3 Code Design

The design of an IRA code involves several steps, which may be listed as: establishing an appropriate design rate for the intended application, finding an optimal degree distribution for this design rate, choosing an appropriate block length (again, keeping in mind the requirements of the application), and defining the graph construction method. The code symbol's alphabet may or may not be binary, but the use of a binary alphabet brings significant advantages during decoding as shown in Section 2.2.3.

Once a degree distribution is chosen, a functioning IRA code can be designed for any reasonably large block length. Although theoretical bounds on the performance of a code can be established through density evolution assuming a random code with infinite block length, an actual code is completely defined only after the Tanner graph is constructed. A Tanner graph for a blocklength-k message and length-n codeword can be described by listing the information-nodes joined with each check-node. Since the graph is right-regular, this list can be conveniently represented as an m-by-a_r matrix where each row lists the indices to the a_r information nodes joined with its corresponding check-node. Some steps, however, should be taken before populating this matrix.

3.1 Graph Construction

Good theoretical degree distributions for codes with infinite block-lengths can be found by linear programming, but the degree distribution of the information nodes in a code with finite block-length should be a relative frequency with fractional values relative to the total number of nodes (i.e. a normalized histogram) rather than a general discrete probability mass vector with real values. Thus, once a valid degree distribution is obtained through optimization techniques such as the one seen in [RSU2001, Sec. IV], it should be modified to ensure that

$$\begin{cases} \Lambda_i k \in \mathbb{N},\ \forall\ i \in \mathbb{N}; & \text{(a)} \\ k a_l = m a_r = E, & \text{(b)} \end{cases} \tag{3-1}$$

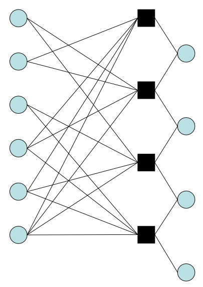

Figure 3.1: Graphical representation of an IRA graph: check-nodes are portrayed as black squares, variable-nodes as circles.

where a_l is the average left-degree, a_r is the average right-degree and E is the total number of edges in the graph[1]. Condition (3-1.a) states that the number of degree-i nodes must be an integer; condition (3-1.b), that the number of edges leaving the information nodes equals the number of edges arriving at the check-nodes.

After obtaining a distribution that satisfy the conditions in (3-1) the edges can be randomly traced. The simplest way to do this would be to list the indices of all information nodes and assign degrees to each node according to the degree distribution Λ, then repeat the elements in the list so each of the $k\Lambda_i$ nodes of degree i are listed i times. This list now has $\sum_{i=1}^{d_{max}} i \cdot k\Lambda_i = ma_l$ elements which we will call *sockets*.

Definition 7 (socket) *A socket is a pointer to a node. Each edge is numerically represented by the association of two sockets. By definition, a degree i variable-node has i incident edges, thus i sockets. A socket that has not been associated with any other to form an edge is considered a free socket.* $\diamond$

The sockets go through a random permutation and the list is formatted as an m-by-a matrix, which is only a compact representation of a sparse right-regular graph. Each of the m rows stands for a check-node, and the a row elements (or sockets) are the indices to the left-ends of the edges arriving at the check-node. The edges joining check-nodes and parity variable-nodes need not be explicitly defined, since they follow the staircase pattern shown previously in Figure 2.1. If the rows of $\mathbf{H}$ are properly arranged, the $\mathbf{H}^{(2)}$ part is taken to be an m-by-m staircase matrix.

[1]See app. A for proof and details on the notation and expressions used in this section.

For example, the graph in Figure 3.1 may be specified by the matrix

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

or, using sparse matrix notation, as in Figure 3.2, where element $S_{1,1} = 2$ of matrix $\mathbf{S}$ indicates that node check-node 1 is joined with information-node 2.

$$\mathbf{S} = \begin{bmatrix} 2 & 4 & 5 & 6 \\ 1 & 2 & 4 & 6 \\ 1 & 3 & 5 & 6 \\ 3 & 4 & 5 & 6 \end{bmatrix}$$

Figure 3.2: The right-regular graph from Figure 3.1 represented by an m-by-a matrix, using sparse matrix notation. Each row lists the connections from one check-node, i.e. the positions of the 1's in each row of the matrix $\mathbf{H}^{(1)}$. The connections involving the parity variable-nodes to the right of the check-nodes are implicitly given by the row order.

