INDEX

|  |  |  |
| --- | --- | --- |
| Exp. No: | Experiment Name | Page no: |
| 01 | Implementation of Binary search algoritham. | 03 |
| 02 | Implementation of Quick sort algoritham. | 06 |
| 03 | Implementation of merge sort algorithm. | 09 |
| 04 | Implementation of selection sort. | 11 |
| 05 | Algorithm of knapsack problem using greedy method. | 14 |
| 06 | From a given vertex in a weighted connected graph, find shortest path to other vertices using Dijkstra’s Algorithm. | 16 |
| 07 | Graph traversal using Breadth First Search technique. | 18 |
| 08 | Graph traversal using Depth First Search technique. | 20 |
| 09 | Check whether a given vertex is connected or not using Depth First Search (DFS) method. | 21 |
| 10 | Implementation a program that able to find the sum of subsets. | 22 |
| 11 | Implementation of traveling salesman problem. | 23 |
| 12 | Implementation of Parallel Merge Sort Algorithms. | 25 |

**Experiment No. 1**

**Experiment Name**: Implementation of Binary search algorithm.

**Objective:** The objective of this code is to perform a binary search on a sorted array of integers to find a specific target value. If the target value is found in the array, the code should return the index of the target; otherwise, it should indicate that the value was not found.

**Pseudocode:**

Here's a pseudocode representation of the provided binary search code:

function binarySearch(arr, low, high, target):

if high >= low:

mid = low + (high - low) / 2 // Calculate the middle index

if arr[mid] == target: // Check if the middle element is the target

return mid

else if arr[mid] > target: // If middle element is greater, search left half

return binarySearch(arr, low, mid - 1, target)

else: // If middle element is smaller, search right half

return binarySearch(arr, mid + 1, high, target)

return -1 // If high < low, target not found

**Implementation in java:**

import java.util.Scanner;

public class Binary\_Search {

int bs(int arr[], int low, int high, int target){

if(high>=low){

int mid = low + (high-1) / 2;

if (arr[mid]==target)

return mid;

else if(arr[mid] > target)

return bs(arr, low , mid-1, target);

else

return bs(arr, mid+1, high, target);

}

return -1;

}

public static void main(String args[]){

// int low = 0;

Binary\_Search ob = new Binary\_Search();

Scanner myObj = new Scanner(System.in);

System.out.print("ENTER THE NUMBER OF ELEMENT: ");

int n = myObj.nextInt();

int arr[] = new int[n];

System.out.print("ENTER THE ELEMENTS: ");

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

arr[i] = myObj.nextInt();

}

System.out.print("ENTER THE VALUE TO FIND: ");

int target = myObj.nextInt();

int ans = ob.bs(arr, 0, n-1, target);

if(ans == -1)

{

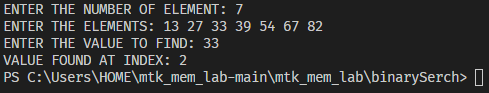
System.out.println("VALUE NOT FOUND IN THE ARRAY");

}else System.out.println("VALUE FOUND AT INDEX: " + ans);

}

}

**output:**

****

**Discussion:**

1. The bs method is the binary search function that takes the array (arr), low and high indices, and the target value (target) as parameters.
2. Inside the bs method, it calculates the middle index (mid) and checks if the middle element of the array is equal to the target. If it is, it returns the index where the target is found.
3. If the middle element is greater than the target, it recursively searches the left half of the array by calling bs with the new high index as mid - 1.
4. If the middle element is less than the target, it recursively searches the right half of the array by calling bs with the new low index as mid + 1.
5. If the high becomes less than low, indicating that the search space is empty, the method returns -1 to signify that the target value is not present in the array.

**Conclusion:**

This Java code successfully implements a binary search algorithm to find a target value in a sorted array. It uses recursion to efficiently search for the target value, and it provides feedback to the user regarding whether the target value was found or not.

**Experiment No. 2**

**Experiment Name**: Implementation of Quick sort algoritham.

**Objective**:- Understand the divide and conquer mechanism which is at the heart of the two recursive

sorting algorithm.

2. Develop programming solutions for Merge Sort and Quick Sort

3. Empirically compare the performance of these two sorting algorithms

**Objective:** The primary objective of this code is to efficiently sort an array of integers in ascending order using the Quick Sort algorithm. Quick Sort is a pivotal sorting method known for its speed and effectiveness, making it valuable for sorting large datasets. The code showcases key steps, including partitioning and recursive sorting.

