CONSTRUCTOR UNIVERSITY

Implementation in Python of Progressive Edge Growth algorithm to generate and visualize Tanner Graphs and Multi-Edge Type LDPC with Parity Check Matrix

by

Jona Bako

Bachelor Thesis in Computer Science

Submission: March 4, 2024 Supervisor: Thesis Supervisor

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Bako, Jona
Matriculation number	30004124
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

.....

Date, Signature

Abstract

Thesis Abstract (summarize purpose and content of thesis)

Contents

1 Introduction

Low-density parity-check (LDPC) codes represent a significant advancement in the realm of error-correcting codes, characterized by their sparse parity-check matrices and impressive decoding capabilities. Defined by an $m \times n$ matrix, where $n > M$ and $M = N - K$ (N: Total number of bits in a codeword; K: Number of information bits; M: Number of parity bits or check nodes.), LDPC codes are predominantly studied in their binary form, offering a compelling balance between performance and complexity in modern communication systems.

The crux of LDPC code construction lies in the creation of a sparse parity-check matrix, where the number of '1' entries is notably smaller than '0' entries. This sparsity is a defining feature, contributing to efficient decoding processes. The row-weight, denoted as k, signifies the number of '1's in a row, while the column-weight, denoted as j, indicates the number of '1's in a column. When both row and column weights remain constant, the LDPC code is considered regular; otherwise, it is irregular.

Constructing LDPC codes involves a meticulous process of determining the connections between rows and columns of the parity-check matrix or between check and symbol nodes within the corresponding Tanner graph. This process, crucial for achieving desired LDPC code parameters such as rate, girth, and length, aims at optimizing both decoding performance and hardware implementation. The fundamental goal is to design LDPC codes that offer robust error correction capabilities while ensuring ease of hardware integration.

In the pursuit of optimal LDPC code construction, the challenge arises in balancing the trade-offs between decoding performance and hardware complexity. Regular codes, with their structured row-column connections, often present easier hardware implementation; however, they may lack the flexibility needed for varied code designs. On the other hand, irregular codes, while offering greater design flexibility, can result in increased hardware complexity due to their unstructured interconnections.

Various construction methods exist, ranging from random (unstructured connections) to structured (predefined connections), each with its own set of advantages and limitations. Random constructions, despite their flexibility, introduce complexities in decoder interconnections, while structured constructions may limit the range of achievable rates, lengths, and girths. Thus, the quest continues for methods capable of generating a diverse range of LDPC codes that balance performance metrics with implementation constraints.

One prominent algorithm in LDPC code construction is the Progressive Edge Growth (PEG) algorithm. This non-algebraic method offers a simple yet effective approach to constructing LDPC codes of varying lengths and rates. The PEG algorithm, as described by Gabofetswe Alafang Malema in his Ph.D. thesis paper: "Low-Density Parity-Check Codes: Construction and Implementation", builds a Tanner graph by iteratively adding edges, ensuring minimal impact on the graph's girth. Notably, codes generated through the PEG algorithm have demonstrated superior performance, particularly at shorter code lengths.

Further advancements in the PEG algorithm have been proposed, aiming to enhance the performance of the obtained codes. Hua Xiao and Amir H. Banihashemi introduced improvements to the standard PEG algorithm, focusing on selecting check nodes with higher connectivity to improve the resulting LDPC codes' performance. Additionally, Lin

Chen and Da-Zheng Feng presented a fast implementation of the PEG algorithm, utilizing a red-black (RB) tree array to manage the check nodes efficiently.

The improved PEG algorithm by Hua Xiao and Amir H. Banihashemi introduces modifications to the original algorithm, particularly in selecting check nodes based on their connectivity to improve the resulting LDPC codes' performance. This modification, effective for constructing irregular codes, aims to enhance the connectivity of the Tanner graph, thereby improving the minimum distance and reducing trapping sets (short cycles within the Tanner graph), crucial for reducing errors in the error-floor region.

On the other hand, the fast implementation of the PEG algorithm by Lin Chen and Da-Zheng Feng offers a novel approach to constructing LDPC codes efficiently. By utilizing an RB-tree array to manage check nodes, this implementation significantly reduces computational complexity while maintaining the quality of generated codes. Through dynamic adjustments of the RB tree structure, the algorithm achieves efficient construction of LDPC codes with large girths, essential for improving error correction capabilities.

In this thesis, we aim to thoroughly explore and implement these advancements in the PEG algorithm using Python. The focus will be on the practical aspects of constructing the PEG algorithm developed by Hua Xiao and Amir H. Banihashemi. This includes creating Tanner graphs and multi-edge LDPC codes step by step, from the initial stages to the final output. Additionally, visualization methods will be developed to display these generated graphs based on user input. Through this detailed approach, a comprehensive understanding of how the algorithm works, its performance characteristics and its real-world applications in LDPC codes will be achieved.