A random permutation of edges produces a random code, but a code which has many unacceptable weaknesses. A simple random permutation does not prevent one check-node from joining twice to the same information node, which cancels out whatever the influence this information-node would have on the check-node. Other problems that are overlooked by this procedure are: the presence of short cycles in the graph; the graph's minimum distance and rank deficiency. These problems are all intimately related to the graph construction method [TJV2003], therefore a few restrictions should be imposed when tracing the edges.

3.1.1 Progressive Edge Growth

Progressive Edge-Growth (PEG) is a method for the construction of Tanner Graphs proposed in [HEA2005], which evolves by extending the graph's shortest cycle such that it is made as long as possible. The length of the shortest cycle in a graph is termed the graph's girth, while the length of the shortest cycle among those that include one particular node is named its local girth. The PEG algorithm proceeds along the variable-nodes, in order of increasing degree, in such a manner that each new edge has an impact as small as possible on the graph's girth.

Some definitions should be made before formally explaining how the PEG algorithm works.

Definition 8 (Path) *A* path *is an ordered list of nodes (variable and checks) such that every two consecutive nodes are joined by an edge. The* distance *between two nodes in a graph is the number of edges that separate them in the shortest path. All the paths branching from the ending node on another path are extensions of this path.* ◇

Therefore, we say two nodes are *connected* when they belong in the same path. Alternatively, we can say that a reaches b when there is at least one path connecting a to b.

Definition 9 (Cycle) *In a graph, a* cycle *is a path that begins and ends in the same node. The cycle's* length *is its total number of edges.* ◇

Definition 10 (Tree) *In the context of Tanner Graphs, a* tree *is a representation of a graph with a hierarchical structure.* ◇

The nodes in a tree can be grouped according to their depths. One node in the graph is chosen as the *root*. The tree starts on the root node, the root and its neighbors are said to be in depth 0. All nodes in the tree, except the root, have a *parent* node. Nodes without *children* are called *leafs*.

The root's children are followed by their remaining neighbors, which are collectively called the *Tier 1*. The nodes in Tier 1 and their children are in *Depth 1*. The hierarchy in a tree can be observed in Figure 3.3.

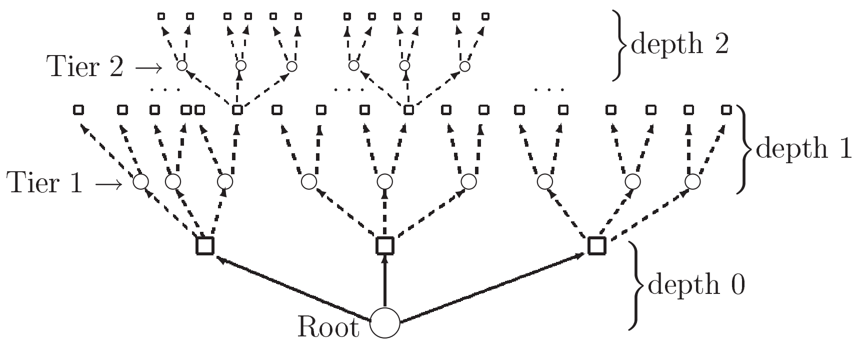

Figure 3.3: An example of a tree drawn to depth 2. The first variable-nodes in the tree following the root make the *Tier 1*. The variable-nodes in Tier 1 and the check-nodes immediately below them (where *below* means further from the root) form *Depth*1, and so on.

We can observe all the paths that include one particular node by expanding it as the root of a tree. Evidently, every element present in a tree reaches all others and a cycle is completed whenever the same node appears more than once in the same tree.

We denote by $\mathcal{M}^l_{c_j}$ the set of check-nodes in the tree stemming from information node c_j up to depth l. We denote the complement of this set by $\overline{\mathcal{M}}^l_{c_j}$. The set grows with increasing l until one of the following cases occur:

1. $|\mathcal{M}^l_{c_j}| = |\mathcal{M}^{l+1}_{c_j}|$, meaning all existing paths involving node c_j form cycles;
2. $|\mathcal{M}^{l+1}_{c_j}| = m$, i.e. all check-nodes in the graph are reached by c_j in the next depth.

