DSA ASSIGNMENT (TASK 4)

Name : **Avinash R Changrani**

Roll_no. : **CS20B1044**

Date of Submission : **28-06-21**

Problem Title : **To find the shortest/fastest path from one place to another.**

Code :

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

// A structure to represent an adjacency list node
struct AdjListNode{
    int dest;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList{
    struct AdjListNode *head;
};

/* A structure to represent graph which has number of vertices and an
adjacency list array to store
adjacent vertices of the vertices */
struct Graph{
    int V;
    struct AdjList* array;
};

// Structure for Queue Linked List Implementation
struct Node{
    int data;
```

```
    struct Node *next;
};

struct Queue{
    struct Node *front;
    struct Node *rear;
};

// A function to create a new queue
struct Queue *CreateQueue() {
    struct Queue *q = (struct Queue*)malloc(sizeof(struct Queue));
    if (q == NULL)
        return NULL;
    q->front = q->rear = NULL;
    return q;
};

// A function to add an element to the queue
void Enqueue(struct Queue *q, int data){
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
    newnode->data = data;
    newnode->next = NULL;
    if (q->rear == NULL){
        q->rear = q->front = newnode;
        return;
    }
    q->rear->next = newnode;
    q->rear = newnode;
}

// A function to remove an element form the queue
void Dequeue(struct Queue *q){
    if (q->front ==  NULL)
        return;
    struct Node *temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL)
        q->rear = NULL;
    free(temp);
}
```

```
// A recursive function to reverse a Queue
void reverseQueue(struct Queue *q){
    // Base case
    if (q->front ==  NULL)
        return;
    // Dequeue current item (from front)
    int data = q->front->data;
    Dequeue(q);
    // Reverse remaining queue
    reverseQueue(q);
    // Enqueue current item (to rear)
    Enqueue(q,data);
}

// A function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest){
    struct AdjListNode* newNode = (struct AdjListNode*)
malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
};

// A function to create a new graph
struct Graph* createGraph(int V){
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;
    // Creating an array of adjacency lists of size V(number of vertices)
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));
    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;
    return graph;
};

// A function to add an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest){
    // Adding an edge from src to dest.  A new node is added to the
beginning of the adjacency list of src.
```

```
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// A function to print the adjacency list representation of graph
void printAdjlist(struct Graph* graph){
    for (int v = 0; v < graph->V; ++v){
        struct AdjListNode* pCrawl = graph->array[v].head;
        printf("\n Adjacency list of vertex %d :\n", v);
        while (pCrawl){
            printf(" %d ", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

/* A function which uses a modified version of BFS that stores predecessor
of each vertex in array pred and
its distance from source in array dist and which returns true or false
depending on the path existence */
bool BFS(struct Graph* graph, int s, int dest, int v, int pred[], int
dist[]){
    /*Creating a queue to maintain queue of vertices whose adjacency list
is to be scanned */
    struct Queue *Q = CreateQueue();
    // An array that stores whether the vertex is visited or not
    bool visited[v];
    /* initially all vertices are unvisited so visited[i] for all i is
false and as no path is yet constructed,
    dist[i] for all i set to infinity */
    for (int i = 0; i < v; i++) {
        visited[i] = false;
        dist[i] = INT_MAX;
```

```
        pred[i] = -1;
    }
    // Now source is visited so visited[s] is true
    visited[s] = true;
    // distance of source from source is 0
    dist[s] = 0;
    Enqueue(Q,s);
    // Normal BFS Algorithm
    while(Q->front != NULL){
        int u = Q->front->data;
        // printf("%d ",u); test
        Dequeue(Q);
        struct AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl){
            if (visited[pCrawl->dest] == false) {
                visited[pCrawl->dest] = true;
                dist[pCrawl->dest] = dist[u] + 1;
                pred[pCrawl->dest] = u;
                Enqueue(Q,pCrawl->dest);
                // Breaking the loop if we have reached the destination
and returning true
                if (pCrawl->dest == dest)
                    return true;
            }
            pCrawl = pCrawl->next;
        }
    }
    // If destination can't be reached after BFS then returning false
    return false;
}