2 Statement and Motivation of Research

The motivation behind this project stems from the ever-evolving landscape of communication technologies, where efficient error-correction mechanisms are crucial for reliable data transmission. LDPC codes, with their sparse parity-check matrices, offer an attractive balance of error correction capability and decoding complexity. However, the challenge lies in constructing these codes with optimal parameters to meet specific performance requirements...

[This part should make clear which question, exactly, you are pursuing, and why your project is relevant/interesting. This is the place to explain the background and to review the existing literature. Where does your project extend the state of the art? What weaknesses in known approaches do you hope to overcome? If you have carried out preliminary experiments, describe them here.]

(target size: 5-10 pages)

3 Description of the Investigation

This section serves as the technical core of the thesis, detailing the meticulous steps taken to implement the Progressive Edge Growth (PEG) algorithm in Python for generating Tanner graphs and Parity Check Matrices essential for Low-Density Parity-Check (LDPC) codes.

3.1 Step 1: User Input

The implementation begins by engaging the user for vital input parameters:

- **Symbol Nodes:** The total number of symbol nodes that will constitute the Tanner graph.
- **Check Nodes:** The number of check nodes to be included in the Tanner graph.
- **S-node Degrees:** The degrees assigned to each symbol node, indicating the number of edges it will form with check nodes. These degrees are provided by the user in a comma-separated format.

3.2 Step 2: Initialization

To construct the LDPC Parity Check Matrix, we initialize an empty matrix with dimensions $(check\_nodes \times symbol\_nodes)$. This matrix will be populated gradually as we apply the PEG algorithm.

```
1    def create_ldpc_matrix(symbol_nodes, check_nodes, s_node_degrees):
2    parity_check_matrix = [[0] * symbol_nodes for _ in range(check_nodes)]
```

3.3 Step 3: Iterative Edge Addition

The heart of the PEG algorithm lies in its iterative approach to adding edges between symbol nodes and check nodes, carefully ensuring the desired connectivity while maintaining a sparse and efficient matrix structure.

Iterating Over Symbol Nodes

For each symbol node, the algorithm follows these steps:

1. **Tracking Covered Check Nodes:** We maintain a list of check nodes already connected to the current symbol node.
2. **Connecting to Check Nodes:**
    - If no check nodes are covered yet, we connect to the check node with the lowest degree in the entire matrix.
    - If there are covered and uncovered check nodes:
        - Connect to the one with the lowest degree among the uncovered nodes.
        - If all check nodes are covered, connect to the one with the lowest degree among the farthest nodes.

3. **Updating Parity Check Matrix:** Connect the symbol node to the chosen check node by updating the corresponding entry in the matrix.

```
def create_ldpc_matrix(symbol_nodes, check_nodes, s_node_degrees):
for symbol_node in range(symbol_nodes):
    covered_check_nodes = [] # List to keep track of covered check
        ↪ nodes
    for edge_number in range(1, s_node_degrees[symbol_node] + 1):
        if len(covered_check_nodes) == 0:
            # If no check nodes are covered, connect to the one with
                ↪ the lowest degree
            check_node = find_check_node_with_lowest_degree(
                ↪ parity_check_matrix)
            parity_check_matrix[check_node][symbol_node] = 1
            covered_check_nodes.append(check_node)
        else:
            if len(covered_check_nodes) < check_nodes:
                # If some check nodes are still uncovered
                uncovered_check_nodes = list(set(range(check_nodes)) -
                    ↪ set(covered_check_nodes))
                check_node = find_check_node_with_lowest_degree(
                    ↪ parity_check_matrix, uncovered_check_nodes)
            else:
                # If all check nodes are covered, connect to the
                    ↪ farthest one
                check_node = find_check_node_with_lowest_degree(
                    ↪ parity_check_matrix, covered_check_nodes)
            parity_check_matrix[check_node][symbol_node] = 1
            covered_check_nodes.append(check_node)
```

3.4 Step 4: Finding Check Nodes with Lowest Degree

To efficiently select check nodes with the lowest degree, we implement a helper function `find_check_node_with_lowest_degree()`. This function evaluates the degrees of check nodes and returns the one with the lowest degree among either all nodes or a specified subset.

```
def find_check_node_with_lowest_degree(matrix, nodes=None):
if nodes:
    # If specific nodes are provided, find the one with the lowest
        ↪ degree among them
    min_degree = float('inf')
    min_degree_node = None
    for node in nodes:
        degree = sum(row[node] for row in matrix)
        if degree < min_degree:
            min_degree = degree
            min_degree_node = node
    return min_degree_node
else:
```

```
        # If no specific nodes are provided, find the one with the lowest
            ↪ degree in the entire matrix
        min_degree = min(sum(row) for row in matrix)
        for node, row in enumerate(matrix):
            if sum(row) == min_degree:
                return node
```