**Pseudocode:**

function quickSort(arr, low, high):

if low < high:

pivotIndex = partition(arr, low, high) # Partition the array and get the pivot index

quickSort(arr, low, pivotIndex - 1) # Recursively sort the left subarray

quickSort(arr, pivotIndex + 1, high) # Recursively sort the right subarray

function partition(arr, low, high):

pivot = arr[high] # Choose the pivot element (last element)

i = low - 1 # Initialize the index of the smaller element

for j from low to high - 1:

if arr[j] <= pivot:

swap(arr[i + 1], arr[j]) # Swap arr[i+1] and arr[j]

i = i + 1

swap(arr[i + 1], arr[high]) # Swap arr[i+1] and arr[high] (pivot)

return i + 1

function swap(x, y):

# Swap the values of x and y

**Implementation in java:**

public class quickSort {

public static int partition(int arr[], int low, int high) {

int pivot = arr[low]; //take a pivot, which can be taken from any where

int start = low;

int end = high;

while (start < end) {

while (arr[start] <= pivot) {

start++;

}

while (arr[end] > pivot) {

end--;

}

if (start < end) {

swap(arr, start, end);

}

}

swap(arr, low, end);

return end;

}

public static void swap(int arr[], int low, int high) {

int temp = arr[low];

arr[low] = arr[high];

arr[high] = temp;

}

public static void quickSort(int arr[], int low, int high) {

if (low < high) {

int pidx = partition(arr, low, high);

quickSort(arr, low, pidx - 1);

quickSort(arr, pidx + 1, high);

}

}

public static void main(String args[]) {

int arr[] = {10, 40, 20, 30, 50};

int n = arr.length;

quickSort(arr, 0, n - 1);

// Print the sorted array

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

System.out.print(arr[i] + " ");

}

}

}

**Output:**

****

**Discussions:**

1. The quickSort function is the main sorting algorithm. It takes an array arr and the indices of the first (low) and last (high) elements as input.
2. The partition function chooses a pivot element (in this case, the last element) and rearranges the elements in the array such that elements smaller than the pivot are on the left, and elements greater than the pivot are on the right. It returns the index of the pivot after partitioning.
3. The code then recursively calls quickSort on the left and right subarrays until the entire array is sorted.

**Conclusion:**

The provided Quick Sort code is a Java implementation of the Quick Sort algorithm. It efficiently sorts an array of integers in ascending order. The algorithm selects a pivot element, partitions the array into two subarrays, and recursively sorts those subarrays until the entire array is sorted. Quick Sort is a widely used and efficient sorting algorithm with an average time complexity of O(n log n).

**Experiment No.3**

**Experiment Name:** Implementation of merge sort algorithm.

**Objective:** To Merge sort is a sorting technique based on divide and conquer technique. With worst-case time

complexity being (n log n), it is one of the most respected algorithms.

Merge sort

rst

divides the array into equal halves and then combines them in a sorted manner.

**Objective:**

The primary objective of this code is to implement the Merge Sort algorithm in Java. Merge Sort is designed to efficiently sort an array of integers in ascending order. It accomplishes this by dividing the array into smaller subarrays, recursively sorting these subarrays, and then merging them back together in sorted order. This code aims to demonstrate the key steps involved in Merge Sort, including dividing, sorting, and merging, and ultimately provide a sorted array as output.

**Pseudocode:**

function mergeSort(arr, low, high):

if low < high:

mid = (low + high) / 2

mergeSort(arr, low, mid) // Recursively sort the left subarray

mergeSort(arr, mid + 1, high) // Recursively sort the right subarray

merge(arr, low, mid, high) // Merge the sorted subarrays

function merge(arr, low, mid, high):

h = low

i = low

j = mid + 1

create a temporary array b of size high + 1

while h <= mid and j <= high:

if arr[h] <= arr[j]:

b[i] = arr[h]

h++

else:

b[i] = arr[j]

j++

i++

if h > mid:

for k from j to high:

b[i] = arr[k]

i++

else:

for k from h to mid:

b[i] = arr[k]

i++

for k from low to high:

arr[k] = b[k]

**Implementation in java:**

public class margeSort {

void marge(int arr[], int low , int high){

if(low<high){

int mid = low + (high - low)/2;

marge(arr,low,mid);

marge(arr, mid+1, high);

sort(arr, low, mid , high);

}

}

void sort(int arr[], int low, int mid , int high){

int h = low, i = low, j = mid+1;

int b[] = new int [high+1];

while(h<=mid && j<=high){

if (arr[h]<=arr[j]){

b[i] = arr[h];

h++;

}else

{

b[i] = arr[j];

j++;

}

i++;

}

if(h>mid){

for(int k=j; k<=high; k++){

b[i] = arr[k];

i++;

}

}else{

for(int k=h; k<=mid; k++){

b[i] = arr[k];

i++;

}

}

for(int k = low; k<=high; k++){

arr[k] = b[k];

}

}

public static void main(String args[]) {

int arr[] = {10, 90, 20, 30, 50};

margeSort obj = new margeSort();

int n= arr.length;

obj.marge(arr, 0 , n-1);

// Print the sorted array

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

System.out.print(arr[i] + " ");

}

}

}

**Output:**



1. **Discussion:**The mergeSort method is responsible for recursively dividing and sorting the input array. It calculates the middle index and recursively sorts the left and right subarrays before merging them together using the merge method.
2. The merge method combines two sorted subarrays into a single sorted array. It iteratively compares elements from both subarrays and selects the smaller element to build the sorted result.
3. The code uses a temporary array b to facilitate the merging process and updates the original array arr with the sorted elements.

