

## PAT 2 Question

26<sup>th</sup> November, 2021

Mayank Gupta 20BCE1538

## BFS and DFS Implementation

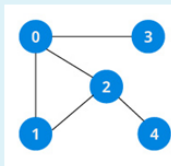
Question 1

Not yet answered

Marked out of 20.00

Flag question

Perform the BFS and DFS operation



## BFS

### BFS pseudocode

```
create a queue Q
```

```
mark v as visited and put v into Q
```

```
while Q is non-empty
```

```
    remove the head u of Q
```

```
    mark and enqueue all (unvisited) neighbours of u
```

# BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

- Step1 Start by putting any one of the graph's vertices at the back of a queue.
- Step2 Take the front item of the queue and add it to the visited list.
- Step3 Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
- Step4 Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

## CODE

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue {
    int items[SIZE];
    int front;
    int rear;
};

struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

struct node {
    int vertex;
    struct node* next;
};
```

```

struct node* createNode(int);

struct Graph {
    int numVertices;
    struct node** adjLists;
    int* visited;
};

// BFS algorithm
void bfs(struct Graph* graph, int startVertex) {
    struct queue* q = createQueue();

    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);

    while (!isEmpty(q)) {
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);

        struct node* temp = graph->adjLists[currentVertex];

        while (temp) {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}

// Creating a node
struct node* createNode(int v) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

```

```

// Creating a graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));

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

    return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Create a queue
struct queue* createQueue() {
    struct queue* q = malloc(sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

// Check if the queue is empty
int isEmpty(struct queue* q) {
    if (q->rear == -1)

```

```

    return 1;
else
    return 0;
}

// Adding elements into queue
void enqueue(struct queue* q, int value) {
    if (q->rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

// Deleting elements from queue
int dequeue(struct queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue is empty");
        item = -1;
    } else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear) {
            printf("Resetting queue ");
            q->front = q->rear = -1;
        }
    }
    return item;
}

// Print the queue
void printQueue(struct queue* q) {
    int i = q->front;

    if (isEmpty(q)) {
        printf("Queue is empty");
    } else {
        printf("\nQueue contains \n");
    }
}

```

```

    for (i = q->front; i < q->rear + 1; i++) {
        printf("%d ", q->items[i]);
    }
}
}

int main() {
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 0, 3);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 4);

    bfs(graph, 0);

    return 0;
}

```

## OUTPUT

```

Queue contains
0 Resetting queue Visited 0

Queue contains
3 2 1 Visited 3

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 Resetting queue Visited 4

...Program finished with exit code 0
Press ENTER to exit console.

```

# DFS

---

## PSEUDO CODE

```
DFS(G, u)
  u.visited = true
  for each v ∈ G.Adj[u]
    if v.visited == false
      DFS(G,v)

init() {
  For each u ∈ G
    u.visited = false
  For each u ∈ G
    DFS(G, u)
}
```

## Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

- Step1 Start by putting any one of the graph's vertices on top of a stack.
- Step2 Take the top item of the stack and add it to the visited list.
- Step3 Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
- Step4 Keep repeating steps 2 and 3 until the stack is empty.

## Code for DFS

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

struct node {
    int vertex;
    struct node* next;
};

struct node* createNode(int v);

struct Graph {
    int numVertices;
    int* visited;
    struct node** adjLists; //to store the array of linked list
};

void DFS(struct Graph* graph, int vertex) {
    struct node* adjList = graph->adjLists[vertex];
    struct node* temp = adjList;

    graph->visited[vertex] = 1;
    printf("Visited %d \n", vertex);

    while (temp != NULL) {
        int connectedVertex = temp->vertex;

        if (graph->visited[connectedVertex] == 0) {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

struct node* createNode(int v) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
```



```

graph->numVertices = vertices;

graph->adjLists = malloc(vertices * sizeof(struct node*));

graph->visited = malloc(vertices * sizeof(int));

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

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

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

// Print the graph
void printGraph(struct Graph* graph) {
    int v;
    for (v = 0; v < graph->numVertices; v++) {
        struct node* temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    struct Graph* graph = createGraph(4);
    addEdge(graph, 0, 1);

```

```
addEdge(graph, 0, 2);
addEdge(graph, 0, 3);
addEdge(graph, 1, 2);
addEdge(graph, 2, 4);

printGraph(graph);


DFS(graph, 2);

return 0;
}
```

### Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of  $O(V + E)$ , where  $V$  is the number of nodes and  $E$  is the number of edges.

The space complexity of the algorithm is  $O(V)$

input

```
Adjacency list of vertex 0
3 -> 2 -> 1 ->

Adjacency list of vertex 1
2 -> 0 ->

Adjacency list of vertex 2
4 -> 1 -> 0 ->

Adjacency list of vertex 3
0 ->
Visited 2
Visited 4
Visited 1
Visited 0
Visited 3
```