When either case happens, a check-node in $\overline{\mathcal{M}}^l_{c_j}$ is randomly chosen. In case 1 this new edge extends the tree without creating new cycles. In case 2 a new cycle is formed, but the node's local girth remains the same. If, additionally, the new edge is always attached to the check-node with the lowest degree — the check-node's degree is, in this case, the number of incoming edges that have been drawn up to this point. A summarized description of the PEG algorithm can be found in Algorithm 1.

Algorithm 1 Progressive Edge-Growth Algorithm for IRA Codes

```
for $j = 0$ to $k - 1$ do
  for $\kappa = 0$ to $d_{u_j} - 1$ do
    if $\kappa = 0$ then
      $E^0_{u_j} \longleftarrow$ edge$(z_i, u_j)$, where $E^0_{u_j}$ is the first edge incident to $u_j$ and $z_i$
      is a check-node such that it has the lowest check-node degree under
      the current graph setting $\bigcup_{\imath=0}^{j-1} E_{u_\imath}$.
    else
      expand a subgraph from information node $u_j$ up to depth $l$ under
      the current graph setting such that $\overline{\mathcal{M}}^l_{u_j} \neq \emptyset$ but $\overline{\mathcal{M}}^{l+1}_{u_j} = \emptyset$, then
      $E^\kappa_{u_j} \longleftarrow$ edge$(z_i, u_j)$, where $E^\kappa_{u_j}$ is the $\kappa^{th}$ edge incident to $u_j$ and $z_i$
      is a check-node randomly picked from $\overline{\mathcal{M}}^l_{u_j}$ having the lowest check-
      node degree.
    end if
  end for
end for
```

It is noteworthy to mention that special care should be taken in the case we want to construct a graph for IRA codes, where not all edges are placed through this method. The edges that join parity and check-nodes are drawn in a pre-determined manner before all others (see figure 2.2), meaning:

$$\forall z_k \in \mathcal{M}^l_{c_j} \begin{cases} \text{if } k = 1, & z_{k+1} \in \mathcal{M}^{l+1}_{c_j}; \\ \text{if } k \in [2, m-1], & \{z_{k+1}, z_{k-1}\} \subset \mathcal{M}^{l+1}_{c_j}; \\ \text{if } k = m, & z_{k-1} \in \mathcal{M}^{l+1}_{c_j}. \end{cases}$$

Therefore, any two edges that are incident on the same information node will form a cycle. What we attempt to do in this case is to maximize the graph's girth.

In a classic LDPC graph, the PEG algorithm would verify at each iteration whether the cardinality of $\mathcal{M}_{c_j}^{l}$ has stopped growing (in other words, if $(|\mathcal{M}_{c_j}^{l}| = |\mathcal{M}_{c_j}^{l+1}|)$). In an IRA code, however, the pre-arranged edges from the parity nodes ensure that all check-nodes are connected, which means every IRA graph is a *connected graph* and these verifications are not necessary. In [SJR2005], we are reminded that in a degree distribution where $k\Lambda_2 \geq m-1$, the search for an optimal graph would lead to an IRA graph.

3.1.2 Level-1 version

A simpler approach to graph construction which avoids length-4 cycles but does not attempt to maximize the girth is described below in algorithm 2.

Algorithm 2 random method for IRA Codes (`PEG-L1`)

```
for $j = 1$ to $k$ do
  for $\kappa = 0$ to $d_{u_j} - 1$ do
    if $\kappa = 0$ then
      $E_{u_j}^{0} \longleftarrow$ edge$(z_i, u_j)$, where $E_{u_j}^{0}$ is the first edge incident on $u_j$ and
      $z_i$ is a randomly chosen check-node with a free socket.
    else
      Expand a tree $u_j$ up to depth 1 under the current graph setting. Then
      $E_{u_j}^{\kappa} \longleftarrow$ edge$(z_i, u_j)$, where $E_{u_j}^{\kappa}$ is the $\kappa^{th}$ edge incident to $u_j$ and $z_i$
      is a check-node randomly picked from $\overline{\mathcal{M}}_{u_j}^{1}$.
    end if
  end for
end for
```

This method can eventually produce a random permutation that leads to good codes. However, when the degree distribution allows for nodes of very high degree — when compared to the total block length — randomly traced edges tend to exhaust the nodes that will not lead to length-4 cycles. In other words, $\mathcal{M}_{u_j}^{1}$ after tracing the edges for a large portion of the check-nodes. This means that the last check-nodes will send less accurate messages throughout decoding. In some cases (typically shorter block lengths and high maximum degrees) even length-2 cycles become inevitable if the edges are drawn randomly and the only solution is to truncate the code, leaving out the check-nodes that can't be traced to any information node without creating a cycle.