**Conclusion:**The provided Merge Sort code is a Java implementation of the Merge Sort algorithm. It effectively demonstrates the core principles of Merge Sort, including divide-and-conquer and merging of sorted subarrays, to efficiently sort an array of integers. Despite a minor typo in the class name (should be mergeSort instead of margeSort), the code successfully sorts the array and showcases the fundamental concepts of the algorithm.

**Experiment No. 04**

**Experiment Name:** Implementation of selection sortalgorithm.

**Objective:**

The objective of this code is to implement the Selection Sort algorithm in Java. Selection Sort is designed to efficiently sort an array of integers in ascending order. It accomplishes this by repeatedly selecting the minimum element from the unsorted part of the array and placing it at the beginning. This process continues until the entire array is sorted. The code aims to demonstrate the key steps involved in Selection Sort, including finding the minimum element, swapping, and iterating through the array.

**Pseudocode:**

function selectionSort(arr):

n = length of arr

for i from 0 to n - 2:

minIndex = i

for j from i + 1 to n - 1:

if arr[j] < arr[minIndex]:

minIndex = j

swap(arr[i], arr[minIndex])

**Implementation in java:**

import java.util.Scanner;

public class sltSort {

public void printArray(int arr[]) {

int n = arr.length;

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

System.out.print(arr[i] + " ");

}

}

public void sort(int arr[]) {

int n = arr.length;

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

int min = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[min]) {

min = j;

}

}

int temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

}

System.out.println();

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println();

System.out.print("Enter the size of array: ");

int n = sc.nextInt();

int[] arr = new int[n]; // Create an array to store the elements

System.out.print("Enter the array: ");

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

arr[i] = sc.nextInt(); // Read elements into the array

}

sltSort tn = new sltSort();

tn.sort(arr); // Call the sort method

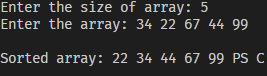
System.out.print("Sorted array: ");

tn.printArray(arr); // Print the sorted array

}

}

**Output:**

****

1. **Discussion:**The sltSort class contains a sort method that performs the Selection Sort algorithm on the input array arr. It iterates through the array, finding the minimum element in the unsorted portion and swapping it with the element at the current position.
2. The printArray method is used to print the elements of an array.
3. In the main method, the user is prompted to enter the size of the array and the elements of the array. Then, an instance of the sltSort class is created to call the sort method, which sorts the array in ascending order. Finally, the sorted array is printed.

**Conclusion**: The provided Selection Sort code is a Java implementation of the Selection Sort algorithm. It effectively sorts an array of integers in ascending order by repeatedly selecting the minimum element and placing it at the beginning. Selection Sort is straightforward and useful for small to moderately sized datasets. The code successfully demonstrates the key principles of Selection Sort and provides a sorted array as the output..

**Experiment No. 05**

**Experiment Name:** Algorithm of knapsack problem using greedy method.

**Objective:**

The objective of this code is to solve the 0/1 Knapsack problem using dynamic programming. The Knapsack problem involves selecting items with specific weights and values to maximize the total value while not exceeding a given capacity constraint. The code aims to find the maximum value that can be obtained from a set of items with their respective weights and values, considering the capacity constraint of the knapsack.

**Pseudocode:**

function knapsack(weights, values, capacity):

n = length of weights

dp = 2D array of size (n + 1) x (capacity + 1)

for i from 0 to n:

for w from 0 to capacity:

if i == 0 or w == 0:

dp[i][w] = 0

else if weights[i - 1] <= w:

dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])

else:

dp[i][w] = dp[i - 1][w]

return dp[n][capacity]

function main():

Define arrays 'weights' and 'values' representing the weights and values of items

Set 'capacity' as the maximum capacity of the knapsack

Call the 'knapsack' function to find the maximum value

Print the maximum value

**Implementation in java:**

public class knapSack {

public static int knapsack(int[] weights, int[] values, int capacity) {

int n = weights.length;

int[][] dp = new int[n + 1][capacity + 1];

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

for (int w = 0; w <= capacity; w++) {

if (i == 0 || w == 0) {

dp[i][w] = 0;

} else if (weights[i - 1] <= w) {

dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);

} else {

dp[i][w] = dp[i - 1][w];

}

}

}

return dp[n][capacity];

