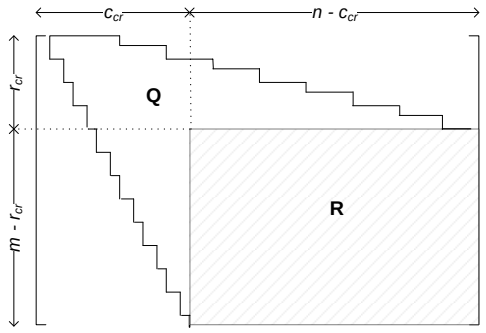

Figure 10: Reordered (J, K)-regular $\mathbf{H}$ matrix with girth $g > \tau$

In Proposition 1, we show that any (J, K)-regular $\mathbf{H}$ matrix of dimensions (m, n) that has sufficiently large girth g can be expressed as in Figure 10 by reordering its rows and columns. Hence, fixing the entries with *VarFix* does not eliminate any regular codes from the solution space.

Some characteristics of the cycles in a BG can be visualized by considering the BG given in Figure 6a and the corresponding matrix $\mathbf{H_1}$ in Figure 7. It can be seen that $S_1 = (v_1, c_1, v_3, c_2)$ and $S_2 = (c_1, v_1, c_2, v_2, c_3, v_3)$ are two cycles in the BG in Figure 6a. Figures 11a and 11b visualize cycles S_1 and S_2 on $\mathbf{H_1}$, respectively.

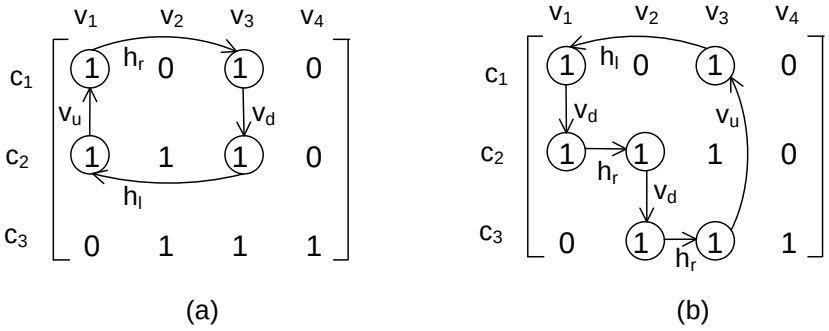

Figure 11: Cycles S_1 and S_2 on $\mathbf{H_1}$

We observe that a cycle is an alternating sequence of horizontal and vertical movements between cells having value 1. In particular, cycle S_1 is a sequence of horizontal right (h_r), vertical down (v_d), horizontal left (h_l), and vertical up (v_u) movements. Similarly, cycle S_2 can be expressed with the sequence $(v_d, h_r, v_d, h_r, v_u, h_l)$. Moreover, we deduce that a cycle should include at least one from each of the h_u, h_d, v_u, and v_d movements.

Proposition 2. *VarFix on* $\mathbf{H}$ *matrix does not form any cycles in the BG (see Appendix for the proof).*

We can use the partial solution obtained with *VarFix* in Figure 8 to generate a feasible solution of MDD. Since the partial solution does not include any cycles (see Proposition 2), setting the nonfixed entries to zero yields a feasible solution (an upper bound). We implement *VarFix* at the beginning of our B&C algorithm. *VarFix* shrinks the problem size by $\frac{(m-r_{cr})\times(n-c_{cr})}{m\times n}\times 100\%$ and provides an initial upper bound to the B&C algorithm.

3.3 Valid Inequalities for Cycle Regions

After applying *VarFix*, MDD problem reduces to locating ones in the reduced rectangle R of size $(m - r_{cr}) \times (n - c_{cr})$ in Figure 10. In this section, we introduce valid inequalities for (J, K)-regular codes to further improve the performance of our B&C algorithm.

First we divide the region $R \bigcup Q$ in Figure 10 into *subblocks* having $(J-1)(K-1)$ rows and $(K-1)$ columns as shown in Figure 12. For each entry (i, j) in a subblock, we investigate the local girth $\rho(i, j)$ by temporarily assuming $x_{ij} = 1$. For example, in Figure 12, we observe that $\rho(i, j)$ is common for all (i, j) entries in a subblock except the subblocks at the boundaries of the fixed 1s. We define a *cycle-α region*, which has a repeating pattern due to (J, K)-regularity, as the collection of (i, j) entries that have $\rho(i, j) = \alpha$.

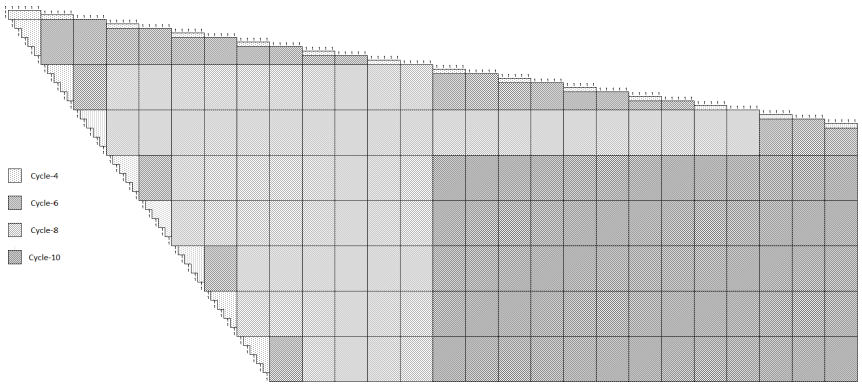

Figure 12: Subblocks and cycle regions with $J = 3$ and $K = 6$

In particular, when there is a 1 in a cycle-4 region (dotted area), we have a cycle of length 4 (see cycles C_1 and C_2 in Figure 13). We note that cycle-4 regions repeat both horizontally and vertically.

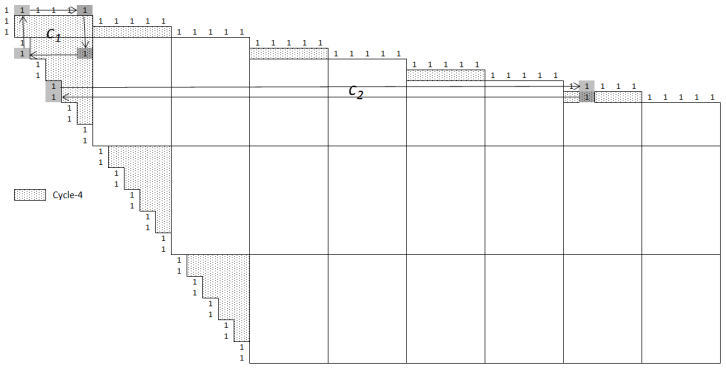

Figure 13: Cycle-4 region with $J = 3$ and $K = 6$

Similar horizontal and vertical repeating patterns can be seen for cycle-6 and cycle-8 regions

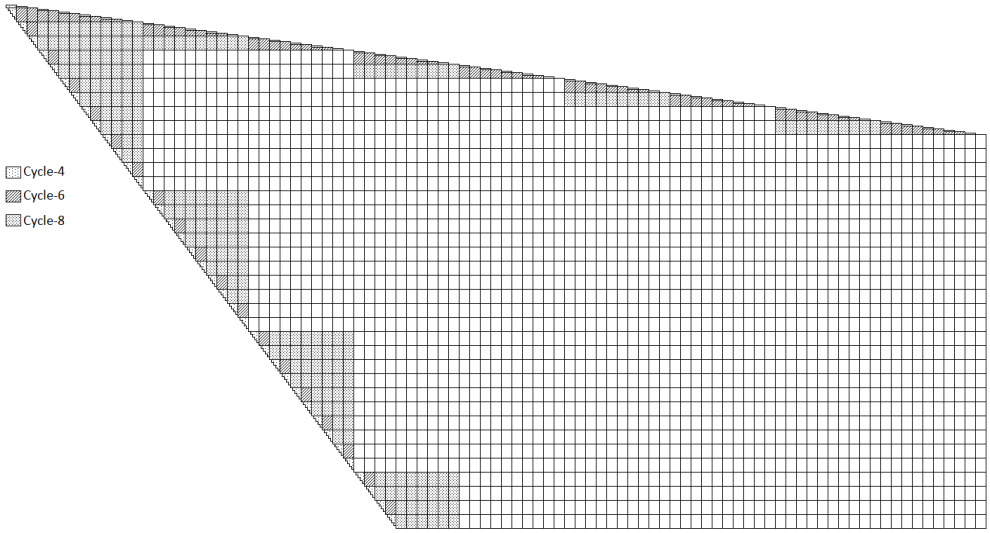

Figure 14: Cycle-4, cycle-6, and cycle-8 regions with $J = 3$ and $K = 6$

in Figure 14. Making use of these patterns, one can express $\rho(i, j)$ of an entry (i, j) as a function. We introduce valid inequalities for MDD based on $\rho(i, j)$ of the entries in the rectangle R.

