**Course: Design and Analysis of Algorithms**

**Assignment Number: 2**

**Student Name: Sumit Bhikhchandani**

**Roll Number: 21BIT026**

**Date of Submission: 26 April,2024**

**Question 1**

* **Title:** Merge Sort Algorithm Implementation and Analysis
* **Introduction:**

Merge sort is a divide and conquer algorithm that divides the input array into two halves, sorts each half recursively, and then merges the sorted halves. It has a time complexity of O(n log n) in all scenarios.

* **pseudocode:**

MergeSort(arr[], l, r)

if l < r

mid = (l + r) / 2

MergeSort(arr, l, mid)

MergeSort(arr, mid+1, r)

Merge(arr, l, mid, r)

Merge(arr[], l, mid, r)

n1 = mid - l + 1

n2 = r - mid

L[1...n1], R[1...n2]

for i = 1 to n1

L[i] = arr[l + i - 1]

for j = 1 to n2

R[j] = arr[mid + j]

i = 1, j = 1, k = l

while i <= n1 and j <= n2

if L[i] <= R[j]

arr[k] = L[i]

i++

else

arr[k] = R[j]

j++

k++

while i <= n1

arr[k] = L[i]

i++

k++

while j <= n2

arr[k] = R[j]

j++

k++

* **Complexity Analysis:**
* Time Complexity: O(n log n) in all scenarios.

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

* Space Complexity: O(n) due to the additional space required for the temporary arrays during the merge process.

Best Case: O(n)

Average Case: O(n)

Worst Case: O(n)

* **Executable Code:**

#include <iostream>

using namespace std;

void merge(int arr[], int l, int mid, int r) {

int n1 = mid - l + 1;

int n2 = r - mid;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int mid = l + (r - l) / 2;

mergeSort(arr, l, mid);

mergeSort(arr, mid + 1, r);

merge(arr, l, mid, r);

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int arr\_size = sizeof(arr) / sizeof(arr[0]);

cout << "Given array is \n";

for (int i = 0; i < arr\_size; i++)

cout << arr[i] << " ";

cout << endl;

mergeSort(arr, 0, arr\_size - 1);

cout << "\nSorted array is \n";

for (int i = 0; i < arr\_size; i++)

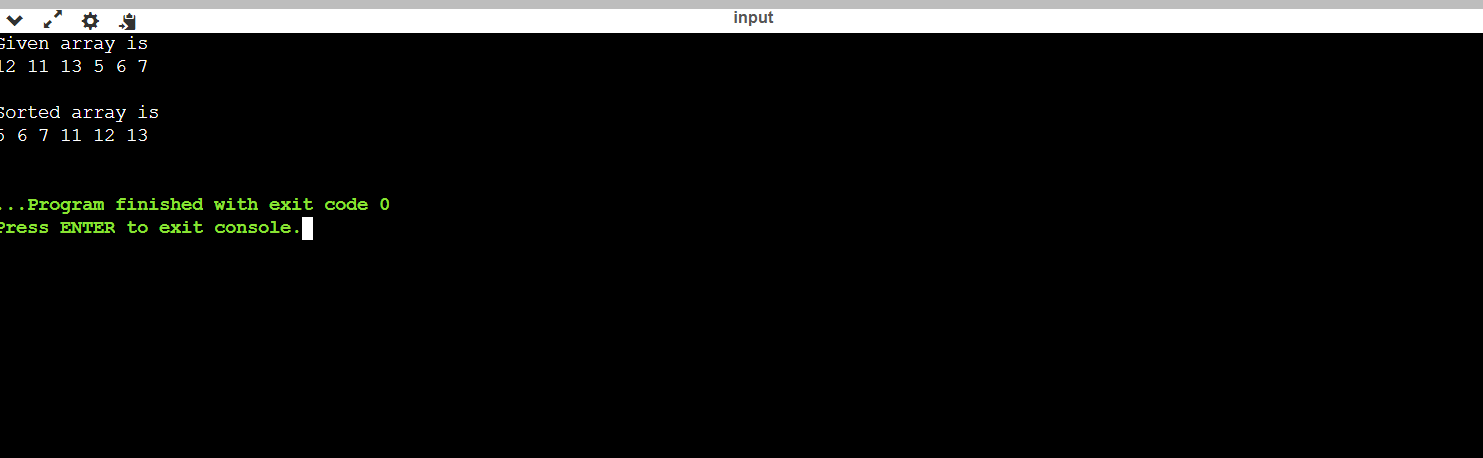
cout << arr[i] << " ";

cout << endl;

return 0;

