Analysis: Implementation of the Progressive Edge Growth (PEG) algorithm with Javascript to generate and visualize Tanner graphs for LDPC (Low-Density Parity-Check) codes

Progressive Edge Growth (PEG) Algorithm

Initialization

- The algorithm starts with an empty Tanner graph.
- It initializes the parity check matrix with all elements set to 0.

Symbol Node Degrees:

- For each symbol node, the algorithm specifies the degrees (number of edges) for each symbol node.
- These degrees are provided as input to the algorithm.

Creating Edges:

- The algorithm iterates through each symbol node.
- For each symbol node, it creates edges based on the specified degrees.
- The edges are formed between the symbol node and check nodes.
- The goal is to progressively grow the graph with edges while avoiding short cycles.
  (a "short cycle" refers to a a closed path in a graph that has a small number of edges - short cycles are undesirable because they can lead to decoding failures.)

Progressive Expansion:

- The algorithm uses a progressive expansion strategy to add edges gradually.
- For each symbol node:
  - If it is the first edge, the algorithm finds the check node with the lowest degree in the current graph configuration and creates an edge.
  - If it is not the first edge, the algorithm gradually deepens the subgraph expanding from the symbol node.
  - The expansion continues until conditions are met (either all check nodes are covered or the subgraph stops expanding).

Callback Hook:

- The algorithm supports a callback mechanism (hook) to execute a function each time a new edge is added.
- This allows for storing intermediate steps and visualizing the graph's progression.

Implementation in the Code (`peg.js`)

Initialization:

- The module initializes variables for the number of check nodes, symbol nodes, and symbol node degrees.
- It also keeps track of the Tanner graph and a hook function for callbacks.

Calculate Edges:

- The `calculateEdges` function generates the initial parity check matrix and creates the initial Tanner graph.
- It iterates through symbol nodes, creating edges based on the progressive expansion strategy.
- The hook function is called each time a new edge is added.

API for Creating Tanner Graph:

- The module provides an API (`create` function) to create a Tanner graph with specified parameters.
- The API allows for setting the number of check nodes, symbol nodes, and symbol node degrees.
- It internally calls the `calculateEdges` function to create the graph.

Hook Mechanism:

- The module allows for registering a callback function (`hook`) that gets executed each time a new edge is added to the graph.
- This is useful for storing and visualizing intermediate steps.

Usage in `main.js`

Event Listeners:

- The script in `main.js` listens for DOMContentLoaded event and sets up event listeners for buttons and inputs.

Create Button:

- On clicking the "Create" button, the script retrieves user input for symbol nodes, check nodes, and symbol node degrees.
- It uses the PEG algorithm to create a Tanner graph and renders the final graph.

Next and Previous Buttons:

- "Next" and "Previous" buttons allow users to navigate through intermediate steps of graph creation.
- The script updates the graph display and matrix based on the stored progress.

Depth Input:

- Users can input the depth for subgraph visualization using the depth input field.
- The subgraph is displayed, and users can change the depth dynamically.

Visualization in `tanner-graph.js`

The `TannerGraph` Class:

- The `TannerGraph` class provides methods for rendering the Tanner graph and subgraph using the `vis.js` library.
- It includes functions for creating edges, finding nodes with the lowest degree, and handling subgraph operations.

Code Explanation:

The provided JavaScript code defines a `TannerGraph` class and a `SubGraph` class for representing and manipulating Tanner graphs. These classes are utilized in the context of LDPC (Low-Density Parity-Check) codes.

`TannerGraph` **Class:**

1. **Constructor:** - Takes a parity check matrix (`matrix`) as an argument. - Decomposes the matrix to create symbol nodes, check nodes, and edges. - Symbol nodes represent the variable nodes, and check nodes represent the parity-check nodes in the Tanner graph.
2. `getNode(id)` **Method:** - Returns a node (symbol or check) with the given `id`.
3. `getClone()` **Method:** - Clones the internal matrix of the Tanner graph, creating a new instance.
4. `createEdge(symbolNodeId, checkNodeId)` **Method:** - Creates a new edge between the given symbol and check nodes. - Modifies the internal matrix and updates the connections in the nodes.
5. `getCheckNodeWithLowestDegree()` **and** `getSymbolNodeWithLowestDegree()` **Methods:** - Return the check or symbol node with the lowest degree (fewest connections).
6. `getSubGraph(nodeId, depth)` **Method:** - Creates a sub-graph for the Tanner graph, expanding from the given node with the specified depth.
7. **Rendering (`render` Method):** - Utilizes the `vis.js` library to render the Tanner graph. - Symbol and check nodes are positioned in the graph layout. - The `onClick` callback can be provided to handle double-click events on the graph.

`SubGraph` **Class:**

1. **Constructor:** - Takes a `TannerGraph` instance (`tannerGraph`), a root node ID, and a depth. - Constructs a sub-graph expansion using a breadth-first-search algorithm.
2. `coveredCheckNodes()` **Method:** - Returns a list of check nodes covered by the sub-graph.
3. `allCheckNodesCovered()` **Method:** - Checks if all check nodes in the original Tanner graph are covered by this specific sub-graph.
4. `getUCCheckNodeWithLowestDegree()` **Method:** - Returns the check node with the lowest degree from the original Tanner graph that is not covered by this sub-graph.
5. **Rendering (`render` Method):** - Utilizes `vis.js` to render the sub-graph in a hierarchical layout.

HTML and CSS (`index.html`)

- The HTML structure defines the layout, form inputs, buttons, and graph containers.
- The styles defined in styles.css are intended to create a clean and readable layout for the HTML elements.
- External libraries like Bootstrap and `vis.js` are linked for styling and graph visualization.