Proposition 3. *Let $(i, j) \in R$, i.e., $i \in \{m - r_{cr}, ..., m\}$ and $j \in \{n - c_{cr}, ..., n\}$, and let $\rho(i, j)$ represent the local girth of the entry. Let σ denote the number of subblocks that intersect with R, and let B_s, $s \in \{1, ..., \sigma\}$ represent the set of (i, j) entries in the subblock s.*

(1) If $\rho(i, j) < g$, then the constraint

$$x_{ij} = 0 \tag{9}$$

is valid.

(2) If $g \geq 8$ and $(i, j) \in B_s$ with $\rho(i, j) = 8$ or 10, then the constraints

$$\sum_{i=1}^{J-1} \sum_{((k-1)(J-1)+i,j) \in B_s} x_{(k-1)(J-1)+i,j} \leq 1, \quad k \in \{1, ..., K-1\} \tag{10}$$

are valid.

(3) If $g \geq 10$ and $(i, j) \in B_s$ with $\rho(i, j) \geq 10$, then the constraint

$$\sum_{(i,j) \in B_s} x_{ij} \leq 1 \tag{11}$$

is valid.

Proof. Let us consider each claim separately.

(1) There cannot be cycles of length smaller than the girth g. If $x_{ij} = 1$, then we have a cycle of length $\rho(i,j) < g$, which is not desired. Hence, $x_{ij} = 0$ in this case.

(2) If $g \geq 8$, then there should not be any cycles of length 6. Let us consider a subblock with cycle region 8 or 10, which is subdivided into $(K-1)$ equal *subpieces* each having $(J-1)$ rows. In Figure 15, we give an example for a cycle-8 subblock with $J = 3$ and $K = 6$ where we have $(K-1) = 5$ subpieces each having $(J-1) = 2$ rows. As seen in Figure 15, a cycle of length 6 forms when there is more than one nonzero entry in a subpiece.

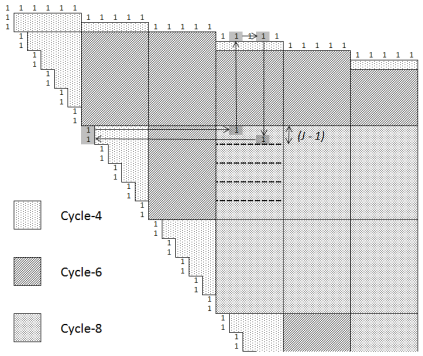

Figure 15: A cycle of length 6 on cycle-8 region with $J = 3$ and $K = 6$

A similar case appears for cycle-10 subblocks. Hence, the constraints (10) are valid, since they ensure the existence of at most one nonzero entry in each subpiece when the cycle region of the subblock is either 8 or 10.

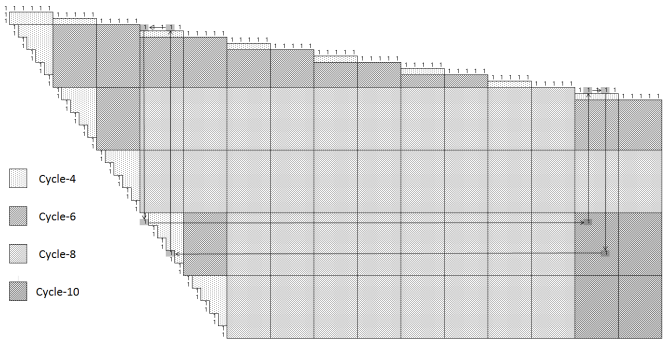

Figure 16: A cycle of length 8 on cycle-10 region with $J = 3$ and $K = 6$

(3) A cycle of length 8 is not allowed when $g \geq 10$. However, when there is more than one nonzero entry in a subblock with a cycle region of at least 10, there is a cycle of length 8 as given in Figure 16. The constraint (11) is valid, since it bounds the number of nonzero entries from above by 1. □

Proposition 4. *Let z^* be the optimal objective value of MDD and z_f^* be the optimal objective value of MDD when VarFix is applied. Let τ be defined as in Proposition 1. Assume that there exists a (J, K)-regular code with dimensions (m, n), then*

(1) $0 = z^* = z_f^*$ *if* $g > \tau$,

(2) $0 = z^* \leq z_f^*$ *if* $g \leq \tau$ *(see Appendix for the proof).*

Since the number of valid inequalities (9), (10), and (11) is polynomial in dimensions (m, n), i.e., $\mathcal{O}(c_{cr}m + r_{cr}n)$, we add them at the beginning of the B&C in order to obtain a tight lower bound at the root node.

3.4 Feasible Solution Generation Heuristics

In this section, we provide our heuristic approaches to generate feasible solutions of MDD. In particular, we adapt a heuristic from the telecommunications literature in Section 3.4.1, and describe how to polish given feasible solutions in Section 3.4.2.

3.4.1 Modified Progressive Edge Growth Algorithm

The last improvement to our B&C algorithm is to introduce a starting solution that will provide an initial upper bound. For this purpose, we adapt an existing algorithm from the literature known as Progressive Edge Growth (PEG) algorithm [17]. We modify this algorithm for our problem by starting PEG from a partial initial solution generated by our *VarFix* in Figure 8, and ensuring that the generated solution has girth at least g.

In our modified PEG (*mPEG*) (see Figure 17), **dv** and **dc** are the target degree vectors for the variable and check nodes, respectively. Let the deviation from the target degrees for the variable and check nodes be given by the slack vectors **s** and **t**, and the current degrees of the variable nodes be listed in the vector **d**. Moreover, let $\mathcal{N}_j^l$ represent the set of all check nodes that can be reached from v_j with a tree of depth l. Hence, the set $\mathcal{N}_j^l \setminus \mathcal{N}_j^{l-1}$ consists of the check nodes that are reached at the lth step from v_j for the first time. We can represent the check nodes in the set $\mathcal{N}_j^l$ with an incidence vector $\mathcal{I}$ as $\mathcal{I}_{c_i} = 1$ if $c_i \in \mathcal{N}_j^l$, and zero otherwise.

Starting from the solution provided by *VarFix* (Step 1), *mPEG* inserts edges to the BG for each variable node v_j with positive slack s_j (Steps 2 and 3). An edge $\{c_i, v_j\}$ is a candidate if the slacks t_i and s_j are positive, and adding the edge does not form a cycle smaller than g. If there are some check nodes that the BFS tree cannot reach, i.e., $|\mathcal{N}_j^l| \leq m$, then we can add the

Input: (m, n) dimensions, **dv** and **dc** vectors, g target girth

0. Initialize $\mathbf{x} \leftarrow \mathbf{0}$, $\mathbf{d} \leftarrow \mathbf{0}$, $\mathbf{s} \leftarrow \mathbf{dv}$, and $\mathbf{t} \leftarrow \mathbf{dc}$, $\mathcal{I} \leftarrow \mathbf{0}$.
1. Apply *VarFix* (Figure 8), and update slacks
   $s_j \leftarrow s_j - \sum_i x_{ij}$ for all j and $t_i \leftarrow t_i - \sum_j x_{ij}$ for all i,
   and current degrees $d_j \leftarrow \sum_i x_{ij}$ for all j.
2. **For** $j \in \{1, ..., n\}$, set $\mathcal{I} \leftarrow \mathbf{0}$
3. **For** $k \in \{0, ..., d_j\}$ with $d_j < d(v_j)$
4. **If** $k = 0$, **Then** set $x_{i'j} = 1$ for $i' = \text{argmax}_i\{t_i\}$.
5. **Else** apply BFS from v_j to reach check nodes, let the search tree have depth l.
6. **If** $2l \geq g$ or $|\mathcal{N}_j^l| \leq m$, let $\mathcal{I}$ be incidence vector for $\mathcal{N}_j^l$.
   If $\exists i' = \text{argmax}_i\{(1 - \mathcal{I}_{c_i})t_i\}$, **Then** set $x_{i'j} = 1$. **Else break**.
7. **End If**
8. Update d_j, s_j, t_i as in Step 1.
9. **End For**
10. **End For**