}

* **Output:**

****

* **Performance Comparison with Quick Sort:**

Quick Sort:

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n^2) (rare, occurs when the pivot selection is poor)

Space Complexity:

O(log n) to O(n) depending on the implementation (in-place partitioning vs. separate auxiliary arrays)

Performance Comparison:

In general, both Merge Sort and Quick Sort have the same average time complexity of O(n log n). However, their performance can vary depending on the characteristics of the input data and the specific implementation details.

Merge Sort:

Stability: Merge Sort is stable, meaning it preserves the order of equal elements in the sorted output.

Performance: It has a consistent performance across all scenarios due to its divide and conquer strategy.

Suitability: Merge Sort performs well on large datasets and is often preferred for external sorting algorithms.

Quick Sort:

In-Place Sorting: Quick Sort is an in-place sorting algorithm, which means it requires only a small, constant amount of additional space.

Cache Efficiency: Due to its in-place partitioning and better cache locality, Quick Sort can be faster in practice, especially for smaller arrays.

Adaptability: Quick Sort can be adapted to efficiently handle partially sorted arrays and is generally faster for nearly sorted data compared to Merge Sort.

Overall:

Choice Dependence: The choice between Merge Sort and Quick Sort depends on various factors such as stability, space efficiency, adaptability to different datasets, and practical performance considerations.

Consideration: Merge Sort is preferred when stability and consistent performance are essential, especially for large datasets. On the other hand, Quick Sort is suitable for in-place sorting and can be faster in practice, particularly for smaller or partially sorted arrays.

* **Code for checking Efficiency:**

#include <iostream>

#include <vector>

#include <chrono>

#include <algorithm>

#include <cstdlib>

#include <ctime>

using namespace std;

using namespace std::chrono;

// Merge Sort implementation

void merge(vector<int>& arr, int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

vector<int> L(n1), R(n2);

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(vector<int>& arr, int l, int r) {

if (l >= r)

return;

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

// Quick Sort implementation

int partition(vector<int>& arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

void quickSort(vector<int>& arr, int low, int high) {

if (low < high) {

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

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Function to generate random array

void generateRandomArray(vector<int>& arr, int n) {

srand(time(NULL));

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

arr.push\_back(rand() % 1000); // Generates random numbers between 0 and 999

}

}

int main() {

vector<int> arr1, arr2;

vector<int> sizes = {10000, 50000, 100000, 200000}; // Example array sizes

for (int n : sizes) {

cout << "Array size: " << n << endl;

// Generate random arrays

generateRandomArray(arr1, n);

arr2 = arr1;

// Measure time taken by Merge Sort

auto startMerge = high\_resolution\_clock::now();

mergeSort(arr1, 0, arr1.size() - 1);

auto stopMerge = high\_resolution\_clock::now();

auto durationMerge = duration\_cast<milliseconds>(stopMerge - startMerge);

cout << "Time taken by Merge Sort: " << durationMerge.count() << " milliseconds" << endl;

// Measure time taken by Quick Sort

auto startQuick = high\_resolution\_clock::now();

quickSort(arr2, 0, arr2.size() - 1);

auto stopQuick = high\_resolution\_clock::now();

auto durationQuick = duration\_cast<milliseconds>(stopQuick - startQuick);

cout << "Time taken by Quick Sort: " << durationQuick.count() << " milliseconds" << endl;