3.5 Step 5: Printing the Parity Check Matrix

Throughout the iterative process, the algorithm prints the current state of the parity check matrix after each edge addition. This provides a visual representation of how the matrix evolves and the connectivity between symbol and check nodes.

```
    def create_ldpc_matrix(symbol_nodes, check_nodes, s_node_degrees):
    for symbol_node in range(symbol_nodes):
        for edge_number in range(1, s_node_degrees[symbol_node] + 1):
            print(f"Parity␣Check␣Matrix␣after␣processing␣edge␣{edge_number
                ↪ }:")
            print_matrix(parity_check_matrix)
```

3.6 Step 6: Final Output

The culmination of the algorithm is the generation of the LDPC Parity Check Matrix, fully populated with optimized connections between symbol nodes and check nodes.

```
    def main():
    symbol_nodes = int(input("Enter␣the␣number␣of␣symbol␣nodes:␣"))
    check_nodes = int(input("Enter␣the␣number␣of␣check␣nodes:␣"))
    s_node_degrees = list(map(int, input("Enter␣the␣S-node␣degrees␣(comma␣
        ↪ separated):␣").split(',')))

    ldpc_matrix = create_ldpc_matrix(symbol_nodes, check_nodes,
        ↪ s_node_degrees)

    print("\nFinal␣LDPC␣Parity␣Check␣Matrix:")
    print_matrix(ldpc_matrix)
```

3.7 Creating the Graph

The first step involves creating an undirected graph using NetworkX, representing the Tanner graph for LDPC code construction. This graph consists of two types of nodes:

- **Symbol Nodes ('s' type):** Representing the variable nodes in the Tanner graph.
- **Check Nodes ('c' type):** Representing the check nodes in the Tanner graph.

The user provides the number of symbol nodes n and check nodes m, along with the degrees of each symbol node as input.

```
    import networkx as nx

```

```
    def create_graph(n, m, s_node_degrees):
    G = nx.Graph()
    s_nodes = ['s{}'.format(i) for i in range(n)]
    G.add_nodes_from(s_nodes, node_type='s')
    c_nodes = ['c{}'.format(i) for i in range(m)]
    G.add_nodes_from(c_nodes, node_type='c')

    for s_node, degree in zip(s_nodes, s_node_degrees):
        for _ in range(degree):
            c_index = int(input(f"Enter index of a c-node connected to {
                ↪ s_node}: "))
            if c_index >= m:
                print(f"Error: Invalid c-node index {c_index}.")
                return
            c_node = 'c{}'.format(c_index)
            G.add_edge(s_node, c_node)

    return G

    # Get user input for n, m, and s-node degrees
    n = int(input("Number of symbol nodes (n): "))
    m = int(input("Number of check nodes (m): "))
    s_node_degrees = list(map(int, input("S-node degrees (comma separated)
        ↪ : ").split(',')))

    # Create the graph
    graph = create_graph(n, m, s_node_degrees)
```

3.8 Visualizing the Graph

After creating the graph, we visualize it using Matplotlib. The nodes are positioned in two rows, with 's' nodes in the top row and 'c' nodes in the bottom row. The nodes are colored differently for visual clarity, with 's' nodes in aqua and 'c' nodes in yellow.

```
    import matplotlib.pyplot as plt

    # Divide nodes into 's' and 'c' nodes
    s_nodes = [node for node in graph.nodes if graph.nodes[node]['
        ↪ node_type'] == 's']
    c_nodes = [node for node in graph.nodes if graph.nodes[node]['
        ↪ node_type'] == 'c']

    # Position nodes in two rows
    pos = {}
    for i, node in enumerate(s_nodes):
    pos[node] = (i, 1) # Top row

    for i, node in enumerate(c_nodes):
    pos[node] = (i, 0) # Bottom row

```

```
    # Draw the original graph with nodes in two rows
    plt.figure(figsize=(8, 4))
    nx.draw(graph, pos, with_labels=True, node_color=['aqua' if node in
        ↪ s_nodes else 'yellow' for node in graph.nodes],
        node_size=800, font_size=12, font_weight='bold')
    plt.title('Original Graph')
    plt.axis('off')
    plt.show()
```