Output: An initial solution for MDD

Figure 17: Modified PEG (*mPEG*) algorithm

edge $\{c_i, v_j\}$ without forming any cycles. Otherwise, if there are check nodes that cannot be reached by a tree with depth of $l \geq g/2$, then the corresponding edge will form a cycle of length $2l \geq g$. Among the candidate check nodes, we pick the one with the maximum slack t_i in order to fit the target degree $d(c_i)$ (Steps 5 and 6). The generated solution is feasible for MDD, since it has girth at least g. Moreover, it yields an initial upper bound for our B&C algorithm. The time complexity of *mPEG* is $\mathcal{O}((m+n)|E_x| + |E_x|^2)$, where $G_x = (V \cup C, E_x)$ is the induced graph by the nonzero x_{ij} values after *VarFix*.

3.4.2 Solution Polishing Algorithms

In this section we introduce our *Polish* heuristic to improve the given feasible solutions of MDD. *Polish* consists of three subheuristics, i.e., *CombineCodes*, *DeleteEdge*, and *AddEdge*.

CombineCodes combines two (J, K)-regular codes with dimensions (m_1, n_1) and (m_2, n_2) with $m_1 + m_2 \geq m$ and the girth of at least g for given (m, n), (J, K), and girth g values. We augment the two codes diagonally and remove the last $m_1 + m_2 - m$ rows and $n_1 + n_2 - n$ columns to fit the dimensions.

DeleteEdge picks the codes with the girth smaller than g, and deletes edges until we end up with a BG with girth g. Deleting an edge $\{c_i, v_j\}$ from a BG increases the slacks t_i and s_j, and the objective function value. Hence, we choose an edge $\{c_i, v_j\}$, which is in a small cycle, i.e., small $\rho(i, j) < g$, has small degree deviations t_i and s_j, and small objective function coefficients $\frac{1}{d(c_i)}$ and $\frac{1}{d(v_j)}$. This means, we choose the entry (i^*, j^*) to delete, i.e., set $x_{i^*j^*} = 0$, using

Equation (12).

$$(i^*, j^*) = \text{argmin}_{(i,j)} \left\{ \rho(i,j) \left(\frac{t_i}{d(c_i)} + \frac{s_j}{d(v_j)} \right) \right\} \tag{12}$$

AddEdge inserts edges to a BG with girth of larger or equal to g without forming cycles smaller than g. Adding an edge $\{c_i, v_j\}$ to a BG improves the objective function value by decreasing the slacks t_i and s_j. We prefer to add an edge $\{c_i, v_j\}$, which forms a large cycle, i.e., large $\rho(i,j) \geq g$, has large degree deviations t_i and s_j, and large objective function coefficients $\frac{1}{d(c_i)}$ and $\frac{1}{d(v_j)}$. Hence, we choose the entry (i^*, j^*) to add, i.e., set $x_{i^*j^*} = 1$, using Equation (13).

$$(i^*, j^*) = \text{argmax}_{(i,j)} \left\{ \rho(i,j) \left(\frac{t_i}{d(c_i)} + \frac{s_j}{d(v_j)} \right) \right\} \tag{13}$$

Polish runs at the beginning of our B&C method by implementing the heuristics in the order of *CombineCodes*, *DeleteEdge*, and *AddEdge* to obtain a tight upper bound for MDD.

4 Computational Results

We carry out the computations on a computer with a 2.0 GHz Intel Xeon E5-2620 processor and 46 GB of RAM working under a Windows Server 2012 R2 operating system. In the computational experiments, we used CPLEX 12.8.0 to test the performance of the B&C algorithm and to evaluate how various improvement strategies given in Section 3 affect the results. We implemented all algorithms in C++ programming language. We summarize the solution methods in Table 2.

Table 2: Summary of the solution methods

Method	*IntSep*	*FracSep*	*VarFix*	Valid Cuts	*mPEG*	*Polish*
BC_0	√	√	-	-	-	-
BC_1	√	√	√	-	-	-
BC_2	√	√	√	√	-	-
BC_3	√	√	√	√	√	√

In BC_0, we apply our B&C algorithm with *IntSep* and *FracSep* to separate integral and fractional solutions, respectively (see Section 2.2). In CPLEX, we implement *IntSep* and *FracSep* with a *Callback* routine, and utilize default branching settings. In the BC_1 method, *VarFix* fixes some of the entries of the **H** matrix at the beginning of the B&C (see Section 3.2). In the BC_2 method, we apply *VarFix* and add the valid inequalities at the root node of the B&C (see

Section 3.3). Finally in the BC_3 method, in addition to utilizing *VarFix* and the valid cuts, the B&C starts from the best feasible solution provided by *mPEG* and *Polish* (see Section 3.4).

Table 3: List of the computational parameters

Parameters	
(J, K)	$(3, 6)$ and $(5, 10)$-regular BGs
(m, n)	(10, 20), (15, 30), (20, 40), (30, 60), (40, 80), (100, 200), (150, 300), (250, 500), (500, 1000)
g	6, 8, 10
Time Limit	10800 secs

We list the parameters used in the computational experiments in Table 3. In practical applications small n values are desired since the information is received in packets of length n. In IEEE 802.11 WLAN 2007 standards, the codes have length $n \in (500, 2300)$, $J/K = 1/2$, and the girth at least 6 [31]. Recent works propose codes of length $n \in (250, 500)$ in order to receive data in smaller packet sizes [23]. In our experiments, we consider $(3, 6)$ and $(5, 10)$-regular $\mathbf{H}$ matrices with girth values $g = 6, 8$, or 10. We test nine different (m, n) dimensions from $n = 20$ to 1000. Since $(5, 10)$-regular codes are denser, we try larger n values, i.e., $n \geq 200$. We report the results that CPLEX found in a time limit of 10800 seconds.