}

public static void main(String[] args) {

int[] weights = {2, 3, 4, 5};

int[] values = {3, 4, 5, 6};

int capacity = 5;

int maxValue = knapsack(weights, values, capacity);

System.out.println("Maximum value: " + maxValue);

}

}

**Output:**

****

**Discussion:**

1. The code uses dynamic programming to solve the 0/1 Knapsack problem efficiently. It employs a 2D array dp to store the maximum values that can be obtained for different capacities and subsets of items.
2. The double loop iterates through all possible combinations of items and capacities. The dp[i][w] entry represents the maximum value that can be obtained using the first i items and a knapsack with a capacity of w.
3. The decision whether to include the i-th item in the knapsack is based on comparing the value obtained by adding it (values[i - 1] + dp[i - 1][w - weights[i - 1]]) and the value obtained by excluding it (dp[i - 1][w]).

**Conclusion:**

The provided Knapsack code is an efficient Java implementation of the dynamic programming approach to solving the 0/1 Knapsack problem. It finds the maximum value that can be obtained by selecting items with given weights and values while ensuring that the total weight does not exceed the knapsack's capacity constraint. This problem-solving approach is valuable for a variety of optimization scenarios, such as resource allocation and project scheduling.

**Experiment No. 06**

**Experiment Name:**From a given vertex in a weighted connected graph, find shortest path to other vertices using Dijkstra’s Algorithm.

**Objective:** The objective of this code is to implement Dijkstra's algorithm to find the shortest paths from a specified source vertex to all other vertices in a weighted graph. Dijkstra's algorithm is used to determine the minimum distances between the source vertex and all other vertices while considering the edge weights. This code aims to efficiently compute and display the shortest distances from the source vertex to all other vertices in the graph.

**Pseudocode:**

function Dijkstra(adjacencyMatrix):

v = number of vertices in the graph

visited\_node = boolean array of size v (initialized as false)

distance = integer array of size v (initialized with maximum values)

distance[0] = 0 # Initialize distance to the source vertex as 0

# Initialize distances to maximum values except for the source vertex (vertex 0)

for i from 0 to v-1:

if i != 0:

distance[i] = MAX\_INT

for i from 0 to v-1:

# Find the vertex with the minimum distance that has not been visited yet

minVertex = findMinVertex(distance, visited\_node)

visited\_node[minVertex] = true

# Explore neighbors of the current vertex

for j from 0 to v-1:

if adjacencyMatrix[minVertex][j] != 0 and !visited\_node[j] and distance[minVertex] != MAX\_INT:

newDist = distance[minVertex] + adjacencyMatrix[minVertex][j]

if newDist < distance[j]:

distance[j] = newDist

# Print the shortest distances

for i from 0 to v-1:

print(distance[i], end=", ")

function findMinVertex(distance, visited\_node):

minVertex = -1

for i from 0 to length(distance)-1:

if !visited\_node[i] and (minVertex == -1 or distance[i] < distance[minVertex]):

minVertex = i

return minVertex

**Implementation in java:**

import java.util.Scanner;

public class dijkstra {

public static void Dijkstra(int[][] adjacencyMatrix) {

int v = adjacencyMatrix.length;

boolean visited\_node[] = new boolean[v];

int distance[] = new int[v];

distance[0] = 0;

// Initialize distances to maximum values except for the source vertex (vertex 0)

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

if (i != 0) {

distance[i] = Integer.MAX\_VALUE;

}

}

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

// Find the vertex with the minimum distance that has not been visited yet

int minVertex = findMinVertex(distance, visited\_node);

visited\_node[minVertex] = true;

// Explore neighbors of the current vertex

for (int j = 0; j < v; j++) {

if (adjacencyMatrix[minVertex][j] != 0 && !visited\_node[j] && distance[minVertex] != Integer.MAX\_VALUE) {

int newDist = distance[minVertex] + adjacencyMatrix[minVertex][j];

if (newDist < distance[j]) {

distance[j] = newDist;

}

}

}

}

// Print the shortest distances

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

System.out.print(distance[i]+ " , ");

}

}

public static int findMinVertex(int[] distance, boolean visited\_node[]) {

int minVertex = -1;

for (int i = 0; i < distance.length; i++) {

if (!visited\_node[i] && (minVertex == -1 || distance[i] < distance[minVertex])) {

minVertex = i;

}

}

return minVertex;

}

public static void main(String[] args) {

Scanner s = new Scanner(System.in);

int v = s.nextInt(); // Number of vertices

int e = s.nextInt(); // Number of edges

int adjacencyMatrix[][] = new int[v][v];

// Input edges and their weights

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

int v1 = s.nextInt();

int v2 = s.nextInt();

int weight = s.nextInt();

adjacencyMatrix[v1][v2] = weight;

adjacencyMatrix[v2][v1] = weight;