This truncation's effects on the code rate are not impressive since only a few among hundreds of check-nodes are truncated, but it does bring noticeable changes to the actual degree distribution. Since degree distributions are carefully chosen in order to maximize performance, this method is sub-optimal and may even lead to very bad codes.

3.1.3 Non-greedy version

The Non-Greedy version of the PEG algorithm expands the tree up to a previously determined depth-level. It does not attempt to maximize the length of the cycles that are completed by each additional edge in the graph, it instead ensures all cycles will have length superior to $2(l_{max}+1)$. The modified algorithm is detailed in Algorithm 3. As in the standard PEG, the non-greedy version chooses check-nodes with the lowest degree among the eligible candidates, resulting in a graph that grows more regularly on the left-hand side.

Algorithm 3 Non-Greedy Progressive Edge-Growth Algorithm for IRA Codes

```
for $j = 0$ to $k - 1$ do
  for $\kappa = 0$ to $d_{u_j} - 1$ do
    if $\kappa = 0$ then
      $E^0_{u_j} \longleftarrow$ edge$(z_i, u_j)$, where $E^0_{u_j}$ is the first edge incident to $u_j$ and $z_i$
      is a check-node such that it has the lowest check-node degree under
      the current graph setting $\bigcup_{\imath=0}^{j-1} E_{u_\imath}$.
    else
      Expand a subgraph from information node $u_j$ up to a previously
      determined depth $l_{max}$. Then $E^{\kappa}_{u_j} \longleftarrow$ edge$(z_i, u_j)$, where $E^{\kappa}_{u_j}$ is the
      $\kappa^{th}$ edge incident to $u_j$ and $z_i$ is a check-node randomly picked from
      $\overline{\mathcal{M}}^{l_{max}}_{u_j}$ having the lowest check-node degree.
    end if
  end for
end for
```

3.1.4 Look-Ahead enhanced version

The Standard PEG Algorithm maximizes the local girth, but does not anticipate the effects that the new edge will have on the graph because it ignores the connections between each of the candidates in $\overline{\mathcal{M}}^{l}_{u_j}$ and other nodes in the graph.

The Look-Ahead enhanced version of the PEG algorithm includes an additional loop through the eligible check-nodes in $\overline{\mathcal{M}}^{l_{max}}_{u_j}$ before deciding which should receive a new edge. The decision criterium, explained in Algorithm 4, maximizes l_{max}. In other words, it chooses the check-node that, if included in the neighborhood of the current root node (u_j), would lead to the *deepest* tree.

