HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

**SCHOOL OF ELECTRONICS AND TELECOMMUNICATIONS**

**PROJECT REPORT**

**DATA STRUCTURE AND ALGORITHMS**

**LIFETIME MAXIMIZATION OF AN IOT NETWORK BASED ON GRAPH**

Name of students: Nguyen Quang Huy

Nguyen Quang Thang

Ho Tuan Tu

Name of instructor: Ph.D. Prof. Tran Thi Thanh Hai

Name of referee: ……………………………………

Hanoi, 07-2023

Table of Contents

I. Problem 3

1. Introduction 3

2. Solution 3

3. Definition of the topic 4

II. Data structures 4

1. Graphs 4

2. Queue and Linked List 5

III. Breath First Search Algorithms 5

1. Idea 5

2. Illustration 6

IV. Program Implementation 7

1. Define the library 7

2. Implement Struct 7

3. Create nodes, add edges and make graphs 8

4. Enqueue and Dequeue for BFS 9

5. Function calculate MSE and print out the graph function 10

6. Checking MSE greater than Threshold or not using BFS 10

7. Main function 11

V. The Result 13

1. **PROBLEM**
2. ***Introduction***

The expansion of Internet of Things (IoT) systems leads to massive data produced from a vast variety of connected devices and sensors, providing unprecedented knowledge about the state and processes of the physical world. These IoT sensors are often cheap, wireless and battery-powered. They transmit data sporadically to a Base Station (BS), which forwards the data to a data analytics module.

We consider a deployment of an IoT network in which the underlying graph structure is induced naturally by the physical deployment of the IoT devices. This concept is shown in Figure 1, where the IoT sensors are attached at the vertices of a water supply network. Other similar examples include IoT devices attached to other types of utility networks, sensors deployed along streets, etc.

A picture containing line, diagram, circle

Description automatically generated

1. ***Solution***

In order to increase the longevity of the massive IoT network, each sensor should try to minimize the number of transmissions, while not compromising the quality of the inference at the data analytics module. We address the optimization of the conflicting objectives of network lifetime and correctness of inference by casting the problem in the context of signal processing on graphs. The majority of the paper focuses on signal processing, while the Data Structures and Algorithms section addresses the problem of dividing the IoT devices into graphs for efficient communication with the base station.

1. ***Definition of the topic***

The objective is to partition the set of sensors into subgraphs, such that the subgraphs are disjoint, and all sensors are assigned to exactly one set. In each sampling round the data analytics module reconstructs the state of the graph based only on the transmission of a subset of sensors. This reduces the duty cycle requirements for each sensor, as it has to transmit less often. Partition sampling can then, in principle, increase the IoT network lifetime. However, it comes at the cost of increased error in the reconstructed signal at the receiver, as the data are incomplete. Finding a partition that minimizes that error is critical to achieve the best possible balance between increased battery lifetime and reconstructed signal quality.

1. **DATA STRUCTURE**
2. ***Graph***

* A form of non-linear data structure that represents a set of objects in which objects are connected to each other.
* Objects are represented by vertices/nodes and are connected by edges.
* In general, the graph G (V, E) is a pair of sets (V, E), where V is the set of vertices, E is the set of edges. - Two vertices are called adjacent if they are connected to each other through one edge.

A diagram of graphs and diagrams

Description automatically generated

1. ***Queue and Linked List***

* We define a “enqueue” function to add a new node (IoT devices) into the end of the queue. The new node is added at the rear of the queue.
* The dequeue function is used to remove the first node from the queue. The node is dequeued from the front of the queue.
* This queue is implemented by using the singly linked list.

1. **BREATH FIRST SEARCH ALGORITHM**
2. ***Idea***

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal. That is, we start examining the node A and then all the neighbors of A are examined. In the next step, we examine the neighbors of neighbors of A, so on and so forth. This means that we need to track the neighbors of the node and guarantee that every node in the graph is processed, and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

1. ***Algorithm Illustration***

Step 1: SET STATUS=1 (ready state) for each node in G.