// Function to print the shortest path from source vertex to destination
vertex
void printshortestpath(struct Graph* graph, int s, int dest, int v){
    // Arrays to store predecessor of i and distance from source
    int pred[v], dist[v];
    // If path not found then return
    if (BFS(graph, s, dest, v, pred, dist) == false) {
        printf("Source and Destination not connected \n");
        return;
```

```
    }
    // Creating a queue to store the shortest path between source and
destination
    struct Queue *path = CreateQueue();
    int crawl = dest;
    Enqueue(path, crawl);
    while (pred[crawl] != -1) {
        Enqueue(path, pred[crawl]);
        crawl = pred[crawl];
    }
    // Reversing the queue as the path is from destination to source and
we want from source to destination
    reverseQueue(path);
    // Printing the path from source to destination
    printf("The shortest path is \n");
    struct Node *ptr = path->front;
    while(ptr != NULL){
        int data = path->front->data;
        Dequeue(path);
        printf("%d ", data);
        ptr = ptr->next;
    }
}

// A function to free the memory allocated to the graph
void deletegraph(struct Graph* graph){
   for (int i = 0; i < graph->V; i++) {
        free(graph->array[i].head);
    }
    free(graph);
}

void main(){
    // As number of vertices in the given Example was 6
    int V = 6;

    // Creating the graph based on the example taken
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1);
    addEdge(graph, 1, 2);
```

```
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);
    addEdge(graph, 4, 5);
    addEdge(graph, 1, 5);
    addEdge(graph, 2, 5);
    addEdge(graph, 2, 4);

    //Printing the adjacency list representation of the above graph
    printf("The Adjacency list of the above graph is :\n");
    printAdjlist(graph);
    // For the example taken in task 2 and task 3
    // printshortestpath(graph, 0,5,6);
    int source, destination;
    printf("\nPlease enter the source and destination to find the shortest
path in the given graph:\n");
    scanf("%d %d",&source,&destination);
    printshortestpath(graph,source,destination,V);
    /* For graph based on user input
    int choice, v, src, dst, source, destination;
    do {
        printf("Graph based on user input : \n 1) Create Graph \n 2) Add
Edge for the graph \n 3) Display Adjacency List \n 4) Shortest Path \n 0)
Exit");
        printf("\nEnter any choice between 0-3 : ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                printf("Enter number of the vertices of the graph you want
to create \n");
                scanf("%d", &v);
                 struct Graph* user_graph = createGraph(v);
                break;
            case 2:
                printf("Enter the source and destination of the edge you
want to create (Indices start from 0 to %d) \n", v - 1);
                scanf("%d %d", &src, &dst);
                if (!range(v,src,dst))
                    printf("Either the source or destination not in
between 0 and %d\n", v - 1);
```

```
                else
                    addEdge(user_graph,src,dst);
                break;
            case 3:
                printf("The Adjacency list of the above graph is :\n");
                printAdjlist(user_graph);
                break;
            case 4:
                printf("Enter the source and destination you want to find
shortest path from \n");
                scanf("%d %d", &source, &destination);
                printshortestpath(user_graph,source,destination,v);
                printf("\n");
                break;
        }
    } while (choice != 0); */
    // Freeing graph
    deletegraph(graph);

}
```

Sample Output (According to code) :

```
PS C:\Users\Avinash\Desktop> gcc Assignment.c
PS C:\Users\Avinash\Desktop> ./a.exe
The Adjacency list of the above graph is :

 Adjacency list of vertex 0 :
 1

 Adjacency list of vertex 1 :
 5  2  0

 Adjacency list of vertex 2 :
 4  5  3  1

 Adjacency list of vertex 3 :
 4  2

 Adjacency list of vertex 4 :
 2  5  3

 Adjacency list of vertex 5 :
 2  1  4

Please enter the source and destination to find the shortest path in the given graph:
0 5
The shortest path is
0 1 5
PS C:\Users\Avinash\Desktop>
```

Sample Output (According to modified code so that it's easy to understand example) :