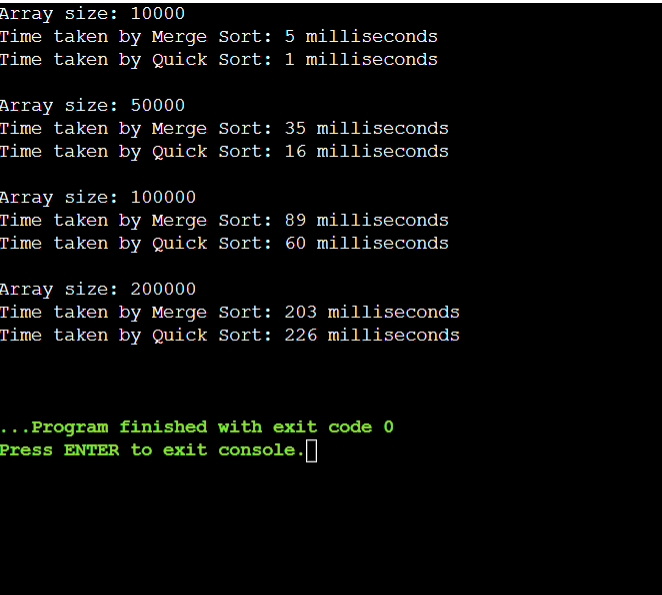
cout << endl;

}

return 0;

}

* **Output:**

****

* **Execution Details of Implementation:**
* Challenges faced:
* Ensuring correct handling of edge cases like empty arrays or single-element arrays.
* How they were addressed:
* Debugged step by step to ensure correct indices manipulation and proper handling of edge cases.
* Implemented appropriate base cases for edge scenarios to avoid errors and ensure the algorithms behave correctly even with small or empty input arrays.

**Question 2**

* **Title:** Kruskal's Algorithm for Minimum Spanning Tree.
* **Introduction:**

Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected graph. It builds the MST by adding edges in increasing order of their weights while avoiding the formation of cycles. An example of Kruskal's algorithm:

Certainly! Let's modify the values of the edges in the given example while preserving the structure of the graph:

Consider a graph with vertices V = {A, B, C, D, E} and edges:

(B, C, 2), (B, D, 3), (A, B, 4), (A, C, 5), (D, E, 4), (C, D, 6), (C, E, 7)

Applying Kruskal's algorithm to this modified graph, it will add edges in the following order:

(B, C, 2), (B, D, 3), (D, E, 4), (A, B, 4)

Resulting in the minimum spanning tree:

(A, B, 4), (B, C, 2), (B, D, 3), (D, E, 4)

* **pseudocode:**

KruskalMST(graph)

sort edges of graph by increasing weight

initialize an empty minimum spanning tree T

for each edge (u, v) in sorted edges

if adding edge (u, v) to T does not create a cycle

add edge (u, v) to T

return T

* **Complexity Analysis:**
* Time Complexity:

Sorting the Edges:

Kruskal's algorithm begins by sorting all edges of the graph in non-decreasing order of their weights.

Sorting |E| edges takes O(E log E) time using efficient algorithms like merge sort or quicksort.

Union-Find Operations:

Kruskal's algorithm employs the disjoint-set data structure (Union-Find) to efficiently detect and merge connected components while avoiding cycles.

The number of Union-Find operations mainly depends on the number of edges and vertices in the graph.

Each Union-Find operation takes nearly constant time, approximately O(α(V)), where α(V) is the inverse Ackermann function, which grows extremely slowly and can be considered constant in practice.

Overall Time Complexity:

Sorting the edges: O(E log E)

Union-Find operations: O(E α(V))

Total time complexity: O(E log E + E α(V))

* Space Complexity:

Space for Storing Edges:

The space required to store the edges of the graph is O(E).

Space for Disjoint-Set Data Structure:

The space complexity of the Union-Find data structure depends on the number of vertices in the graph.

In the worst case, where each vertex is initially in a separate set, the space required is O(V).

Overall Space Complexity:

Total space complexity: O(E + V)

* **Executable Code:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Edge {

int src, dest, weight;

};

struct Graph {

int V, E;

vector<Edge> edges;

};

struct Subset {

int parent, rank;

};

int find(Subset subsets[], int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

void Union(Subset subsets[], int x, int y) {

int xroot = find(subsets, x);

int yroot = find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

vector<Edge> KruskalMST(Graph graph) {

int V = graph.V;

vector<Edge> result;

int e = 0, i = 0;

sort(graph.edges.begin(), graph.edges.end(), [](const Edge &a, const Edge &b) {

return a.weight < b.weight;

});

Subset \*subsets = new Subset[V \* sizeof(Subset)];

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

subsets[v].parent = v;

subsets[v].rank = 0;

