**Batch: A2 Roll No.: 16010123032**

**Experiment / assignment / tutorial No. 8**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title:**  **Study of Graph traversal methods** |

**Algorithm for Implementation of DFS:**

 Initialization:

* Create an empty stack S to store vertices during traversal.
* Create a visited[] array of size n (number of vertices) to keep track of visited vertices. Initialize all entries to 0 (unvisited).

 Push the Starting Vertex:

* Push the starting vertex onto the stack.
* Mark the starting vertex as visited.

 DFS Traversal:

* While the stack is not empty:
  1. Pop a vertex current from the stack.
  2. Print or process the current vertex.
  3. For each neighbor i of current, starting from the highest index (for correct order):
     + If the neighbor i is connected to current (i.e., graph[current][i] == 1) and the neighbor has not been visited (visited[i] == 0):
       - Mark i as visited.
       - Push i onto the stack.

 Repeat:

* Continue the process until all vertices are visited or the stack is empty.

 End:

* The traversal ends when all reachable vertices from the starting vertex have been visited.

**Algorithm for Implementation of BFS:**

 Initialization:

* Create an empty queue Q to store vertices to be explored.
* Create a visited[] array of size n (number of vertices) to keep track of visited vertices. Initialize all entries to 0 (unvisited).

 Insert the Starting Vertex:

* Enqueue the starting vertex into the queue Q.
* Mark the starting vertex as visited.

 BFS Traversal:

* While the queue is not empty:
  1. Dequeue a vertex current from the front of the queue.
  2. Print or process the current vertex.
  3. For each neighbor i of current:
     + If the neighbor i is connected to current (i.e., graph[current][i] == 1), and the neighbor has not been visited (visited[i] == 0):
       - Mark i as visited.
       - Enqueue the neighbor i into the queue.

 Repeat:

* Continue the process until all vertices are visited or the queue is empty.

 End:

* The traversal ends when all the reachable vertices from the starting vertex have been visited.

**Implementation Details:**

* 1. **Dfs**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

int graph[MAX][MAX];

int visited[MAX];

int n;

struct Stack {

int items[MAX];

int top;

};

void initialize\_stack(struct Stack\* s) {

s->top = -1;

}

int isFull(struct Stack\* s) {

return s->top == MAX - 1;

}

int isEmpty(struct Stack\* s) {

return s->top == -1;

}

void push(struct Stack\* s, int item) {

if (!isFull(s)) {

s->items[++s->top] = item;

}

}

int pop(struct Stack\* s) {

if (!isEmpty(s)) {

return s->items[s->top--];

}

return -1;

}

void graphTraversal(int start) {

struct Stack s;

initialize\_stack(&s);

push(&s, start);

visited[start] = 1;

while (!isEmpty(&s)) {

int current = pop(&s);

printf("%d ", current);

for (int i = n - 1; i >= 0; i--) {

if (graph[current][i] && !visited[i]) {

visited[i] = 1;

push(&s, i);

}

}

}

}

int main() {

printf("Enter number of vertices: ");

scanf("%d", &n);

printf("Enter adjacency matrix:\n");

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

scanf("%d", &graph[i][j]);

for (int i = 0; i < n; i++)

visited[i] = 0;

int start;

printf("Enter the starting vertex (0 to %d): ", n - 1);

scanf("%d", &start);

if (start < 0 || start >= n) {

printf("Invalid starting vertex.\n");

return 1;

}

printf("Graph traversal starting from vertex %d:\n", start);

graphTraversal(start);

printf("\n");

return 0;

}

* 1. **bfs**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

int graph[MAX][MAX];

int visited[MAX];

int n;

struct Queue {

int items[MAX];

int front;

int rear;

};

void initialize\_queue(struct Queue\* q) {

q->front = -1;

q->rear = -1;

}

int overflow(struct Queue\* q) {

return q->rear == MAX - 1;

}

int underflow(struct Queue\* q) {

return q->front == -1 || q->front > q->rear;

