#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Structure to represent an edge in the graph

struct Edge {

int src, dest; // Source and destination vertices

double weight; // Weight of the edge

};

// Class to represent a graph with V vertices and E edges

class Graph {

int V, E; // Number of vertices and edges

vector<Edge> edges; // Vector of edges

public:

// Constructor to initialize the graph with V vertices and E edges

Graph(int V, int E);

// Function to add an edge to the graph

void addEdge(int u, int v, double w);

// Function to get all edges of the graph

vector<Edge> getEdges();

// Function to get the number of vertices

int getV();

// Function to get the number of edges

int getE();

};

// Constructor implementation

Graph::Graph(int V, int E) {

this->V = V;

this->E = E;

}

// Function to add an edge to the graph

void Graph::addEdge(int u, int v, double w) {

Edge edge = {u, v, w};

edges.push\_back(edge);

}

// Function to get all edges of the graph

vector<Edge> Graph::getEdges() {

return edges;

}

// Function to get the number of vertices

int Graph::getV() {

return V;

}

// Function to get the number of edges

int Graph::getE() {

return E;

}

// Function to find the parent of a node in the disjoint-set

int findParent(int parent[], int i) {

if (parent[i] == i) {

return i;

}

return findParent(parent, parent[i]);

}

// Function to perform union of two sets

void unionSets(int parent[], int rank[], int x, int y) {

int xroot = findParent(parent, x);

int yroot = findParent(parent, y);

if (rank[xroot] < rank[yroot]) {

parent[xroot] = yroot;

} else if (rank[xroot] > rank[yroot]) {

parent[yroot] = xroot;

} else {

parent[yroot] = xroot;

rank[xroot]++;

}

}

// Function to compare two edges based on their weights

bool compareEdges(Edge a, Edge b) {

return a.weight < b.weight;

}

// Kruskal's algorithm to find Minimum Spanning Tree

vector<Edge> kruskalMST(Graph& graph) {

int V = graph.getV();

vector<Edge> result; // Store the result edges

// Step 1: Sort all the edges in non-decreasing order of their weight

vector<Edge> edges = graph.getEdges();

sort(edges.begin(), edges.end(), compareEdges);

// Allocate memory for creating V subsets

int\* parent = new int[V];

int\* rank = new int[V];

// Create V subsets with single elements

for (int v = 0; v < V; ++v) {

parent[v] = v;

rank[v] = 0;

}

// Number of edges to be taken is equal to V-1

int e = 0; // Initial count of edges in MST

int i = 0; // Initial index of sorted edges

// Process the edges in sorted order

while (e < V - 1 && i < graph.getE()) {

// Pick the smallest edge and increment the index for next iteration

Edge next\_edge = edges[i++];

int x = findParent(parent, next\_edge.src);

int y = findParent(parent, next\_edge.dest);

// If including this edge doesn't cause a cycle

if (x != y) {

result.push\_back(next\_edge); // Include the edge in result

unionSets(parent, rank, x, y); // Union the sets

e++; // Increment the count of edges in MST

}

}

// Free allocated memory

delete[] parent;

delete[] rank;

return result;

}

// Utility function to print the constructed MST

void printMST(vector<Edge> mst) {

cout << "Following are the edges in the constructed MST" << endl;

for (auto edge : mst) {

cout << edge.src << " -- " << edge.dest << " == " << edge.weight << endl;

}

}

// Main function to test the code

int main() {

int V = 4; // Number of vertices

int E = 5; // Number of edges

Graph graph(V, E);

// Adding edges to the graph

graph.addEdge(0, 1, 10);

graph.addEdge(0, 2, 6);

graph.addEdge(0, 3, 5);

graph.addEdge(1, 3, 15);

graph.addEdge(2, 3, 4);

// Run Kruskal's algorithm to find MST

vector<Edge> mst = kruskalMST(graph);

// Print the constructed MST

printMST(mst);

return 0;

}

**Understanding of the Codebase**

The codebase focuses on automatic documentation generation for code using Large Language Models (LLMs) and other NLP techniques. The primary goal is to streamline the documentation process by automating it, ensuring it remains consistent, accurate, and efficient.