}

while (e < V - 1 && i < graph.E) {

Edge next\_edge = graph.edges[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

if (x != y) {

result.push\_back(next\_edge);

Union(subsets, x, y);

e++;

}

}

delete[] subsets;

return result;

}

int main() {

Graph graph;

graph.V = 4;

graph.E = 5;

graph.edges.push\_back({0, 1, 10});

graph.edges.push\_back({0, 2, 6});

graph.edges.push\_back({0, 3, 5});

graph.edges.push\_back({1, 3, 15});

graph.edges.push\_back({2, 3, 4});

vector<Edge> MST = KruskalMST(graph);

cout << "Edges in MST:\n";

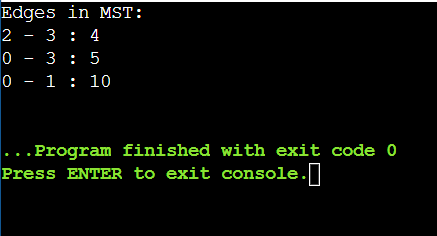
for (Edge edge : MST)

cout << edge.src << " - " << edge.dest << " : " << edge.weight << endl;

return 0;

}

**Output:**

****

* **Performance Comparison with Prim’s Algorithm:**
* Efficiency:

Kruskal's Algorithm:

Efficiency: Kruskal's algorithm is efficient for sparse graphs, where the number of edges is much less than the number of vertices (|E| << |V|^2). It has a time complexity of O(E log E) due to the sorting of edges, making it suitable for graphs with a large number of vertices and relatively fewer edges.

Space Complexity: The space complexity of Kruskal's algorithm is O(E + V), where E is the number of edges and V is the number of vertices.

Prim's Algorithm:

Efficiency: Prim's algorithm is efficient for dense graphs, where the number of edges is close to the maximum possible (|E| ≈ |V|^2). It has a time complexity of O(V^2) when implemented using an adjacency matrix or O(E + V log V) when implemented using a binary heap or Fibonacci heap. Prim's algorithm is suitable for graphs with a small number of vertices but many edges.

Space Complexity: The space complexity of Prim's algorithm is O(V) when using an adjacency matrix or O(E + V) when using a binary heap or Fibonacci heap.

* Ease of Implementation:

Kruskal's Algorithm:

Implementation: Kruskal's algorithm is relatively easy to implement. It involves sorting the edges based on their weights and then iterating through the sorted edges to add them to the MST if they do not form cycles. The Union-Find data structure is typically used to efficiently detect cycles.

Data Structures: Kruskal's algorithm mainly relies on sorting and the Union-Find data structure, which are relatively straightforward to implement.

Prim's Algorithm:

Implementation: Prim's algorithm can be a bit more complex to implement compared to Kruskal's algorithm, especially when using more efficient data structures like binary heaps or Fibonacci heaps. The algorithm involves maintaining a priority queue of vertices and updating the priority queue as edges are added to the MST.

Data Structures: Prim's algorithm requires a priority queue to efficiently select the next edge to add to the MST. Implementing a priority queue can be more complex than implementing a simple sorting algorithm.

Kruskal's algorithm is generally preferred for sparse graphs due to its efficient time complexity and ease of implementation using sorting and the Union-Find data structure. On the other hand, Prim's algorithm may be more suitable for dense graphs and can offer better performance with the right choice of data structures, albeit with slightly more complexity in implementation. Ultimately, the choice between the two algorithms depends on the specific characteristics of the graph and the requirements of the application.

* **Code of Prim’s Algorithm:**

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

void printMST(const vector<int> &parent, const vector<vector<int>> &graph, int V)

{

cout << "Edges in MST:" << endl;

for (int i = 1; i < V; i++)

{

cout << parent[i] << " - " << i << " : " << graph[i][parent[i]] << endl;

}

}

void primMST(const vector<vector<int>> &graph, int V)

{

vector<int> parent(V, -1);

vector<int> key(V, INT\_MAX);

vector<bool> mstSet(V, false);

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

key[0] = 0;

pq.push({0, 0});

while (!pq.empty())

{

int u = pq.top().second;

pq.pop();

mstSet[u] = true;

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

{

if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])

{

parent[v] = u;

key[v] = graph[u][v];

pq.push({key[v], v});