Table 4: Computational results for BC_0

						CPU	Gap		#Cuts		#Calls		CPU (%)		Actual
g	J	n	z_l	z	z_u^i	(secs)	(%)	Nodes	Int	Frac	Int	Frac	Int	Frac	g
6	3	20	0	6.5	30	*time*	100	255556	7970	0	872	495073	0.03	53.06	6
		30	0	3	45	60.39	100	2927	9725	0	1189	4338	8.64	83.48	6
		40	0	0	60	0.27	0	3	125	0	17	2	33.46	13.24	6
		60	0	0	90	0.95	0	10	172	0	31	11	28.75	45.03	6
		80	0	0	120	1.11	0	5	182	0	27	8	28.21	48.07	6
		200	0	0	300	0.96	0	0	117	0	6	1	26.54	40.75	6
		300	0	0	450	3.53	0	1	170	0	7	2	16.11	50.44	6
		500	0	0	750	6.18	0	0	153	0	6	1	23.40	40.14	6
		1000	0	0	1500	153.81	0	7	166	0	10	6	29.15	52.39	6
	5	200	0	291.3	300	*time*	100	7545	523558	0	4849	10078	4.80	14.30	10
		300	0	411.6	450	*time*	100	5152	598856	0	9059	3618	18.47	11.90	10
		500	0	0	750	146.64	0	50	13301	0	324	24	70.81	11.85	6
		1000	0	0	1500	341.37	0	30	1613	0	28	32	29.47	46.72	6
8	3	20	0	7.5	30	*time*	100	131931	68109	15103	2002	253443	0.06	11.48	10
		30	0	10	45	*time*	100	31627	167270	7397	2367	57591	0.13	3.22	8
		40	0	16	60	*time*	100	23712	213134	5308	2454	42933	0.20	3.57	8
		60	0	14.5	90	*time*	100	8609	294480	1685	1928	14287	0.31	2.31	8
		80	0	24	120	*time*	100	9671	421986	1866	2780	15210	0.74	4.07	8
		200	0	171	300	*time*	100	8602	468624	701	4039	11967	3.78	13.74	8
		300	0	213	450	*time*	100	7339	469399	357	5102	8663	8.49	19.09	8
		500	0	308	750	*time*	100	5223	371370	97	5856	4285	31.64	30.16	8
		1000	0	0	1500	511.96	0	59	1657	0	49	68	32.10	56.46	8
	5	300	0	352.8	450	*time*	100	1090	23642	110	17	2167	0.06	8.81	8
		500	0	672.9	750	*time*	100	195	30778	13	4	392	0.14	5.05	10
		1000	0	1253.4	1500	*time*	100	1	21743	0	3	3	0.94	0.15	10
10	3	20	0	6.5	30	*time*	100	67218	190343	73472	453	118026	0.06	5.89	10
		30	0	13.5	45	*time*	100	11916	516532	10526	1077	18449	0.25	1.93	10
		40	0	25	60	*time*	100	10161	582038	7162	1135	16002	0.32	2.15	10
		60	0	37.5	90	*time*	100	7783	812030	4514	1008	11785	0.57	3.16	10
		80	0	27.5	120	*time*	100	5701	549745	2691	684	9508	0.49	3.20	10
		200	0	122	300	*time*	100	2013	216090	610	305	3681	0.53	3.65	10
		300	0	184	450	*time*	100	761	33023	25	61	1517	0.16	4.13	10
		500	0	515	750	*time*	100	782	113682	136	149	1444	1.89	10.81	10
		1000	0	1218	1500	*time*	100	780	152898	103	179	1441	14.20	68.34	10
	5	300	0	450	450	*time*	100	0	255378	0	1	0	1.50	0	10
		500	0	750	750	*time*	100	0	265866	0	1	0	2.61	0	10
		1000	0	1500	1500	*time*	100	0	271433	0	1	0	12.52	0	10

From Table 4 to 7, the column "z" is the best upper bound of MDD at termination and the column "z_l" is the best known lower bound found by CPLEX within the time limit. For each of the methods, we have an initial feasible solution (an upper bound) with the objective value z_u^i. In the BC_0 method, $\mathbf{H} = \mathbf{0}$ is a trivial solution providing an initial upper bound. In the methods BC_1, BC_2 and BC_3, initial feasible solutions are obtained from *VarFix* (see Section 3.2), *mPEG* or *Polish* heuristics (see Section 3.4), respectively. The computational time in seconds is given in the column "CPU (secs)" and the percentage difference among z_l and z is presented in the column "Gap (%)." *VarFix*, adding the valid cuts, *mPEG*, and *Polish* methods take negligible time, and are not reported in the tables. The "Nodes" column gives the number of nodes processed by the B&C algorithm. The "#Cuts" column summarizes the number of *IntSep* and *FracSep* cuts generated while "#Calls" shows the number of times that our *Callback* routine is called by CPLEX for finding these cuts. We report the percentage of time within the total time "CPU (secs)" consumed for the *IntSep* and *FracSep* algorithms in the column "CPU (%)." The column "Actual g" shows the girth of the generated BG within the time limit.

As discussed in Section 2.1, we have a (J, K)-regular code if $z_l = z = 0$. We can conclude that it is not possible to have a (J, K)-regular code with given (m, n) and the girth g when we have $z \geq z_l > 0$. In Table 4, BC_0 finds (J, K)-regular code for 10 instances ($z = 0$). For the instances that cannot be solved to optimality (Gap > 0), the number of nodes processed decreases as n gets larger. On average, *FracSep* algorithm is called more frequently (see #Calls) and takes more time than *IntSep* (see CPU(%)) whereas the number of the *IntSep* cuts are more than the *FracSep* cuts (see #Cuts). Hence, *IntSep* is more effective than *FracSep* in generating the constraints (4). As g gets larger, we observe that the CPU(%) values of *IntSep* and *FracSep* decrease since CPLEX consumes the majority of the total time.

Table 5: Computational results for BC_1

						CPU	Gap		# Cuts		# Calls		CPU (%)		Actual
g	J	n	z_l	z	z_u^i	(secs)	(%)	Nodes	Int	Frac	Int	Frac	Int	Frac	g
6	3	20	3.25	6.5	15.5	*time*	50	5299967	250	0	51	10221752	0.01	90.44	6
		30	0.25	2	23	*time*	87.5	2407393	2496	0	668	4671452	0.01	88.31	6
		40	0	0	30.5	0.14	0	1	152	0	14	1	10.71	0	6
		60	0	0	45.5	0.19	0	0	130	0	13	2	17.02	17.02	6
		80	0	0	60.5	0.25	0	3	120	0	18	3	31.60	6.40	6
		200	0	0	150.5	0.58	0	0	130	0	12	1	40.31	10.90	6
		300	0	0	225.5	1.23	0	0	153	0	6	2	26.58	22.77	6
		500	0	0	375.5	3.56	0	0	209	0	6	2	30.26	24.12	6
		1000	0	0	750.5	31.77	0	0	139	0	5	2	47.96	24.99	6
	5	200	0	189	210.3	*time*	100	13883	337710	0	4763	22414	4.17	29.78	6
		300	0	0	315.3	4108.08	0	3814	327121	0	7641	2710	30.48	19.13	6
		500	0	0	525.3	48.54	0	20	3989	0	112	16	61.53	22.18	6
		1000	0	0	1050.3	194.30	0	13	1724	0	25	14	45.96	34.45	6
8	3	20	15.5	15.5	15.5	0.06	0	0	0	0	1	0	0	0	10
		30	21.5	21.5	21.5	1.39	0	666	268	0	29	918	0	80.88	8
		40	13	21.5	30.5	*time*	39.5	1051701	2341	10	147	2048148	0	85.91	8
		60	3	16	45.5	*time*	81.3	185607	42013	0	1952	362494	0.14	35.52	8
		80	0	16.5	60.5	*time*	100	64513	112803	0	3159	126280	0.44	23.49	8
		200	0	133	150.5	*time*	100	14284	347040	0	4686	22792	3.02	23.10	8
		300	0	183.5	225.5	*time*	100	11541	397646	0	7290	15148	11.73	35.24	8
		500	0	0	375.5	556.28	0	602	49313	0	1082	374	54.73	33.63	8
		1000	0	0	750.5	2600.96	0	330	1097	0	42	606	4.83	89.27	8
	5	300	21	311.1	315.3	*time*	93.2	2251	393396	0	111	4492	1.46	13.48	8
		500	0	520.5	525.3	*time*	100	572	38731	0	43	981	0.33	10.16	8
		1000	0	1050.3	1050.3	*time*	100	92	21719	0	9	169	1.20	10.77	10
10	3	20	15.5	15.5	15.5	0.08	0	0	0	0	1	0	0	0	10
		30	23	23	23	0.08	0	0	0	0	1	0	0	0	10
		40	30.5	30.5	30.5	0.11	0	0	0	0	1	0	0	0	10
		60	45.5	45.5	45.5	0.11	0	0	0	0	1	0	0	0	10
		80	59	59	60.5	696.07	0	38404	3138	19	498	42661	0.43	90.26	10
		200	16.5	148	150.5	*time*	88.9	6729	357143	12	659	12886	0.98	13.35	10
		300	0	219	225.5	*time*	100	5364	555047	0	663	10116	3.13	24.77	10
		500	0	375.5	375.5	*time*	100	4447	764122	0	762	8142	9.36	55.67	10
		1000	0	750.5	750.5	*time*	100	1058	209274	0	230	1884	17.02	77.73	10
	5	300	315.3	315.3	315.3	0.13	0	0	0	0	1	0	37.60	0	10
		500	369.3	525.3	525.3	*time*	29.7	748	1645923	93	76	1255	6.75	11.99	10
		1000	294.3	1050.3	1050.3	*time*	72	297	639635	3	17	575	29.26	30.11	10

BC_1 solves 20 instances to optimality as given in Table 5. We note that for 16 instances there is no (J, K)-regular code since $z_l > 0$. Compared with BC_0, we have tighter z_u^i values provided by *VarFix*, and we improve the best known upper bound z for 26 instances. Moreover, we process more nodes by adding fewer *IntSep* and *FracSep* cuts, and improve the computation time by 32% on average.

