Social Network Analysis and Visualization project

□ **Interactive Graph Exploration.**

□ **community detection.**

Promod chandra Das
Id: 231002005
Dept of cse
Cse-206 (223-D5)

□ Code

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>

#define MAX_NODES 100

typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct Graph {
    Node* adjList[MAX_NODES];
    int numVertices;
} Graph;

typedef struct Queue {
    int items[MAX_NODES];
    int front;
    int rear;
} Queue;

// Function to create a new node
Node* createNode(int v) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = v;
    newNode->next = NULL;
    return newNode;
}

// Function to create a graph
Graph* createGraph(int vertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = vertices;

    for (int i = 0; i < vertices; i++) {
        graph->adjList[i] = NULL;
    }

    return graph;
}

// Function to add an edge to the graph
void addEdge(Graph* graph, int src, int dest) {
    Node* newNode = createNode(dest);
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjList[dest];
    graph->adjList[dest] = newNode;
}

// Function to print the graph
void printGraph(Graph* graph) {
    for (int v = 0; v < graph->numVertices; v++) {
        Node* temp = graph->adjList[v];
        printf("Vertex %d:", v);
        while (temp) {
            printf(" -> %d", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}

// Function to create a queue
Queue* createQueue() {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->front = -1;
    queue->rear = -1;
    return queue;
}

// Function to check if the queue is empty
bool isEmpty(Queue* queue) {
    return queue->rear == -1;
}

// Function to enqueue an element
void enqueue(Queue* queue, int value) {
    if (queue->rear == MAX_NODES - 1) {
        printf("Queue is full\n");
        return;
    }
    if (queue->front == -1)
```

```
        queue->front = 0;
    queue->rear++;
    queue->items[queue->rear] = value;
}

// Function to dequeue an element
int dequeue(Queue* queue) {
    int item;
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        return -1;
    } else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
        return item;
    }
}

// Function to perform BFS and find the shortest path
int bfs(Graph* graph, int startVertex, int endVertex) {
    Queue* queue = createQueue();
    bool visited[MAX_NODES] = { false };
    int distance[MAX_NODES];
    for (int i = 0; i < MAX_NODES; i++) {
        distance[i] = INT_MAX;
    }

    visited[startVertex] = true;
    distance[startVertex] = 0;
    enqueue(queue, startVertex);

    while (!isEmpty(queue)) {
        int currentVertex = dequeue(queue);
        Node* temp = graph->adjList[currentVertex];
        while (temp) {
            int adjVertex = temp->data;

            if (!visited[adjVertex]) {
                distance[adjVertex] = distance[currentVertex] + 1;
                visited[adjVertex] = true;
                enqueue(queue, adjVertex);

                if (adjVertex == endVertex) {
                    free(queue);
                    return distance[adjVertex];
                }
            }
            temp = temp->next;
        }
    }

    free(queue);
    return -1; // Return -1 if there's no path between startVertex and endVertex
}

// Function to calculate degree centrality
void degreeCentrality(Graph* graph, int* centrality) {
    for (int i = 0; i < graph->numVertices; i++) {
        centrality[i] = 0;
        Node* temp = graph->adjList[i];
        while (temp) {
            centrality[i]++;
            temp = temp->next;
        }
    }
}

// Function to detect communities using modularity optimization
void detectCommunities(Graph* graph, int* community) {
    // Placeholder implementation for demonstration
    for (int i = 0; i < graph->numVertices; i++) {
        community[i] = i % 2; // Assign vertices alternately to two communities
    }
}

// Main function
int main() {
    int vertices, edges, startVertex, endVertex;

    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    Graph* graph = createGraph(vertices);

    printf("Enter the number of edges: ");
    scanf("%d", &edges);

    printf("Enter the edges (format: src dest):\n");
    for (int i = 0; i < edges; i++) {
        int src, dest;
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }

    printf("Graph adjacency list:\n");
    printGraph(graph);

    printf("Enter the start vertex: ");
    scanf("%d", &startVertex);

    printf("Enter the end vertex: ");
    scanf("%d", &endVertex);

    int distance = bfs(graph, startVertex, endVertex);
    if (distance != -1) {
        printf("Shortest path between %d and %d is %d\n", startVertex, endVertex, distance);
    } else {
        printf("No path found between %d and %d\n", startVertex, endVertex);
    }

    int centrality[MAX_NODES];
    degreeCentrality(graph, centrality);
    printf("Degree centrality of vertices:\n");
    for (int i = 0; i < vertices; i++) {
        printf("Vertex %d: %d\n", i, centrality[i]);
    }

    int community[MAX_NODES];
    detectCommunities(graph, community);
    printf("Community assignment of vertices:\n");
    for (int i = 0; i < vertices; i++) {
        printf("Vertex %d: Community %d\n", i, community[i]);
    }

    // Free memory
    for (int i = 0; i < vertices; i++) {
        Node* temp = graph->adjList[i];
        while (temp) {
            Node* next = temp->next;
            free(temp);
            temp = next;
        }
    }
    free(graph);

    return 0;
}
```

□ **Output**

```
Enter the number of vertices: 5
Enter the number of edges: 4
Enter the edges (format: src dest):
0 1
0 2
1 3
2 4
Graph adjacency list:
Vertex 0: -> 2 -> 1
Vertex 1: -> 3 -> 0
Vertex 2: -> 4 -> 0
Vertex 3: -> 1
Vertex 4: -> 2
Enter the start vertex: 0
Enter the end vertex: 3
Shortest path between 0 and 3 is 1
Degree centrality of vertices:
Vertex 0: 2
Vertex 1: 2
Vertex 2: 2
Vertex 3: 1
Vertex 4: 1
Community assignment of vertices:
Vertex 0: Community 0
Vertex 1: Community 1
Vertex 2: Community 0
Vertex 3: Community 1
```

Vertex 4: Community 0