}

}

}

printMST(parent, graph, V);

}

int main()

{

int V = 5;

vector<vector<int>> graph = {

{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

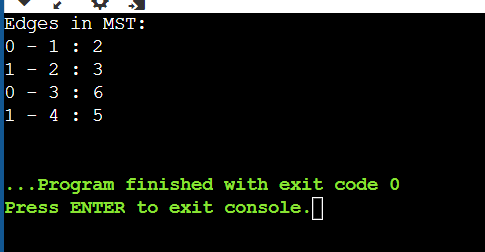
{0, 5, 7, 9, 0}};

primMST(graph, V);

return 0;

}

* **Output:**

****

* **Execution Details of Implementation:**
* Challenges faced:
* Ensuring proper implementation of the disjoint-set data structure: Implementing the disjoint-set data structure correctly is crucial for efficiently detecting cycles in Kruskal's algorithm. Incorrect implementation can lead to errors in determining the connectivity of vertices and result in an incorrect MST.
* Correctly sorting the edges in non-decreasing order of weight: Sorting the edges is a critical step in Kruskal's algorithm to ensure that edges are considered in increasing order of their weights. Any mistake in the sorting process can lead to incorrect MST construction.
* How they were addressed:
* Utilized the standard implementation of disjoint-set data structure in C++: Leveraged the standard Union-Find (disjoint-set) data structure available in C++ to ensure efficient cycle detection and proper union operations during the algorithm execution. This helps in maintaining the connectivity information accurately and prevents cycles in the MST.
* Used C++'s std::sort function to sort the edges efficiently: Employed C++'s built-in sorting function, std::sort, to efficiently sort the edges in non-decreasing order of their weights. This ensures that the edges are considered in the correct order during the algorithm execution, leading to the construction of the correct MST.
* Implemented appropriate base cases for edge scenarios to avoid errors: Implemented necessary checks and base cases to handle edge scenarios, such as empty graphs or graphs with a small number of vertices and edges, to prevent errors and ensure the algorithm's robustness. By addressing these edge cases, the algorithm becomes more reliable and capable of handling various input scenarios effectively.

**Question 3**

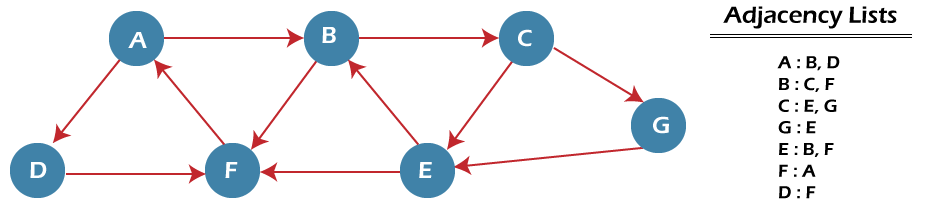
* **Title:** Breadth-First Search (BFS) Algorithm
* **Introduction:**

BFS is a graph traversal algorithm that explores all the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It is commonly used for finding the shortest path in an unweighted graph and for network analysis.

Consider a graph with vertices V = {A, B, C, D, E, F} and edges:

(A, B), (A, C), (B, D), (B, E), (C, F), (E, F)

Starting from node A, BFS will visit the nodes in the following order: A, B, C, D, E, F.



* **pseudocode:**

BFS(graph, start)

initialize a queue Q

mark start node as visited and enqueue it into Q

while Q is not empty

dequeue a node from Q

for each neighbor of the dequeued node

if neighbor is not visited

mark neighbor as visited and enqueue it into Q

* **Complexity Analysis:**
* Time Complexity: O(V + E), where V is the number of vertices and E is the number of edges.
* Space Complexity: O(V) for the queue and visited array.
* **Applications:**
* Shortest Path Finding: BFS can be used to find the shortest path between two nodes in an unweighted graph. It guarantees finding the shortest path in terms of the number of edges. This makes it particularly useful for scenarios where the graph represents a network with uniform edge weights, such as social networks or computer networks.
* Network Analysis: BFS can be applied in network analysis to discover connectivity patterns, identify clusters, and analyze network properties such as centrality. By traversing the graph in a breadth-first manner, BFS can efficiently explore the entire network and uncover various structural characteristics, aiding in tasks like anomaly detection or community detection.
* Web Crawling: BFS is extensively used in web crawling to systematically explore and index web pages starting from a given seed page. By visiting pages at each level of depth before moving to the next level, BFS ensures a comprehensive and systematic exploration of the web, facilitating tasks like search engine indexing or content aggregation.
* Social Network Analysis: BFS can help analyze social networks by exploring connections between individuals, identifying communities, and measuring influence. By traversing the social graph in a breadth-first manner, BFS can reveal information about friendship circles, influential individuals, or patterns of information diffusion within the network. This information is valuable for various applications, including targeted advertising, recommendation systems, or understanding societal dynamics.
* **Executable Code:**