Table 6: Computational results for BC_2

						CPU	Gap		# Cuts		# Calls		CPU (%)		Actual
g	J	n	z_l	z	z_u^i	(secs)	(%)	Nodes	Int	Frac	Int	Frac	Int	Frac	g
6	3	20	3.25	6.5	15.5	*time*	50	4835210	246	0	48	9361686	0.01	89.44	6
		30	0.25	2	23	*time*	87.5	2344931	2506	0	669	4501860	0.01	87.48	6
		40	0	0	30.5	0.14	0	1	152	0	14	1	33.33	0	6
		60	0	0	45.5	0.16	0	0	130	0	13	2	9.62	10.26	6
		80	0	0	60.5	0.27	0	3	120	0	18	3	35.71	5.64	6
		200	0	0	150.5	0.63	0	0	130	0	12	1	42.24	10.08	6
		300	0	0	225.5	1.27	0	0	153	0	6	2	28.36	24.72	6
		500	0	0	375.5	5.27	0	0	209	0	6	2	40.67	23.16	6
		1000	0	0	750.5	43.05	0	0	139	0	5	2	52.13	25.80	6
	5	200	0	184.8	210.3	*time*	100	13952	338378	0	4863	22468	4.48	30.61	6
		300	0	0	315.3	3368.63	0	3335	305752	0	6891	2202	35.15	19.32	6
		500	0	0	525.3	51.89	0	21	4519	0	115	14	66.64	17.73	6
		1000	0	0	1050.3	190.10	0	12	1727	0	21	10	52.28	31.48	6
8	3	20	15.5	15.5	15.5	0.11	0	0	0	0	1	0	0	0	10
		30	21.5	21.5	23	0.14	0	0	0	0	1	0	0	0	10
		40	21.5	21.5	30.5	24.74	0	5892	441	0	68	6348	0.25	87.70	8
		60	7	14	45.5	*time*	50	136315	61461	0	2001	264944	0.24	39.63	8
		80	1.5	16.5	60.5	*time*	90.9	44405	111784	0	2080	85627	0.34	17.83	8
		200	0	99.5	150.5	*time*	100	16276	298919	0	5063	25810	5.70	35.76	8
		300	0	175	225.5	*time*	100	10182	328829	0	6456	12884	10.52	30.01	8
		500	0	0	375.5	1316.96	0	1075	79070	0	2144	486	67.86	23.36	8
		1000	0	0	750.5	257.44	0	27	1149	0	35	20	54.27	37.72	8
	5	300	83.4	310.5	315.3	*time*	73.2	1797	307602	0	137	3246	1.01	8.54	8
		500	1.5	452.4	525.3	*time*	99.7	346	18107	0	36	610	0.21	6.54	8
		1000	0	1050.3	1050.3	*time*	100	31	10716	0	2	61	0.54	5.09	10
10	3	20	15.5	15.5	15.5	0.13	0	0	0	0	1	0	0	0	10
		30	23	23	23	0.17	0	0	0	0	1	0	0	0	10
		40	30.5	30.5	30.5	0.16	0	0	0	0	1	0	9.55	0	10
		60	45.5	45.5	45.5	0.16	0	0	0	0	1	0	10.26	0	10
		80	59	59	60.5	0.17	0	0	1	0	2	0	8.72	0	10
		200	65	99.5	150.5	*time*	34.7	31037	174760	10	2258	58528	2.95	83.81	10
		300	26	205.5	225.5	*time*	87.4	5471	539611	0	845	9556	3.48	23.64	10
		500	0	354.5	375.5	*time*	100	2102	277233	0	308	3750	4.25	33.01	10
		1000	0	750.5	750.5	*time*	100	567	91086	0	94	1046	10.70	68.23	10
	5	300	315.3	315.3	315.3	0.15	0	0	0	0	1	0	31.97	0	10
		500	516.3	516.3	525.3	104.95	0	63	1813	0	401	30	73.49	11.30	10
		1000	861.3	1050.3	1050.3	*time*	18	1640	86147	9	26	2067	4.25	91.96	10

BC_2 utilizes the valid cuts (1), (2), and (3) given in Proposition 3 (see Section 3.3). Note that the cuts (1) fix some of the x_{ij} variables to zero. The cuts (2) are valid for $g \geq 8$, whereas the cuts (3) are applicable for $g \geq 10$. In Table 6, we report the results when we add the valid cuts (1), (2), and (3) wherever they are applicable. Compared with BC_1, BC_2 improves z_l for 9 instances, z for 10 instances, and the CPU time decreases by 13% on average. Furthermore, the number of processed nodes decreases by 18%, whereas the number of *IntSep* cuts is reduced by 51%, and the *FracSep* cuts decreases by 86% on average. BC_2 solves 22 instances to optimality and concludes that there cannot be a (J, K)-regular code with the given dimensions for 19 instances ($z_l > 0$).

We also investigate the contributions of each valid cut family on z_l, z, and time figures by gradually including them to BC_2. As we report in Table 9, we first use only the valid cuts (1), then (1) and (2), and finally (1), (2) and (3) together (see Appendix). Compared with BC_1, using only the valid cuts (1) improves z_l for one instance, and z for 3 instances. Adding the valid cuts (2), we further improve z_l for 5 instances and tighten z for 6 instances while saving time and separation cuts. We observe an improvement on z_l for 4 instances and z for 3 instances

when we also utilize the valid cuts (3). The number of valid cuts (1), (2), and (3) used in BC_2 are given in the last three columns of Table 9.

Table 7: Computational results for BC_3

					z_u^i		CPU	Gap		# Cuts		# Calls		CPU (%)		Actual
g	J	n	z_l	z	*mPEG*	*Polish*	(secs)	(%)	Nodes	Int	Frac	Int	Frac	Int	Frac	g
6	3	20	3.25	4	6.5	4^a	*time*	18.75	2980922	167	1048962	37	5566509	0	88.25	6
		30	0.25	2	2	–	*time*	87.5	1302865	721	3347818	213	2066319	0	77.04	6
		40	0	0	0.5	0^a	0.08	0	0	0	0	1	0	2.47	0	6
		60	0	0	0.5	0^c	0.29	0	0	0	0	1	0	17.01	0	6
		80	0	0	0.5	0^c	0.22	0	0	0	0	1	0	29.41	0	6
		200	0	0	1	0^a	0.21	0	0	0	0	1	0	14.08	0	6
		300	0	0	0.5	0^a	0.35	0	0	0	0	1	0	20.06	0	6
		500	0	0	0.5	0^c	3.29	0	0	0	0	1	0	46.46	0	6
		1000	0	0	1	0^c	27.05	0	0	0	0	1	0	57.19	0	6
	5	200	0	0.9	2.4	0.9^a	*time*	100	12773	232692	871827	3581	11594	2.75	12.98	6
		300	0	0	2.1	0^a	0.67	0	0	0	0	1	0	44.94	0	6
		500	0	0	2.7	0^a	1.81	0	0	0	0	1	0	41.39	0	6
		1000	0	0	1.5	0^c	1.66	0	0	0	0	1	0	17.56	0	6
8	3	20	15.5	15.5	15.5	–	0.07	0	0	0	0	1	0	0	0	10
		30	21.5	21.5	21.5	–	0.07	0	0	0	0	1	0	0	0	8
		40	21.5	21.5	21.5	–	0.91	0	0	0	0	1	0	0.77	0	8
		60	7	9.5	16.5	9.5^a	*time*	26.31	0	1869	102172	24	1494	0.01	2.99	8
		80	1.63	9.5	9.5	–	*time*	82.89	0	891	127852	11	1060	0.01	4.98	8
		200	0	0	4	0^a	0.41	0	0	0	0	1	0	21.48	0	8
		300	0	0	2.5	0^a	0.31	0	0	0	0	1	0	0.00	0	8
		500	0	0	2.5	0^c	0.84	0	0	0	0	1	0	27.87	0	8
		1000	0	0	3	0^c	0.87	0	0	0	0	1	0	40.23	0	8
	5	300	83.4	136.8	136.8	–	*time*	39.03	0	23821	180035	11	48	0.07	10.24	8
		500	1.5	150.9	150.9	–	*time*	99.01	0	7148	53622	3	7	0.07	5.71	8
		1000	0	106.2	106.2	–	*time*	100	0	6833	43110	1	5	0.31	3.57	8
10	3	20	15.5	15.5	15.5	–	0.06	0	0	0	0	1	0	1.59	0	10
		30	23	23	23	–	0.07	0	0	0	0	1	0	0	0	10
		40	30.5	30.5	30.5	–	0.07	0	0	0	0	1	0	0	0	10
		60	45.5	45.5	45.5	–	0.06	0	0	0	0	1	0	1.56	0	10
		80	59	59	59	–	0.08	0	0	0	0	1	0	6.25	0	10
		200	65	76	78.5	76^a	*time*	14.47	0	13288	247672	66	867	0.03	4.78	10
		300	26	68.5	68.5	–	*time*	62.04	0	9736	643235	24	567	0.05	11.67	10
		500	0	43.5	43.5	–	*time*	100	0	3261	335890	4	180	0.04	17.98	10
		1000	0	15	15	–	*time*	100	0	2918	191128	2	112	0.27	22.14	10
	5	300	315.3	315.3	315.3	–	0.14	0	0	0	0	1	0	33.33	0	10
		500	516.3	516.3	516.3	–	0.76	0	0	0	0	1	0	44.69	0	10
		1000	861.3	875.4	875.4	–	*time*	1.61	0	70712	150362	668	1186	27.03	49.45	10