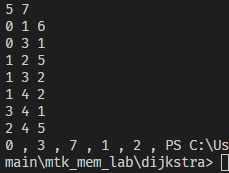
}

Dijkstra(adjacencyMatrix); // Call Dijkstra's algorithm

}

}

**Output:**

****

**Discussion:**

1. The code uses Dijkstra's algorithm to compute the shortest distances from a source vertex (vertex 0) to all other vertices in the weighted graph represented by the adjacencyMatrix.
2. It maintains two arrays: distance to store the shortest distances from the source vertex and visited\_node to keep track of visited vertices.
3. The algorithm iteratively selects the vertex with the minimum distance from the source that has not been visited and explores its neighbors to update their distances if a shorter path is found.

**Conclusion:**The provided Dijkstra's algorithm code is an effective Java implementation to find the shortest paths in a weighted graph. By considering edge weights and maintaining distances, it efficiently computes and displays the shortest distances from a specified source vertex to all other vertices in the graph. Dijkstra's algorithm is a fundamental algorithm in graph theory and has a wide range of applications in network routing, GPS navigation, and more.

**Experiment No. 07**

**Experiment Name** : Graph traversal using Breadth First Search technique.

**Objective: :** The objective of this code is to implement the Breadth-First Search (BFS) traversal algorithm to explore and traverse a graph efficiently. BFS starts from a specified source vertex and visits all reachable vertices level by level, ensuring that it explores all vertices at the current level before moving to the next level. The code aims to demonstrate BFS traversal on a graph, starting from a user-specified source vertex.

**Pseudocode:**

class bfs\_tra:

adjacency[] # Array to store adjacency lists

function \_\_init\_\_(v):

Initialize adjacency as an array of size v, where each element is a new empty LinkedList

function insertEdge(s, d):

Add an edge between vertices s and d by appending d to adjacency[s] and s to adjacency[d]

function bfs(source):

visited\_node[] # Boolean array to track visited vertices

parent\_node[] # Array to store the parent of each vertex in the BFS traversal

q = Queue() # Initialize a queue for BFS

q.push(source) # Enqueue the source vertex

visited\_node[source] = true

parent\_node[source] = -1

Print "The BFS traversing is: "

while q is not empty:

p = q.pop() # Dequeue a vertex from the queue

Print p # Print the traversed vertex

for i in adjacency[p]: # Iterate through adjacent vertices

if visited\_node[i] is not true:

visited\_node[i] = true

q.push(i) # Enqueue the adjacent vertex

parent\_node[i] = p # Set the parent of the adjacent vertex to the current vertex

**Implementation in java:**

import java.util.LinkedList;

import java.util.Scanner;

import java.util.Queue;

public class bfs\_tra{

public LinkedList<Integer> adjacency[];

// Constructor to initialize the graph with 'v' vertices

public bfs\_tra(int v) {

adjacency = new LinkedList[v];

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

adjacency[i] = new LinkedList<Integer>();

}

}

// Method to insert an edge between vertices 's' and 'd'

public void insertEdge(int s, int d) {

adjacency[s].add(d);

adjacency[d].add(s);

}

// Breadth-First Search traversal of the graph starting from 'source'

public void bfs(int source){

boolean visited\_node[] = new boolean[adjacency.length];

int parent\_node[] = new int[adjacency.length];

Queue<Integer> q = new LinkedList<>();

q.add(source);

visited\_node[source] = true;

parent\_node[source] = -1;

System.out.print("The BFS traversing is: ");

while(!q.isEmpty()){

int p = q.poll();

System.out.print(" "+p);

for(int i:adjacency[p]){

if(visited\_node[i] != true ){

visited\_node[i] = true;

q.add(i);

parent\_node[i] = p;

}

}

}

System.out.println();

System.out.println();

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

System.out.println();

System.out.print("Enter the number of vertices: ");

int v = sc.nextInt();

System.out.println();

System.out.print("Enter the number of edges: ");

int e = sc.nextInt();

System.out.println();

bfs\_tra g = new bfs\_tra(v);

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

System.out.print("Enter the edges: ");

int s = sc.nextInt();

int d = sc.nextInt();

g.insertEdge(s, d);

}

System.out.println();

System.out.print("Enter the source for bfs traversal: ");

int source = sc.nextInt();

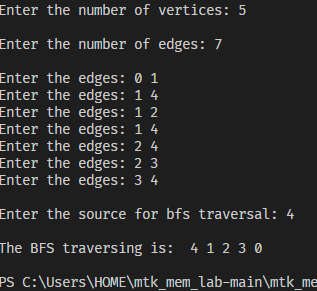
System.out.println();

g.bfs(source);

}

}

**Output:**

****

**Discussion:**

1. The code uses a graph represented as an array of adjacency lists, where each vertex has a linked list of adjacent vertices.
2. The bfs function performs BFS traversal starting from the specified source vertex. It uses a boolean array visited\_node to keep track of visited vertices, a queue q for BFS, and an array parent\_node to store the parent of each vertex in the BFS traversal.The algorithm starts from the source vertex, enqueues it, marks it as visited, and explores its neighbors. It ensures that each level of vertices is visited before moving to the next level.