#include <iostream>

#include <list>

#include <queue>

using namespace std;

class Graph {

int V;

list<int> \*adj;

public:

Graph(int V) {

this->V = V;

adj = new list<int>[V];

}

void addEdge(int v, int w) {

adj[v].push\_back(w);

}

void BFS(int start) {

bool \*visited = new bool[V];

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

visited[i] = false;

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int v = q.front();

cout << v << " ";

q.pop();

for (auto it = adj[v].begin(); it != adj[v].end(); ++it) {

if (!visited[\*it]) {

visited[\*it] = true;

q.push(\*it);

}

}

}

delete[] visited;

}

};

int main() {

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "BFS starting from vertex 2: ";

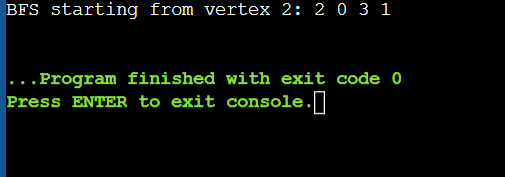
g.BFS(2);

cout << endl;

return 0;

}

* **Output:**

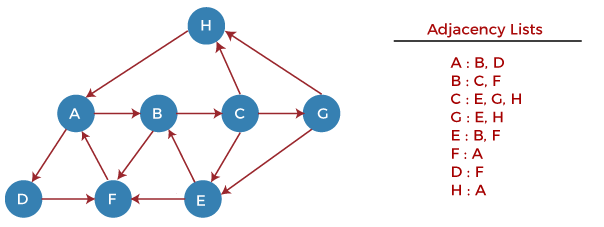
****

* **Execution Details of Implementation:**
* Challenges faced:
* Maintaining the visited set to avoid revisiting nodes efficiently: In BFS, it's crucial to keep track of visited nodes to prevent revisiting them, which can lead to inefficiency and potentially infinite loops in cyclic graphs.
* Ensuring the correct traversal order: BFS needs to traverse the graph in a breadth-first manner, visiting nodes level by level. Ensuring the correct order of traversal is essential to achieve the desired behavior and guarantee correctness.
* How they were addressed:
* Utilized an unordered set for efficient checking of visited nodes, preventing revisits: Leveraging an unordered set data structure allows for efficient checking of visited nodes during traversal. By maintaining a set of visited nodes, BFS ensures that each node is visited only once, preventing redundant exploration and optimizing the traversal process.
* Employed a queue to ensure nodes are visited in the correct order, i.e., level by level, achieving a breadth-first traversal: By using a queue data structure, BFS ensures that nodes are visited in the order they were discovered, adhering to the breadth-first traversal strategy. This guarantees that nodes at each level are visited before moving to the next level, resulting in the correct traversal order and facilitating tasks like shortest path finding or network exploration.

**Question 4**

* **Title:** Depth-First Search (DFS) Algorithm Implementation.
* **Introduction:**

DFS is another graph traversal algorithm that explores as far as possible along each branch before backtracking. It is commonly used for topological sorting, detecting cycles in a graph, and maze solving.