}

void insert(struct Queue\* q, int item) {

if (!overflow(q)) {

if (q->front == -1) q->front = 0;

q->items[++q->rear] = item;

}

}

int delete(struct Queue\* q) {

if (!underflow(q)) {

return q->items[q->front++];

}

return -1;

}

void graphTraversal(int start) {

struct Queue q;

initialize\_queue(&q);

insert(&q, start);

visited[start] = 1;

while (!underflow(&q)) {

int current = delete(&q);

printf("%d ", current);

for (int i = 0; i < n; i++) {

if (graph[current][i] && !visited[i]) {

visited[i] = 1;

insert(&q, i);

}

}

}

}

int main() {

printf("Enter number of vertices: ");

scanf("%d", &n);

printf("Enter adjacency matrix:\n");

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

scanf("%d", &graph[i][j]);

for (int i = 0; i < n; i++)

visited[i] = 0;

int start;

printf("Enter the starting vertex (0 to %d): ", n - 1);

scanf("%d", &start);

if (start < 0 || start >= n) {

printf("Invalid starting vertex.\n");

return 1;

}

printf("Graph traversal starting from vertex %d:\n", start);

graphTraversal(start);

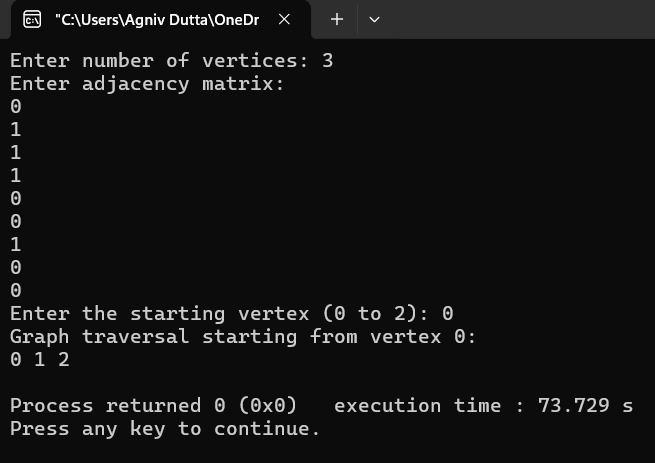
printf("\n");

return 0;

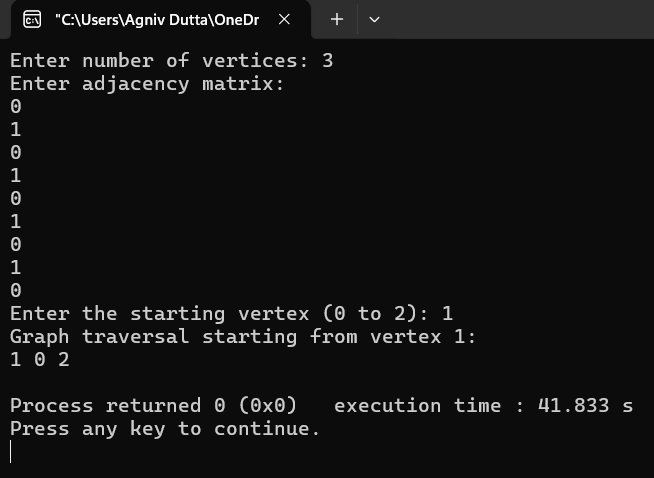
}

**Output Screenshots for Each Operation:**

* + 1. **dfs**

****

**2.) bfs**

****

**Conclusion:-**

In our study of graph traversal techniques, we explored Depth-First Search (DFS) and Breadth-First Search (BFS). DFS explores as far down a branch as possible before backtracking, making it useful for problems like finding connected components and topological sorting. BFS, on the other hand, explores all adjacent nodes level by level, making it ideal for finding the shortest path in unweighted graphs. By implementing these techniques, we gained a deeper understanding of how to efficiently explore and process graph structures.