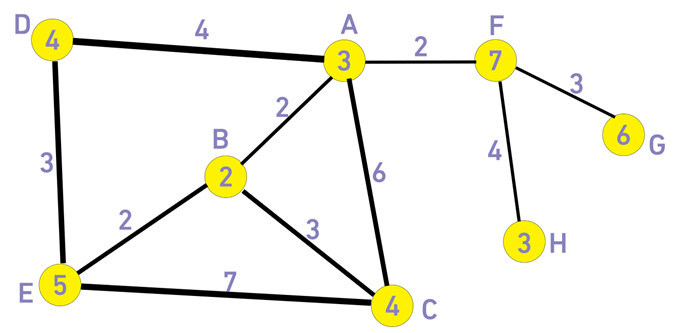
Step 2: Enqueue the starting node A and set its STATUS=2 (waiting state).

Step 3: Repeat Steps 4 and 5 until QUEUE is empty.

Step 4: Dequeue a node N. Process it and set its STATUS=3 (processed state).

Step 5: Enqueue all the neighbors of N that are in the ready state (whose STATUS=1) and set their STATUS=2. (Waiting state) [END OF LOOP]

Step 6: EXIT



|  |  |  |  |
| --- | --- | --- | --- |
| Initialize | Not Traverse | Traversed |  |
| (1) | B, C, D, E, F, G, H | A |  |
| (2) | B, C, E, F, G, H | A, D | A, D |
| (3) | C, E, F, G, H | A, D, B | A, D, B |
| (4) | E, F, G, H | A, D, B, C | A, D, B, C |
| (5) | E, G, H | A, D, B, C, F | A, D, B, C, F |
| (6) | G, H | A, D, B, C, F, E | A, D, B, C, F, E |
| (7) | G | A, D, B, C, F, E, H | A, D, B, C, F, E, H |
| (8) | \0 | A, D, B, C, F, E, H, G |  |

1. **PROGRAM IMPLEMENTATION**
2. ***Define the library:***

#include <stdio.h>

#include <stdlib.h>

#define MAX\_NODES 26

#define MAX\_EDGES 100

1. ***Implement “struct”:***

typedef struct Node {

    char name;

    int weight;

    int num\_edges;

    struct Node\* edges[MAX\_EDGES];

    int edge\_weights[MAX\_EDGES];

} Node;

typedef struct Graph {

    int num\_nodes;

    Node\* nodes[MAX\_NODES];

} Graph;

typedef struct QueueNode {

    Node\* node;

    struct QueueNode\* next; //Linked list FIFO (First-In-First-Out)

} QueueNode;

typedef struct Queue {

    QueueNode\* front; //Queue phía trước

    QueueNode\* rear; //Queue phía sau

} Queue;

1. ***Create nodes, add edges and make graph***

// Create a new node with its name and weights.

Node\* createNode(char name, int weight) {

    Node\* newNode = (Node\*)malloc(sizeof(Node));

    newNode->name = name;

    newNode->weight = weight;

    newNode->num\_edges = 0;

    return newNode;

}

// Create a new graph

Graph\* createGraph() {

    Graph\* newGraph = (Graph\*)malloc(sizeof(Graph));

    newGraph->num\_nodes = 0;

    return newGraph;

}

// Add one edge between 2 nodes with its weight

void addEdge(Node\* node, Node\* edge, int weight) {

    if (node->num\_edges < MAX\_EDGES && edge->num\_edges < MAX\_EDGES) {

        node->edges[node->num\_edges] = edge;

        node->edge\_weights[node->num\_edges] = weight;

        node->num\_edges++;

        edge->edges[edge->num\_edges] = node;

        edge->edge\_weights[edge->num\_edges] = weight;

        edge->num\_edges++;

    }

}

1. ***Enqueue and Dequeue for BFS***

// Add one node into QUEUE

void enqueue(Queue\* queue, Node\* node) {

    QueueNode\* newNode = (QueueNode\*)malloc(sizeof(QueueNode));

    newNode->node = node;

    newNode->next = NULL;

    if (queue->rear == NULL) {

        queue->front = newNode;

        queue->rear = newNode;

    } else {

        queue->rear->next = newNode;

        queue->rear = newNode;

    }

}

// Take the first node out of the Queue

Node\* dequeue(Queue\* queue) {

    if (queue->front == NULL) {

        return NULL;

    } else {

        QueueNode\* temp = queue->front;

        Node\* node = temp->node;

        queue->front = queue->front->next;

        if (queue->front == NULL) {

            queue->rear = NULL;

        }

        free(temp);

        return node;