Algorithm 4 Look-Ahead enhanced Progressive Edge-Growth Algorithm for IRA Codes

```
1: for $j = 0$ to $k - 1$ do
2:   for $\kappa = 0$ to $d_{u_j} - 1$ do
3:     if $\kappa = 0$ then
4:       $E^0_{u_j} \longleftarrow \mathbf{edge}(z_i, u_j)$, where $E^0_{u_j}$ is the first edge incident to $u_j$ and
         $z_i$ is a check-node such that it has the lowest check-node degree
         under the current graph setting $\bigcup_{i=0}^{j-1} E_{u_i}$.
5:     else
6:       Expand a tree-like subgraph from information node $u_j$ up to depth $l$
         under the current graph setting such that $\overline{\mathcal{M}}^{l}_{u_j} \neq \emptyset$ but $\overline{\mathcal{M}}^{l+1}_{u_j} = \emptyset$.
7:       A subset $\mathcal{Z}^{(j)} \subset \overline{\mathcal{M}}^{l}_{u_j}$ is composed exclusively of the check-nodes in
         $\overline{\mathcal{M}}^{l}_{u_j}$ with the lowest degree under the current graph setting.
8:       for $t = 1$ to $|\mathcal{Z}^{(j)}|$ do
9:         Define $\mathbf{E}^{(t)} = \mathbf{E}$, a hypothetical graph identical to the current
           graph.
10:        $E^{(t,\kappa)}_{u_j} \longleftarrow \mathbf{edge}(z^{(j)}_t, u_j)$, where $E^{(t,\kappa)}_{u_j}$ is the $\kappa^{th}$ edge incident to
           $u_j$ in the hypothetical graph and $z_t$ is $t^{th}$ check-node in $\mathcal{Z}^{(j)}$.
11:        do step 6 under $\mathbf{E}^{(t)}$ and make $\ell(t) = l$, where $\boldsymbol{\ell}$ is a vector
           containing the maximum depth level of the hypothetical graphs.
12:      end for
13:      $E^{\kappa}_{u_j} \longleftarrow \mathbf{edge}(z^{(j)}_{t_{max}}, u_j)$, where $\ell(t_{max})$ is the maximum value in $\boldsymbol{\ell}$.
14:    end if
15:  end for
16: end for
```

3.1.5 Look-Ahead Reverse version

As an alternative to the Look-Ahead enhanced algorithm (PEG-LA), we present the Look-Ahead Reverse algorithm (PEG-LAR). The sole difference between the former and the latter is that the reverse version of the Look-Ahead algorithm tries to minimize the depth of the tree during graph construction. By minimizing the tree's depth we do not maximize the graph's girth but we attempt to build a graph where every node reaches all others in fewer steps (or iterations of the BP decoder) than they do in the graphs constructed through the Look-Ahead enhanced version of the PEG algorithm.

The motivation behind this variation of the PEG-LA algorithm comes from the conclusion that the PEG-LA variation does not effectively maximize the graph's girth. The PEG-LA algorithm does maximize the length of every new cycle that is closed by a new edge in the graph, which does not guarantee a large girth.

Algorithm 5 Look-Ahead Reverse Progressive Edge-Growth Algorithm for IRA Codes

```
1: for $j = 0$ to $k - 1$ do
2:   for $\kappa = 0$ to $d_{u_j} - 1$ do
3:     if $\kappa = 0$ then
4:       $E^0_{u_j} \longleftarrow \textbf{edge}(z_i, u_j)$, where $E^0_{u_j}$ is the first edge incident to $u_j$ and
         $z_i$ is a check-node such that it has the lowest check-node degree
         under the current graph setting $\bigcup_{\imath=0}^{j-1} E_{u_\imath}$.
5:     else
6:       Expand a tree-like subgraph from information node $u_j$ up to depth $l$
         under the current graph setting such that $\overline{\mathcal{M}}^l_{u_j} \neq \emptyset$ but $\overline{\mathcal{M}}^{l+1}_{u_j} = \emptyset$.
7:       A subset $\mathcal{Z}^{(j)} \subset \overline{\mathcal{M}}^l_{u_j}$ is composed exclusively of the check-nodes in
         $\overline{\mathcal{M}}^l_{u_j}$ with the lowest degree under the current graph setting.
8:       for $t = 1$ to $|\mathcal{Z}^{(j)}|$ do
9:         Define $\mathbf{E}^{(t)} = \mathbf{E}$, a hypothetical graph identical to the current
           graph.
10:        $E^{(t,\kappa)}_{u_j} \longleftarrow \textbf{edge}(z^{(j)}_t, u_j)$, where $E^{(t,\kappa)}_{u_j}$ is the $\kappa^{th}$ edge incident to
           $u_j$ in the hypothetical graph and $z_t$ is $t^{th}$ check-node in $\mathcal{Z}^{(j)}$.
11:        do step 6 under $\mathbf{E}^{(t)}$ and make $\ell(t) = l$, where $\boldsymbol{\ell}$ is a vector
           containing the maximum depth level of the hypothetical graphs.
12:      end for
13:      $E^{\kappa}_{u_j} \longleftarrow \textbf{edge}(z^{(j)}_{t_{min}}, u_j)$, where $\ell(t_{min})$ is the minimum value in $\boldsymbol{\ell}$.
14:    end if
15:  end for
16: end for
```