The BC_3 method first implements *mPEG* and then *Polish* heuristics to start with a tight upper bound (see Section 3.4). We populate the feasible solutions that we have obtained from BC_0 to BC_2 in a solution pool for the *Polish* heuristic. In Table 7, the "z_u^i" column gives the initial upper bounds of *mPEG* and *Polish*. The symbol "–" indicates that the upper bound of *Polish* is the same as *mPEG*, the label "*a*" shows that the best solution is obtained from *AddEdges*, and the label "*c*" means that *CombineCodes* generates the best upper bound. *Polish* can find better feasible solutions than *mPEG* for 18 instances.

Compared with BC_2, BC_3 improves z_l for one instance, z for 14 instances, and the time decreases by 16% on average. The percentages of the time consumed by *IntSep* and *FracSep* decrease by 28% and 68% on average, respectively. Better initial upper bounds allow us to generate the constraints (4) by fewer calls to the separation algorithms.

Among the methods from BC_0 to BC_3, we can see that BC_3 solves the highest number of instances to optimality (24 instances out of 37 instances) in the shortest time. Furthermore, BC_3

provides a computational evidence that there cannot be a (J, K)-regular code (when $z_l > 0$) for 19 instances within the given time limit. BC_3 can determine the smallest n of a (J, K)-regular code with girth g by starting from a small n and gradually increasing it while $z_l > 0$.

Taking into account that the code design problem is an offline problem, one can implement the BC_3 method to construct a (J, K)-regular code providing sufficiently large time. Furthermore, BC_3 can generate alternative (J, K)-regular codes with given (m, n) dimensions and girth g. For example, let $\mathbf{H}$ be a code generated by BC_3. We can randomly pick an entry (i, j) with $H_{ij} = 1$, and resolve the MDD model after adding the constraint $x_{ij} = 0$ via the *Callback* routine of CPLEX to obtain another code.

In Table 8, we compare the performance of BC_3 with some commonly used methods from the literature. The (J, K)-regular codes in [29] and [30] are generated by carrying out complete enumeration, which is the common method for code generation in the telecommunications literature, for the mentioned (m, n) dimensions in Table 8. There are several criteria to pick the best LDPC code: small (m, n) dimensions and regular degree distributions (easier hardware implementation), small (J, K) values (faster decoding with a sparse code), and high girth g (better error correction). Since the code design is an offline problem, usually months are reserved to choose an LDPC code and the algorithm run time is not reported in the literature.

Table 8: Comparison of the BC_3 method with the existing methods in [29] and [30]

		Existing Codes			BC_3			
(J, K)	(m, n)	Reference	z	g	z	z_u^i	CPU (secs)	g
(2, 3)	(14, 21)	[29]	0	12	0	35	0.77	12
(3, 4)	(78, 104)	[29]	0	6	0	1.8	4.95	8
(3, 5)	(93, 155)	[29]	0	8	0	2.1	569.41	8
(3, 4)	(27, 36)	[30]	0	8	0	39.1	2352.01	8
(3, 5)	(39, 65)	[30]	0	8	0	66.7	18638.64	8
(3, 6)	(54, 108)	[30]	0	8	0	6.5	26302.15	8
(4, 5)	(92, 115)	[30]	0	8	0	9.9	31649.13	8
(4, 6)	(96, 144)	[30]	0	8	0	23.3	51432.21	8
(4, 7)	(120, 210)	[30]	0	8	0	33.8	78594.53	8
(4, 8)	(156, 312)	[30]	0	8	0	43.1	112643.57	8
(5, 6)	(165, 210)	[30]	0	8	0	43.9	19455.85	8
(5, 7)	(265, 371)	[30]	0	8	0	48.7	287421.15	8

In Table 8, we consider the (J, K), (m, n), and g parameters used in [29] and [30], and implement our BC_3 method to solve the instances to optimality without any time limitations. We do not utilize *Polish* heuristic, since there is no solution pool. The z_u^i column is the initial upper bound obtained from *mPEG*. BC_3 needs 3.3 days, which is the longest time among the instances, to find a $(5, 7)$-regular code of dimensions (265, 371). Hence, we can say that our BC_3 method is a strong candidate to generate LDPC codes for the practical applications. Besides, it has the capability to detect the smallest (m, n) dimensions that one can obtain a (J, K)-regular code with the girth of at least g.

5 Conclusions

In this work, we investigate the LDPC code design problem and provide an IP formulation for the design of a bipartite graph (BG) with a given degree distribution. For the solution of the problem, we propose a branch-and-cut (B&C) algorithm. We analyze structural properties of the problem and improve our B&C algorithm by using techniques such as variable fixing, adding valid inequalities, and providing an initial solution using a heuristic. The computational experiments indicate that each of these techniques improves the B&C one step further. Among all, the method that combines all of these strategies, i.e., the BC_3 method, can solve the largest number of instances to optimality and gives the smallest gap values on average in an acceptable amount of time. One important gain of the method is that it can provide evidence as to whether a (J, K)-regular code with the given dimensions exists or not.

In this study, our focus has been on (J, K)-regular BGs. In telecommunication applications, irregular LDPC codes are also utilized. Hence, extending these techniques to irregular BGs can be a direction of future research. Spatially-coupled (SC) LDPC codes are another code family, which have become popular due to their channel capacity approaching error correction capability. The design of SC LDPC codes without small cycles will be a valuable contribution to future communication standards. Furthermore, a graph algorithm can perform better on higher girth graph instances as described in [16]. Hence, the BG instances generated with our methods can be used for the problems defined on BGs to have better results.

Acknowledgments

This research has been supported by the Turkish Scientific and Technological Research Council with grant no 113M499, and partially supported by Boğaziçi University Research Fund with grant no 14451P.

References

[1] Abreu, M., Araujo-Pardo, G., Balbuena, C., and Labbate, D. (2012). Families of small regular graphs of girth 5. *Discrete Mathematics*, 312(18):2832 – 2842.

[2] Ahuja, R. K. (2017). *Network Flows: Theory, Algorithms, and Applications.* Pearson Education, 1st edition.

[3] Araujo-Pardo, G. and Balbuena, C. (2010). Constructions of small regular bipartite graphs of girth 6. *Networks*, 57(2):121–127.

[4] Bandi, S., Tralli, V., Conti, A., and Nonato, M. (2011). On girth conditioning for low-density parity-check codes. *IEEE Transactions on Communications*, 59(2):357–362.