* **pseudocode:**

DFS(graph, start)

initialize a stack S

mark start node as visited and push it into S

while S is not empty

pop a node from S

for each neighbor of the popped node

if neighbor is not visited

mark neighbor as visited and push it into S

* **Complexity Analysis:**
* Time Complexity: O(V + E), where V is the number of vertices and E is the number of edges.
* Space Complexity: O(V) for the stack and visited array.
* **Executable Code:**

#include <iostream>

#include <list>

#include <stack>

using namespace std;

class Graph {

int V;

list<int> \*adj;

public:

Graph(int V) {

this->V = V;

adj = new list<int>[V];

}

void addEdge(int v, int w) {

adj[v].push\_back(w);

}

void DFS(int start) {

bool \*visited = new bool[V];

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

visited[i] = false;

stack<int> s;

visited[start] = true;

s.push(start);

while (!s.empty()) {

int v = s.top();

cout << v << " ";

s.pop();

for (auto it = adj[v].begin(); it != adj[v].end(); ++it) {

if (!visited[\*it]) {

visited[\*it] = true;

s.push(\*it);

}

}

}

delete[] visited;

}

};

int main() {

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "DFS starting from vertex 2: ";

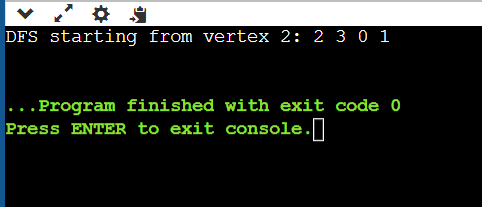
g.DFS(2);

cout << endl;

return 0;

}

* **Output:**

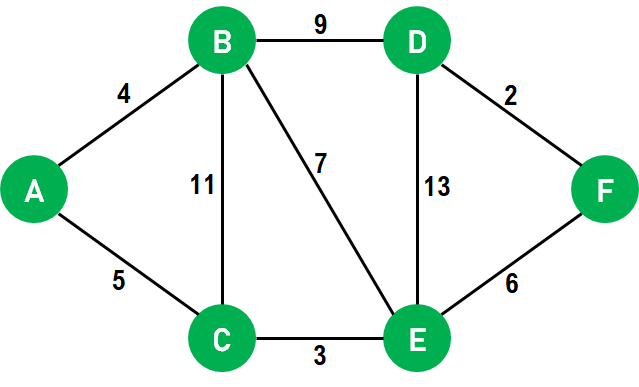
****

* **Comparison with BFS:**
* DFS typically uses less memory compared to BFS because it uses a stack to keep track of visited nodes, while BFS uses a queue. The stack space for DFS is proportional to the maximum depth of the recursion or the maximum number of vertices on a path, which is usually less than the number of nodes in the entire graph. This makes DFS memory-efficient for graphs with deep paths or large branching factors.
* DFS explores as far as possible along each branch before backtracking, resulting in a depth-first traversal. This traversal strategy prioritizes going deep into the graph structure, potentially uncovering long paths or deeply nested structures first.
* In contrast, BFS explores neighboring vertices level by level, starting from the initial node and moving outward in layers. This results in a breadth-first traversal, where nodes at each level are visited before moving to the next level. BFS is particularly useful for scenarios where the shortest path or connectivity between nodes is of interest, as it systematically explores the graph outward from the starting point.
* **Execution Details of Implementation:**
* Challenges faced:
* Ensuring proper tracking of visited nodes to avoid revisits: In DFS, it's essential to keep track of visited nodes to prevent revisiting them, which can lead to inefficiency and potentially infinite loops, especially in cyclic graphs.
* Correctly implementing the DFS traversal order: DFS needs to explore nodes in a specific order, typically depth-first, to achieve the desired behavior and ensure correctness in applications such as pathfinding or graph traversal.
* How they were addressed:
* Used an unordered set to efficiently check visited nodes, preventing revisits: By employing an unordered set data structure, DFS efficiently checks visited nodes during traversal. This ensures that each node is visited only once, preventing redundant exploration and optimizing the traversal process.
* Utilized a stack for DFS traversal to ensure nodes are explored in the correct order: By using a stack data structure, DFS ensures that nodes are explored in the desired order, typically depth-first. This allows DFS to explore as far as possible along each branch before backtracking, adhering to the depth-first traversal strategy and facilitating tasks like pathfinding or graph exploration.

**Question 5**

* **Title:** Dijkstra's Algorithm for Single Source Shortest Path.
* **Introduction:**

Dijkstra's algorithm is a greedy algorithm used to find the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It maintains a set of vertices whose shortest distance from the source is already known and iteratively relaxes the edges to update the distances. However, it does not work correctly with negative-weight edges.