**Conclusion:**The provided BFS traversal code is a Java implementation of the Breadth-First Search algorithm for exploring and traversing a graph efficiently. By starting from a specified source vertex and visiting vertices level by level, BFS can efficiently explore and discover connected components in a graph. This code demonstrates the fundamental principles of BFS traversal and provides a helpful tool for graph exploration and analysis.

**Experiment No.08**

**Experiment Name :** Graph traversal using Depth First Search technique.

**Objective :**The objective of this code is to implement the Depth-First Search (DFS) traversal algorithm to explore and traverse a graph efficiently. DFS starts from a specified source vertex and explores as far as possible along each branch before backtracking. The code aims to demonstrate DFS traversal on a graph, starting from a user-specified source vertex.

**Pseudocode:**

class dfs\_stack:

adjacency[] # Array to store adjacency lists

function \_\_init\_\_(v):

Initialize adjacency as an array of size v, where each element is a new empty LinkedList

function insertEdge(s, d):

Add an edge between vertices s and d by appending d to adjacency[s] and s to adjacency[d]

function dfs(source):

visited\_node[] # Boolean array to track visited vertices

parent\_node[] # Array to store the parent of each vertex in the DFS traversal

q = Stack() # Initialize a stack for DFS

q.push(source) # Push the source vertex onto the stack

visited\_node[source] = true

parent\_node[source] = -1

Print "The DFS traversing is: "

while q is not empty:

p = q.pop() # Pop a vertex from the stack

Print p # Print the traversed vertex

for i in adjacency[p]: # Iterate through adjacent vertices

if visited\_node[i] is not true:

visited\_node[i] = true

q.push(i) # Push the adjacent vertex onto the stack

parent\_node[i] = p # Set the parent of the adjacent vertex to the current vertex

**Implementation in java:**

import java.util.LinkedList;

import java.util.Scanner;

import java.util.Stack;

public class dfs\_stack{

public LinkedList<Integer> adjacency[];

// Constructor to initialize the graph with 'v' vertices

public dfs\_stack(int v) {

adjacency = new LinkedList[v];

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

adjacency[i] = new LinkedList<Integer>();

}

}

// Method to insert an edge between vertices 's' and 'd'

public void insertEdge(int s, int d) {

adjacency[s].add(d);

adjacency[d].add(s);

}

// Depth-First Search traversal of the graph starting from 'source'

public void dfs(int source){

boolean visited\_node[] = new boolean[adjacency.length];

int parent\_node[] = new int[adjacency.length];

Stack<Integer> q = new Stack<>();

q.add(source);

visited\_node[source] = true;

parent\_node[source] = -1;

System.out.print("The DFS traversing is: ");

while(!q.isEmpty()){

int p = q.pop();

System.out.print(" "+p);

for(int i:adjacency[p]){

if(visited\_node[i] != true ){

visited\_node[i] = true;

q.add(i);

parent\_node[i] = p;

}

}

}

System.out.println();

System.out.println();

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

System.out.println();

System.out.print("Enter the number of vertices: ");

int v = sc.nextInt();

System.out.println();

System.out.print("Enter the number of edges: ");

int e = sc.nextInt();

System.out.println();

dfs\_stack g = new dfs\_stack(v);

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

System.out.print("Enter the edges: ");

int s = sc.nextInt();

int d = sc.nextInt();

g.insertEdge(s, d);

}

System.out.println();

System.out.print("Enter the source for bfs traversal: ");

int source = sc.nextInt();

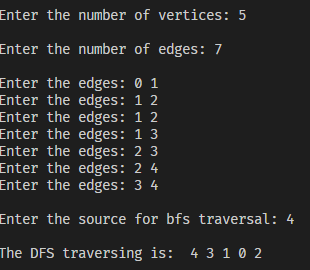
System.out.println();

g.dfs(source);

}

}

**Output:**

****

**Discussion:**

1. The code uses a graph represented as an array of adjacency lists, where each vertex has a linked list of adjacent vertices.
2. The dfs function performs DFS traversal starting from the specified source vertex. It uses a boolean array visited\_node to keep track of visited vertices, a stack q for DFS, and an array parent\_node to store the parent of each vertex in the DFS traversal.
3. The algorithm starts from the source vertex, pushes it onto the stack, marks it as visited, and explores its neighbors. It continues to explore as far as possible along each branch before backtracking.