[5] Bocharova, I. E., Johannesson, R., and Kudryashov, B. D. (2014). A unified approach to optimization of LDPC codes for various communication scenarios. In *2014 8th International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, pages 243–248.

[6] Broulim, J., Davarzani, S., Georgiev, V., and Zich, J. (2016). Genetic optimization of a short block length LDPC code accelerated by distributed algorithms. In *2016 24th Telecommunications Forum (TELFOR)*, pages 1–4.

[7] Campello, J. and Modha, D. S. (2001). Extended bit-filling and LDPC code design. In *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*, volume 2, pages 985–989 vol.2.

[8] Chandran, L. (2003). A high girth graph construction. *SIAM Journal on Discrete Mathematics*, 16(3):366–370.

[9] Chang, H. and Lu, H. (2013). Computing the girth of a planar graph in linear time. *SIAM Journal on Computing*, 42(3):1077–1094.

[10] Chen, J., Dholakia, A., Eleftheriou, E., Fossorier, M. P. C., and Hu, X.-Y. (2005). Reduced-complexity decoding of LDPC codes. *IEEE Transactions on Communications*, 53(8):1288–1299.

[11] Costello Jr, D. J. (2009). An introduction to low-density parity check codes.

[12] Dantzig, G. B., Blattner, W., and Rao, M. (1966). Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem. Technical report, Stanford University Operations Research House.

[13] Diestel, R. (2010). *Graph Theory (Graduate Texts in Mathematics)*. Springer.

[14] Gschwind, T. and Irnich, S. (2011). A note on symmetry reduction for circular traveling tournament problems. *European Journal of Operational Research*, 210(2):452 – 456.

[15] He, X., Zhou, L., Du, J., and Shi, Z. (2015). The multi-step PEG and ACE constrained PEG algorithms can design the LDPC codes with better cycle-connectivity. In *2015 IEEE International Symposium on Information Theory (ISIT)*, pages 46–50.

[16] Hoppen, C. and Wormald, N. (2018). Local algorithms, regular graphs of large girth, and random regular graphs. *Combinatorica*, 38(3):619–664.

[17] Hu, X.-Y., Eleftheriou, E., and Arnold, D. . (2001). Progressive edge-growth tanner graphs. In *GLOBECOM'01. IEEE Global Telecommunications Conference*, volume 2, pages 995–1001 vol.2.

[18] Hu, X.-Y., Eleftheriou, E., and Arnold, D. M. (2005). Regular and irregular progressive edge-growth tanner graphs. *IEEE Transactions on Information Theory*, 51(1):386–398.

[19] Jiang, X., Hai, H., Wang, H., and Lee, M. H. (2017). Constructing large girth QC protograph LDPC codes based on PSD-PEG algorithm. *IEEE Access*, 5:13489–13500.

[20] Krushinsky, D. and Woensel, T. V. (2015). An approach to the asymmetric multi-depot capacitated arc routing problem. *European Journal of Operational Research*, 244(1):100 – 109.

[21] McGowan, J. A. and Williamson, R. C. (2003). Loop removal from LDPC codes. In *Information Theory Workshop, 2003. Proceedings. 2003 IEEE*, pages 230–233. IEEE.

[22] Orlin, J. B. and Ahuja, R. K. (1992). New scaling algorithms for the assignment and minimum mean cycle problems. *Mathematical Programming*, 54(1):41–56.

[23] Pramanik, A., Patil, G., and Borman, L. (2013). Small length quasi-cyclic LDPC code for wireless applications. In *2013 Annual International Conference on Emerging Research Areas and 2013 International Conference on Microelectronics, Communications and Renewable Energy*, pages 1–5.

[24] Richardson, T. (2003). Error floors of LDPC codes. In *Proceedings of the annual Allerton conference on communication control and computing*, volume 41, pages 1426–1435. The University; 1998.

[25] Roditty, L. and Tov, R. (2013). Approximating the girth. *ACM Transactions on Algorithms (TALG)*, 9(2):15.

[26] Shebl, S., Shokair, M., and Gomaa, A. (2014). Novel construction and optimization of LDPC codes for NC-OFDM cognitive radio systems. *Wireless personal communications*, 79(1):69–83.

[27] Sherali, H. D. and Smith, J. C. (2001). Improving discrete model representations via symmetry considerations. *Management Science*, 47(10):1396–1407.

[28] Tanner, R. (1981). A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547.

[29] Tanner, R. M., Sridhara, D., Sridharan, A., Fuja, T. E., and Costello, D. J. (2004). LDPC block and convolutional codes based on circulant matrices. *IEEE Transactions on Information Theory*, 50(12):2966–2984.

[30] Tasdighi, A., Banihashemi, A. H., and Sadeghi, M.-R. (2016). Symmetrical constructions for regular girth-8 QC-LDPC codes. *IEEE Transaction on Communications*, 14(8).

[31] IEEE Std 802.11$^{\text{TM}}$ (2007). Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. Technical report.

[32] Thomassen, C. (1983). Girth in graphs. *Journal of Combinatorial Theory, Series B*, 35(2):129–141.

[33] Zhang, J. and Fossorier, M. P. (2005). Shuffled iterative decoding. *IEEE Transactions on Communications*, 53(2):209–213.

Appendix

Proof of Proposition 1: We first prove that for a (J, K)-regular code of dimensions (m, n), $r_{cr} \leq c_{cr}$ where $r_{cr} = \lfloor (n-1)/(K-1) \rfloor$ and $c_{cr} = \lfloor (m-1)/(J-1) \rfloor$.

In practice, $J < K < n$ relationship is valid. Let $\frac{J}{K} = a \in (0, 1)$, then $mK = nJ \implies m = na$. We can write, $\frac{m-1}{J-1} = \frac{na-1}{Ka-1} = \frac{a(n-1)+a-1}{a(K-1)+a-1} > \frac{n-1}{K-1}$, since $a < 1$. From here we obtain $\lfloor \frac{n-1}{K-1} \rfloor \leq \lfloor \frac{m-1}{J-1} \rfloor \implies r_{cr} \leq c_{cr}$.

Let $\mathbf{H}$ be (J, K)-regular matrix of dimensions (m, n) with girth $g > \tau$. Let us apply the following reordering algorithm with time complexity $\mathcal{O}(c_{cr})$ on the $\mathbf{H}$.

At step 1 of *Reordering* in Figure 18, J ones are located in the first column. For the second row, i.e., $s = 2$, first available $(K-1)$ columns to locate ones are the columns $(K+1, ..., 2K-1)$, since otherwise a cycle with length of less than g exists. Similarly for the second column, i.e., $s = 2$, first available $(J-1)$ rows are the rows $(J+1, ..., 2J-1)$ without creating a cycle. The algorithm continues in this fashion for r_{cr} rows and columns. Since we have $r_{cr} \leq c_{cr}$, we continue to locate ones for the remaining $(c_{cr} - r_{cr})$ many columns. $\square$

Input: **H**, (m, n) dimensions, (J, K) values, g target girth

1. Pick row 1, reorder columns such that all ones are in first K columns.
   Pick column 1, reorder rows such that all ones are in first J rows.
2. **For** $s \in \{2, ..., r_{cr}\}$
3. Pick row s, reorder columns such that $(K-1)$ ones are in first available columns.
   Pick column s, reorder rows such that $(J-1)$ ones are in first available rows.
4. **End For**
5. **For** $s \in \{r_{cr}+1, ..., c_{cr}\}$
6. Pick column s, reorder rows such that $(J-1)$ ones are in first available rows.
7. **End For**

Output: Reordered **H** matrix

Figure 18: *Reordering* algorithm

Proof of Proposition 2: Assume we applied *VarFix* and consider cells whose x_{ij} values have been fixed to 1. There are four cases to have an alternating sequence among variable and check nodes as given in Figures 19 and 20.