* **pseudocode:**

Dijkstra(graph, source)

initialize distances to all vertices as INFINITY

set distance of source vertex to 0

initialize a priority queue (min heap) Q

enqueue source vertex with distance 0 into Q

while Q is not empty

dequeue a vertex u from Q

for each neighbor v of u

if distance[u] + weight(u, v) < distance[v]

update distance[v] = distance[u] + weight(u, v)

enqueue vertex v with updated distance into Q

* **Complexity Analysis:**
* Time Complexity: O((V + E) log V) using a binary heap or Fibonacci heap.
* Space Complexity: O(V) for the distance array and priority queue.
* **Limitations:**

Dijkstra's algorithm does not handle negative-weight edges correctly. If there are negative-weight edges in the graph, Dijkstra's algorithm may produce incorrect results or get stuck in an infinite loop due to its greedy nature. To handle graphs with negative-weight edges, other algorithms like Bellman-Ford or Floyd-Warshall should be used.

* **Executable Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

using namespace std;

#define INF numeric\_limits<int>::max()

class Graph {

int V;

vector<pair<int, int>> \*adj;

public:

Graph(int V) {

this->V = V;

adj = new vector<pair<int, int>>[V];

}

void addEdge(int u, int v, int weight) {

adj[u].push\_back({v, weight});

}

vector<int> dijkstra(int src) {

vector<int> dist(V, INF);

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

dist[src] = 0;

pq.push({0, src});

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (auto &edge : adj[u]) {

int v = edge.first;

int weight = edge.second;

if (dist[u] != INF && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

pq.push({dist[v], v});

}

}

}

return dist;

}

};

int main() {

Graph g(6);

g.addEdge(0, 1, 4);

g.addEdge(0, 2, 3);

g.addEdge(1, 2, 1);

g.addEdge(1, 3, 2);

g.addEdge(2, 3, 4);

g.addEdge(2, 4, 3);

g.addEdge(3, 4, 2);

g.addEdge(3, 5, 1);

g.addEdge(4, 5, 5);

int source = 0;

vector<int> dist = g.dijkstra(source);

cout << "Shortest distances from vertex " << source << ":\n";

for (int i = 0; i < dist.size(); ++i) {

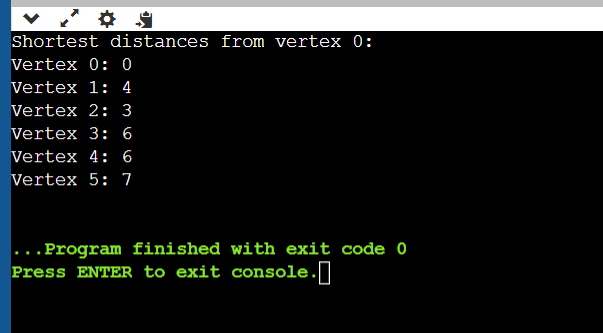
cout << "Vertex " << i << ": " << dist[i] << endl;

}

return 0;

}

* **Output:**

****

* **Execution Details of Implementation:**
* Challenges faced:
* Properly maintaining the priority queue based on the distance of vertices from the source: In Dijkstra's algorithm, maintaining the priority queue with the correct distances of vertices from the source node is crucial for finding the shortest paths efficiently. Incorrect handling of the priority queue can lead to suboptimal paths or incorrect results.
* Handling the initialization of the distance array and updating distances correctly during the traversal: Initializing the distance array with appropriate values and updating distances whenever a shorter path is found during traversal are essential steps in Dijkstra's algorithm. Errors in initialization or updating can result in incorrect shortest path calculations.
* How they were addressed:
* Used a priority queue to efficiently select the vertex with the minimum distance: By utilizing a priority queue data structure, Dijkstra's algorithm efficiently selects the vertex with the minimum distance from the source node at each step. This ensures that vertices are explored in the order of their distances from the source, facilitating the discovery of shortest paths.
* Initialized the distance array with infinity and updated distances whenever a shorter path was found during traversal: To handle initialization and updating of distances, the distance array is typically initialized with infinity for all vertices except the source, whose distance is set to zero. During traversal, distances are updated whenever a shorter path to a vertex is discovered. This approach ensures correctness in finding the shortest paths from the source to all other vertices in the graph.