    }

}

1. ***Function Calculate MSE and print out Graph function***

int calculateMSE(Node\* node) {

    int mse = 0;

    for (int i = 0; i < node->num\_edges; i++) {

        mse += node->weight \* node->edge\_weights[i];

    }

    return mse;

}

// Print out the information of one node and its edges

void printGraph(Node\* node, int nodeNum) {

    printf("Graph %d:\n%c\n", nodeNum, node->name);

}

// Print out the information of the remaining nodes or subgraphs

void printGraphRemain(Node\* node, int nodeNum) {

printf("%c ", node->name);

}

1. ***Checking MSE greater than Threshold or not using BFS***

// Find và print out all subgraphs that are satisfied with MSE > Threshold

void findConnectedGraphs(Graph\* graph, int threshold) {

    int visited[MAX\_NODES] = {0};

    int graphCount = 0; // Count the number of graphs that are satisfied

    for (int i = 0; i < graph->num\_nodes; i++) {

        Node\* node = graph->nodes[i]; //BFS

        if (!visited[node->name - 'A']) {

            Queue\* queue = (Queue\*)malloc(sizeof(Queue));

            queue->front = NULL;

            queue->rear = NULL;

            enqueue(queue, node);

            visited[node->name - 'A'] = 1; //Examine nodes visited

            while (queue->front != NULL) {

                Node\* currNode = dequeue(queue);

                int mse = calculateMSE(currNode);

                if (mse >= threshold) {

                    graphCount++;

                    printGraph(currNode, graphCount);

                }else{

// Add the node to the remainingGraph since it doesn't meet the threshold

remainingGraph->nodes[remainingGraph->num\_nodes++] = currNode;

}

                for (int j = 0; j < currNode->num\_edges; j++) {

                    Node\* nextNode = currNode->edges[j];

                    if (!visited[nextNode->name - 'A']) {

                        enqueue(queue, nextNode);

                        visited[nextNode->name - 'A'] = 1;

                    }

                }

            }

            free(queue);

        }

    }

    printf("\nNumbers of nodes that have MSE > Threshold: %d\n", graphCount);

// Print out the subgraphs (MSE < Threshold)

printf("Remaining Graph (MSE < Threshold) and still connected:\n");

for (i = 0; i < remainingGraph->num\_nodes; i++) {

Node\* node = remainingGraph->nodes[i];

printGraphRemain(node, i + 1);

}

}

1. ***Main Function***

int main() {

    Graph\* graph = createGraph();

    // Add nodes into the graph

    Node\* nodeA = createNode('A', 3);

    Node\* nodeB = createNode('B', 2);

    Node\* nodeC = createNode('C', 4);

    Node\* nodeD = createNode('D', 4);

    Node\* nodeE = createNode('E', 5);

    Node\* nodeF = createNode('F', 7);

    Node\* nodeG = createNode('G', 6);

    Node\* nodeH = createNode('H', 3);

    // Add edges

    addEdge(nodeA, nodeB, 2);

    addEdge(nodeA, nodeC, 6);

    addEdge(nodeA, nodeD, 4);

    addEdge(nodeA, nodeF, 2);

    addEdge(nodeB, nodeC, 3);

    addEdge(nodeB, nodeE, 2);

    addEdge(nodeC, nodeE, 7);

    addEdge(nodeD, nodeE, 3);

    addEdge(nodeE, nodeF, 1);

    addEdge(nodeF, nodeG, 3);

    addEdge(nodeF, nodeH, 4);

    graph->nodes[graph->num\_nodes++] = nodeA;

    graph->nodes[graph->num\_nodes++] = nodeB;

    graph->nodes[graph->num\_nodes++] = nodeC;

    graph->nodes[graph->num\_nodes++] = nodeD;

    graph->nodes[graph->num\_nodes++] = nodeE;

    graph->nodes[graph->num\_nodes++] = nodeF;

    graph->nodes[graph->num\_nodes++] = nodeG;

    graph->nodes[graph->num\_nodes++] = nodeH;

    int threshold;

    printf("Enter Threshold: ");

    scanf("%d", &threshold); //Enter Threshold

    findConnectedGraphs(graph, threshold);

    return 0;

}

1. THE RESULT

This demo (below) means 3 graphs (nodes) need to be run separately to increasing the lifetime for others IoT devices with the trust of the information is 45 unit.