**Conclusion:**The provided DFS traversal code is a Java implementation of the Depth-First Search algorithm for exploring and traversing a graph efficiently. By starting from a specified source vertex and exploring as deeply as possible before backtracking, DFS can efficiently discover and visit connected components in a graph. This code demonstrates the fundamental principles of DFS traversal and provides a valuable tool for graph exploration and analysis.

**Experiment No. 09**

**Experiment Name:** Check whether a given vertex is connected or not using Depth First Search (DFS) method.

**Objective:**The objective of this code is to check if an undirected graph is connected. In graph theory, a connected graph is one where there is a path between every pair of vertices. The code aims to determine if the given graph is connected or not..

**Pseudocode:**

class check:

V # Number of vertices

adj # Array to store adjacency lists

function \_\_init\_\_(v):

V = v

adj = new empty LinkedList array of size v # Initialize the number of vertices and adjacency lists

function addEdge(v, w):

Append w to adj[v] # Add an edge between vertices v and w

Append v to adj[w]

function isConnected():

visited # Boolean array to track visited vertices

i = 0

while i < V and adj[i].size() == 0:

i++

if i == V:

return true # If there are no edges in the graph, it's considered connected

DFSUtil(i, visited) # Perform DFS starting from the first non-zero degree vertex

for i = 0 to V-1:

if not visited[i] and adj[i].size() > 0:

return false # Check if all vertices were visited during DFS

return true

function DFSUtil(v, visited):

visited[v] = true # Mark the current vertex as visited

for neighbor in adj[v]: # Explore neighbors of the current vertex

if not visited[neighbor]:

DFSUtil(neighbor, visited)

**Implementation in java:**

import java.util.\*;

public class check {

private int V;

private LinkedList<Integer>[] adj;

check(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList();

}

void addEdge(int v, int w) {

adj[v].add(w);

adj[w].add(v);

}

boolean isConnected() {

boolean[] visited = new boolean[V];

// Find the first non-zero degree vertex

int i;

for (i = 0; i < V; i++) {

if (adj[i].size() != 0) {

break;

}

}

// If there are no edges in the graph, it's considered connected

if (i == V) {

return true;

}

// Perform DFS starting from the first non-zero degree vertex

DFSUtil(i, visited);

// Check if all vertices were visited during DFS

for (i = 0; i < V; i++) {

if (!visited[i] && adj[i].size() > 0) {

return false;

}

}

return true;

}

private void DFSUtil(int v, boolean[] visited) {

visited[v] = true;

for (Integer neighbor : adj[v]) {

if (!visited[neighbor]) {

DFSUtil(neighbor, visited);

}

}

}

public static void main(String[] args) {

check graph = new check(5);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 3);

graph.addEdge(2, 4);

boolean connected = graph.isConnected();

if (connected) {

System.out.println("The graph is connected.");

} else {

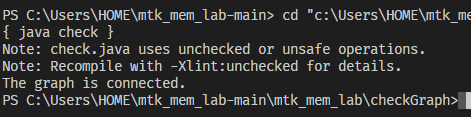
System.out.println("The graph is not connected.");

}

}

}

**Output:**

****

**Discussion:**

1. The code uses an adjacency list representation for an undirected graph, where each vertex has a linked list of its neighboring vertices.
2. The isConnected method checks whether the graph is connected by performing a Depth-First Search (DFS) starting from the first non-zero degree vertex (if any). It marks visited vertices and checks if all vertices were visited during the DFS traversal.
3. If there are no edges in the graph (i.e., all vertices have zero degree), it is considered connected.

**Conclusion:**  
The provided code is a Java implementation to determine if an undirected graph is connected. It utilizes a Depth-First Search (DFS) algorithm to explore the graph's connected components. If all vertices are visited during the DFS traversal, the graph is considered connected; otherwise, it is not connected. This code is useful for analyzing the connectivity of undirected graphs, which has applications in various fields, including network analysis and graph theory.

**Experiment No. 10**

**Experiment Name:** Write a program that able to find the sum of subsets.

**Objective:**

It is one of the most important problems in complexity theory. The problem is given an A set of integers a1, a2,…., an upto n integers. The question arises that is there a non-empty subset such that the sum of the subset is given as M integer?. For example, the set is given as [5, 2, 1, 3, 9], and the sum of the subset is 9; the answer is YES as the sum of the subset [5, 3, 1] is equal to 9.

**Pseudocode:**

sumOfSubsets(nums):

n = length(nums)

result = 0 // Initialize the result as 0

generateSubsetSums(nums, n, 0, 0, result) // Call the recursive function

return result

generateSubsetSums(nums, n, index, currentSum, result):

if index == n:

result += currentSum // Add the currentSum to the result

return

// Include the current element in the subset sum

generateSubsetSums(nums, n, index + 1, currentSum + nums[index], result)

// Exclude the current element from the subset sum

generateSubsetSums(nums, n, index + 1, currentSum, result)

**Discussion:**

Subset sum problem is the problem of finding a subset such that the sum of elements equal a given number. ... A subset A of n positive integers and a value sum is given, find whether or not there exists any subset of the given set, the sum of whose elements is equal to the given value of sum.