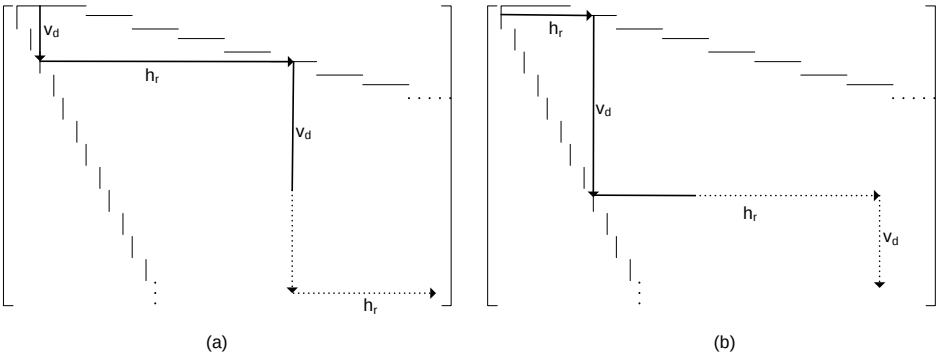

Figure 19: Alternating variable and check nodes, cases 1 and 2

In Figure 19a, the sequence of case 1 is $(v_d, h_r, v_d, h_r, ...)$ and in Figure 19b for case 2, we have the sequence $(h_r, v_d, h_r, v_d, ...)$. Both of the sequences do not include v_u and h_l movements. Hence, there cannot be any cycles in these cases.

In Figure 20a (case 3), we have two options to start, i.e., h_r or h_l movements. Then the sequence will be $(h_r$ or $h_l, v_d, h_r, v_d, h_r, ...)$, which does not include v_u movement. In Figure 20b (case 4), v_d or v_u are candidates to begin the sequence. In this case, the sequence will be $(v_d$ or $v_u, h_r, v_d, h_r, v_d, ...)$, which does not include h_l movement. Hence, there are no cycles in these cases either. □

Proof of Proposition 4: For any dimensions (m, n), we have $z^* \leq z_f^*$, since *VarFix* fixes some x_{ij} variables. If there exists a (J, K)-regular code, then there is an optimal solution with

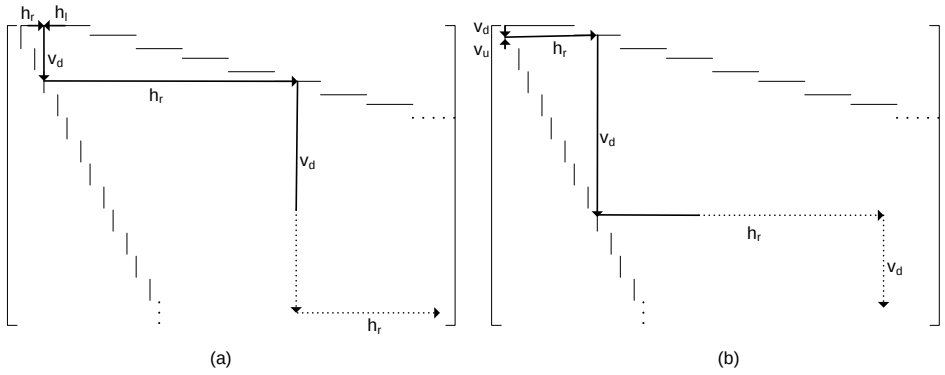

Figure 20: Alternating variable and check nodes, cases 3 and 4

objective value $z^* = 0$. We know from Proposition 1 when $g > \tau$, a (J, K)-regular BG can be expressed as in Figure 10. Hence, we have $z_f^* = z^* = 0$.

In MDD, if $\rho(i, j) \geq g$, then x_{ij} can be nonzero without harming the girth g. When $g \leq \tau$, there are $(i, j) \in Q$ in Figure 10 with $\rho(i, j) \geq g$ and they are fixed to zero, since *VarFix* fixes all entries in the region Q to zero. Then, we have $0 = z^* \leq z_f^*$ in this case. $\square$

Table 9: The impact of the valid cuts (1), (2), and (3) in Proposition 3 on BC_2

			Valid Cuts (1)						Valid Cuts (1), (2)						Valid Cuts (1), (2), (3)								
					CPU		# Cuts				CPU		# Cuts				CPU		# Cuts		# Cuts		
g	J	n	z_l	z	(secs)	Nodes	Int	Frac	z_l	z	(secs)	Nodes	Int	Frac	z_l	z	(secs)	Nodes	Int	Frac	(1)	(2)	(3)
6	3	20	3.25	6.5	*time*	4835210	246	0													81		
		30	0.25	2	*time*	2344931	2506	0													164		
		40	0	0	0.14	1	152	0													282		
		60	0	0	0.16	0	130	0													613		
		80	0	0	0.27	3	120	0													1074		
		200	0	0	0.63	0	130	0													6570		
		300	0	0	1.27	0	153	0													14725		
		500	0	0	5.27	0	209	0													40785		
		1000	0	0	43.05	0	139	0													162810		
	5	200	0	184.8	*time*	13952	338378	0													4029		
		300	0	0	3368.63	3335	305752	0													8577		
		500	0	0	51.89	21	4519	0													23003		
		1000	0	0	190.10	12	1727	0													89370		
8	3	20	15.5	15.5	0.05	0	0	0	15.5	15.5	0.11	0	0	0							171	0	
		30	21.5	21.5	1.72	676	268	0	21.5	21.5	0.14	0	0	0							378	3	
		40	13.5	21.5	*time*	1758125	2309	78	21.5	21.5	24.74	5892	441	0							573	15	
		60	3	16	*time*	170375	38275	0	7	14	*time*	136315	61461	0							963	72	
		80	0	16.5	*time*	61678	108441	0	1.5	16.5	*time*	44405	111784	0							1563	138	
		200	0	124	*time*	14507	349094	0	0	99.5	*time*	16276	298919	0							7693	910	
		300	0	183.5	*time*	11397	391908	0	0	175	*time*	10182	328829	0							16363	1832	
		500	0	0	1400.91	1112	89264	0	0	0	1316.96	1075	79070	0							43473	1792	
		1000	0	0	498.32	47	1345	0	0	0	257.44	27	1149	0							168133	1451	
	5	300	21	311.1	*time*	2078	349105	0	83.4	310.5	*time*	1797	307602	0							16865	783	
		500	0	520.5	*time*	758	25183	0	1.5	452.4	*time*	346	18107	0							36385	2277	
		1000	0	1050.3	*time*	108	21751	0	0	1050.3	*time*	31	10716	0							112779	9384	
10	3	20	15.5	15.5	0.06	0	0	0	15.5	15.5	0.08	0	0	0	15.5	15.5	0.13	0	0	0	171	0	0
		30	23	23	0.08	0	0	0	23	23	0.06	0	0	0	23	23	0.17	0	0	0	406	3	0
		40	30.5	30.5	0.05	0	0	0	30.5	30.5	0.06	0	0	0	30.5	30.5	0.16	0	0	0	741	15	0
		60	45.5	45.5	0.05	0	0	0	45.5	45.5	0.13	0	0	0	45.5	45.5	0.16	0	0	0	1711	72	0
		80	59	59	729.42	37892	3173	8	59	59	732.06	37892	3173	8	59	59	0.17	0	1	0	2983	129	3
		200	16.5	146.5	*time*	6135	332259	3	16.5	146.5	*time*	6418	338386	3	65	99.5	*time*	31037	174760	10	11403	149	175
		300	0	219	*time*	4783	453199	0	0	217.5	*time*	5186	482806	0	26	205.5	*time*	5471	539611	0	21383	99	428
		500	0	354.5	*time*	3625	644638	0	0	354.5	*time*	4404	751600	0	0	354.5	*time*	2102	277233	0	51493	99	1184
		1000	0	750.5	*time*	820	159362	0	0	750.5	*time*	964	187364	0	0	750.5	*time*	567	91086	0	183843	161	4639
	5	300	315.3	315.3	0.14	0	0	0	315.3	315.3	0.15	0	0	0	315.3	315.3	0.15	0	0	0	44551	783	0
		500	369.3	525.3	*time*	895	1581125	66	369.3	525.3	*time*	893	1581027	66	516.3	516.3	104.95	63	1813	0	115801	2037	30
		1000	294.3	1050.3	*time*	326	831424	2	294.3	1050.3	*time*	322	831058	2	861.3	1050.3	*time*	1640	86147	9	300051	3924	630