**Machine Learning Processes Involved**

**Codebase Traversal**

Traversing the codebase involves reading and analyzing the code structure, identifying key components like classes, functions, and their relationships. This forms the foundation for generating relevant documentation.

**Code Embeddings**

The code uses embedding algorithms to convert code snippets into vector representations. Currently, CodeBERT is used, which encodes both natural language and programming language. These embeddings help in understanding the context and semantics of the code.

**Handling Large Code Files**

For large code files, efficient parsing and segmentation are crucial. The algorithm breaks down the code into manageable chunks to process and analyze without running into memory or performance issues.

**Maintaining Context with Agglomerative Clustering**

Agglomerative Clustering groups similar code snippets together based on their hierarchical relationships. This helps in maintaining context across different parts of the code, making the generated documentation more coherent and relevant.

**Efficient Documentation Generation**

The system generates documentation by summarizing and explaining the code based on its embeddings and clusters. This involves prompt engineering and customizing the output format as per user requirements.

**Tasks Tackled**

1. **Dendrogram Feature Addition**
   * **What**: Implemented a feature to generate dendrograms based on the current clustering algorithm.
   * **Why**: Dendrograms visually represent the hierarchical structure of the clusters, helping in understanding the relationships among code snippets.
   * **Methods**

addition of the using namespace std; directive does not change the core functionality or logic of the code. Therefore, the previously provided explanations and answers remain valid. Here they are adjusted for consistency:

**Understanding of the Codebase**

The codebase focuses on automatic documentation generation for code using Large Language Models (LLMs) and other NLP techniques. The primary goal is to streamline the documentation process by automating it, ensuring it remains consistent, accurate, and efficient.

**Machine Learning Processes Involved**

**Codebase Traversal**

Traversing the codebase involves reading and analyzing the code structure, identifying key components like classes, functions, and their relationships. This forms the foundation for generating relevant documentation.

**Code Embeddings**

The code uses embedding algorithms to convert code snippets into vector representations. Currently, CodeBERT is used, which encodes both natural language and programming language. These embeddings help in understanding the context and semantics of the code.

**Handling Large Code Files**

For large code files, efficient parsing and segmentation are crucial. The algorithm breaks down the code into manageable chunks to process and analyze without running into memory or performance issues.

**Maintaining Context with Agglomerative Clustering**

Agglomerative Clustering groups similar code snippets together based on their hierarchical relationships. This helps in maintaining context across different parts of the code, making the generated documentation more coherent and relevant.

**Efficient Documentation Generation**

The system generates documentation by summarizing and explaining the code based on its embeddings and clusters. This involves prompt engineering and customizing the output format as per user requirements.

**Tasks Tackled**

1. **Dendrogram Feature Addition**
   * **What**: Implemented a feature to generate dendrograms based on the current clustering algorithm.
   * **Why**: Dendrograms visually represent the hierarchical structure of the clusters, helping in understanding the relationships among code snippets.
   * **Methods**: Used agglomerative clustering to generate the dendrogram. Other methods like K-means were not used as they do not provide hierarchical relationships.
2. **Documentation Customization**
   * **What**: Added functionality to customize documentation based on user preferences.
   * **Why**: Customization ensures that the documentation meets specific user needs and formats.
   * **Methods**: Implemented prompt engineering to allow users to define their documentation format. Other methods like template-based generation were considered but deemed less flexible.
3. **Additional Code Features**
   * **What**: Enhanced the code with additional features like code refactoring and automatic test generation.
   * **Why**: These features improve the quality and maintainability of the code.
   * **Methods**: Used static code analysis tools and libraries for refactoring and test generation. Alternative methods were considered but not used due to complexity and integration issues.