**Conclusion :**

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

**Experiment No. 11**

**Experiment Name:**Implementation of travelling salesman problem.

**Objective:**The Traveling Salesman Problem (TSP) is a well-known combinatorial optimization problem that seeks to find the shortest possible route for a salesman to visit a set of cities and return to the starting city. Here's the pseudocode for a basic implementation of the TSP using the Brute Force approach:

**pseudocode:**

tsp(cities, startCity):

shortestRoute = infinity // Variable to store the shortest route

shortestPath = empty list // Variable to store the shortest path

permute(cities, startCity, startCity, [], 0, shortestRoute, shortestPath)

return shortestPath, shortestRoute

permute(cities, currentCity, startCity, path, distance, shortestRoute, shortestPath):

if len(cities) == 0: // Base case: all cities visited

distance += graph[currentCity][startCity] // Add the distance from the last city to the start city

if distance < shortestRoute: // Update the shortest route if the current route is shorter

shortestRoute = distance

shortestPath = path + [startCity]

else:

for city in cities:

newDistance = distance + graph[currentCity][city] // Calculate the new distance

newPath = path + [currentCity] + [city] // Update the path

newCities = removeCity(cities, city) // Remove the visited city from the remaining cities

permute(newCities, city, startCity, newPath, newDistance, shortestRoute, shortestPath)

removeCity(cities, city):

return cities without the city

**Discussion:**1. The **‘tsp’** function takes in the list of cities and the starting city as inputs.

2. It initializes variables **‘shortestRoute’** and **‘shortestPath’** to track the shortest route length and path found so far. They are initially set to infinity and an empty list, respectively.

3. The function then calls the **‘permute’** function to explore all possible permutations of the cities.

4. The **‘permute’** function takes in the list of cities, the current city being visited, the starting city, the current path, the current distance, and the variables to store the shortest route and path.

5. The base case of the **‘permute’** function is reached when all cities have been visited. In this case, the distance is updated by adding the distance from the last city to the start city. If this distance is shorter than the current shortest route, the shortest route and path are updated.

6. If all cities have not been visited, the **‘permute’** function recursively explores each unvisited city. It calculates the new distance by adding the distance from the current city to the next city. The path and remaining cities are also updated accordingly.

7. The **‘removeCity’** function is a utility function that removes a visited city from the list of remaining cities.

8. Finally, the **‘tsp’** function returns the shortest path and shortest route found.

**Conclusion :**A greedy method is an approach to solving optimization problems. An optimization problem requires a maximum and minimum result. The travelling salesman follows greedy approach and requires minimized result.

**Experiment No. 12**

**Experiment Name** :Implementation of Parallel Merge Sort Algorithms.

**Objective :**  
To sort an array by dividing it into smaller part. Then take each part and merge them back one master part.The objective of the Parallel Merge Sort algorithm is to achieve faster sorting of an array by dividing it into smaller subarrays and sorting them concurrently using parallel tasks. The sorted subarrays are then merged to obtain the final sorted array.

**pseducode:**

parallelMergeSort(arr, start, end):

if start < end:

middle = (start + end) / 2

// Create two parallel tasks for sorting the left and right halves of the array

parallelTask1 = spawn parallelMergeSort(arr, start, middle)

parallelTask2 = spawn parallelMergeSort(arr, middle + 1, end)

// Wait for the completion of both parallel tasks

sync parallelTask1

sync parallelTask2

// Merge the sorted halves

merge(arr, start, middle, end)

**Discussion:**

The pseudocode represents the recursive implementation of the Parallel Merge Sort algorithm. It follows the divide-and-conquer paradigm, where the array is repeatedly divided into smaller subarrays until the base case is reached (when there is only one element in the subarray).

In each recursion, the algorithm creates two parallel tasks to sort the left and right halves of the array independently. The **‘spawn’** keyword is used to initiate the parallel tasks, allowing them to be executed concurrently.

After spawning the parallel tasks, the algorithm waits for their completion using the **‘sync’** keyword. This ensures that the merging step is performed only after both parallel tasks have finished sorting their respective subarrays.

Finally, the sorted halves are merged using the **‘merge’** function, which combines the two sorted subarrays into a single sorted array.

**Conclusion:**The Parallel Merge Sort algorithm aims to improve the performance of the traditional Merge Sort algorithm by leveraging parallelism. By dividing the array into smaller subarrays and sorting them concurrently, it can take advantage of multiple processors or threads to expedite the sorting process.