3.9 Visualizing a Subgraph

Next, we allow the user to select a specific 's' node to visualize its subgraph. The user provides the selected 's' node as input, and the depth of the subgraph is fixed to 2 for this example.

```
    selected_node = input(f"Select an 's' node to visualize its subgraph (
        ↪ e.g., s0, s1, ...): ")

    # Check if the selected node is valid
    if selected_node not in s_nodes:
    print("Error: Selected node is not a valid 's' node.")
    else:
    depth = 2 # Depth of the subgraph

    # Create the subgraph
    subgraph = nx.ego_graph(graph, selected_node, radius=depth)

    # Divide nodes into 's' and 'c' nodes
    subgraph_s_nodes = [node for node in subgraph.nodes if subgraph.nodes[
        ↪ node]['node_type'] == 's']
    subgraph_c_nodes = [node for node in subgraph.nodes if subgraph.nodes[
        ↪ node]['node_type'] == 'c']

    # Draw the subgraph with nodes in two colors
    plt.figure(figsize=(6, 6))
    nx.draw(subgraph, with_labels=True, node_color=['aqua' if node in
        ↪ subgraph_s_nodes else 'yellow' for node in subgraph.nodes],
            node_size=800, font_size=12, font_weight='bold')
    plt.title(f'Subgraph of {selected_node} at depth {depth}')
    plt.axis('off')
    plt.show()
```

3.10 Difficulties Encountered and Thought Process

During the implementation of the PEG algorithm, several challenges were encountered, particularly in the selection of check nodes to connect with symbol nodes. The algorithm needed to strike a balance between covering all check nodes efficiently while maintaining the desirable properties of LDPC codes, such as a high girth and low error-floor characteristics.

- **Handling Uncovered Check Nodes:**
  - The algorithm had to consider cases where not all check nodes were covered by the subgraph expanded from the current symbol node.
  - To address this, we calculated the set of uncovered check nodes and selected the one with the lowest degree among them.
- **Dealing with Covered Check Nodes:**
  - When all check nodes were already covered, the algorithm needed to select the one with the lowest degree among the farthest nodes.
- **Thoughtful Check Node Selection:**
  - The `find_check_node_with_lowest_degree()` function played a crucial role in efficiently selecting check nodes.
  - We carefully considered whether to find the node with the lowest degree among all nodes or among a specific subset, depending on the situation.

3.11 Integration and Future Development

The processes of graph creation, testing, and visualization presented above form integral components of the Progressive Edge Growth (PEG) algorithm for LDPC code construction. These steps showcase the algorithm's functionality in creating vertices and edges, as well as updating the graph with each iteration.

The integration of these components demonstrates the full workflow of the algorithm, from the initial construction of the Tanner graph to the visualization of intermediate steps. The created graph provides a visual representation of the evolving connectivity between symbol nodes and check nodes, crucial for understanding the algorithm's progress.

Moving forward, the next phase of development will involve the implementation of a proper frontend. This frontend will enhance user interaction and provide a more intuitive concept and visualization of the Progressive Edge Growth algorithm. This simulation tool will be integrated to facilitate the testing and analysis of LDPC code generation using various parameter values.

Through this integration and future development, the Progressive Edge Growth algorithm will offer a comprehensive solution for constructing LDPC codes, combining efficient graph creation with intuitive visualization and simulation capabilities.

[This is the technical core of the thesis. Here you lay out your how you answered your research question, you specify your design of experiments or simulations, point out difficulties that you encountered, etc.]

(target size: 5-10 pages)

4 Evaluation of the Investigation

[This section discusses criteria that are used to evaluate the research results. Make sure your results can be used to published research results, i.e., to the already known state-of-the-art.]

(target size: 5-10 pages)

5 Conclusions

In this project, we have explored the implementation of the Progressive Edge Growth (PEG) algorithm for constructing Low-Density Parity-Check (LDPC) codes. The main objective was to develop a Python implementation of the PEG algorithm, focusing on generating Tanner graphs and corresponding parity check matrices. Through this implementation, we aimed to provide a tool for efficiently creating LDPC codes with varying lengths and rates.

The investigation began with a thorough review of LDPC codes and the PEG algorithm, emphasizing the importance of sparse parity check matrices and efficient graph construction. By implementing the PEG algorithm, we have demonstrated its effectiveness in generating LDPC codes with desirable properties, such as high girth and low error-floor characteristics.

The technical core of the thesis detailed the step-by-step process of implementing the PEG algorithm in Python. We described the creation of Tanner graphs and the generation of parity check matrices, highlighting the challenges faced and the strategies employed to overcome them. The code provided offers a practical tool for researchers and practitioners in the field of error-correcting codes.

Furthermore, the testing and visualization of graphs and subgraphs using NetworkX and Matplotlib provided valuable insights into the algorithm's behavior and the connectivity between nodes. These visualizations illustrated the evolving structure of the Tanner graph, showcasing the algorithm's ability to efficiently connect symbol nodes to check nodes.

The integration of these components into a frontend ...

In conclusion, ...

[Summarize the main aspects and results of the research project. Provide an answer to the research questions stated earlier.]

(target size: 1/2 page)