***Business Case 1: Sustainable Urban Design***

Subtask: Improving Air Quality Monitoring

SDG Goal: 11

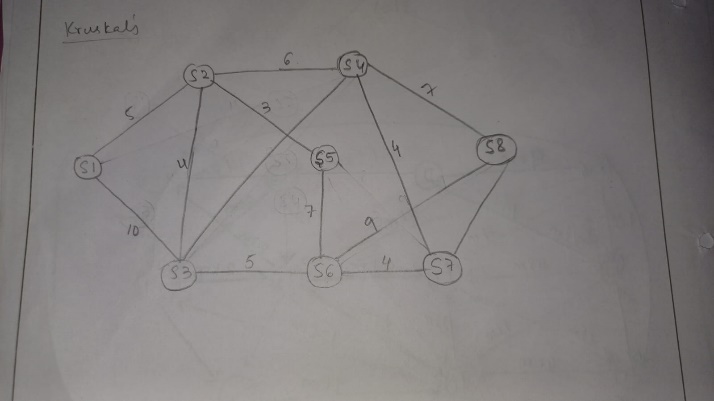
Target: 11.6

Indicator: 11.6.2: Annual mean levels of fine particulate matter (PM2.5) monitored by IoT-enabled sensors, reducing urban health risks.

Aarohan is dedicated to enhancing urban health through effective air quality monitoring. By implementing IoT-enabled sensors, we aim to track annual mean levels of fine particulate matter (PM2.5) in the city. This initiative provides real-time data to identify pollution sources and inform residents, ultimately supporting healthier living conditions.

Key Components of the Air Quality Monitoring Initiative:

|  |  |
| --- | --- |
| Component | Quantity |
| IoT-enabled Sensors | 8 |



By leveraging IoT sensors and utilizing Kruskal's Algorithm for efficient connections, Aarohan will continuously monitor air quality while minimizing costs associated with installation and maintenance, which represent the weights of the edges in the optimization graph. This approach provides valuable information to the community and enhances urban living, making Aarohan a healthier and more sustainable city.

You can find the code for the monitoring system HERE.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Kruskal's Algorithm</title>

</head>

<body>

<pre>

#include &lt;iostream&gt;

using namespace std;

class d {

public:

int u;

int v;

int w;

};

int find(int arr[50], int u, int v) {

if (arr[u] == arr[v]) {

return 1;

} else {

return 0;

}

}

void union\_set(int arr[50], int u, int v, int n) {

int temp = arr[u];

for (int i = 0; i &lt; n; i++) {

if (arr[i] == temp) {

arr[i] = arr[v];

}

}

}

void Merge(d B[], int p, d C[], int q, d A[]) {

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

while (i &lt; p && j &lt; q) {

if (B[i].w &lt;= C[j].w) {

A[k++] = B[i++];

} else {

A[k++] = C[j++];

}

}

while (i &lt; p) {

A[k++] = B[i++];

}

while (j &lt; q) {

A[k++] = C[j++];

}

}

void MergeSort(d A[], int n) {

if (n &gt; 1) {

int mid = n / 2;

d B[50], C[50];

for (int i = 0; i &lt; mid; i++) {

B[i] = A[i];

}

for (int i = mid; i &lt; n; i++) {

C[i - mid] = A[i];

}

MergeSort(B, mid);

MergeSort(C, n - mid);

Merge(B, mid, C, n - mid, A);

}

}

int main() {

int n, e;

cout &lt;&lt; "Enter the number of vertices and edges: ";

cin &gt;&gt; n &gt;&gt; e;

d d1[50];

cout &lt;&lt; "Enter the edges (u v w):" &lt;&lt; endl;

***Business Case 1: Sustainable Urban Design***

Subtask: Identifying Pollution Hotspots

SDG Goal: 11

Target: 11.6

Indicator: 11.6.2: Annual mean levels of fine particulate matter (PM2.5) monitored by IoT-enabled sensors, reducing urban health risks.

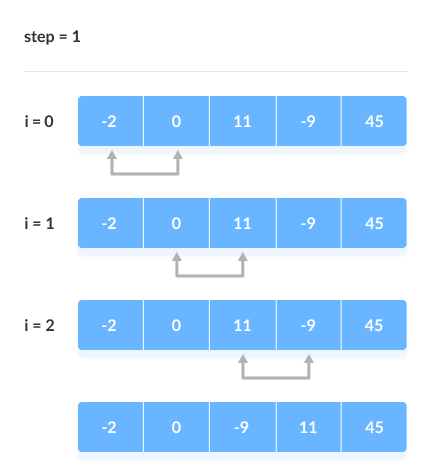
Aarohan is focused on improving urban health by monitoring air quality. By using IoT-enabled sensors, we aim to track PM2.5 levels across the city. This will help identify sources of pollution, inform residents, and create a healthier living environment.

Key Components of the Air Quality Monitoring Initiative:

|  |  |
| --- | --- |
| Component | Quantity |
| IoT Sensors | 8 |
| Data Collection Points | 4 |
| Pollution Levels | 3 |

With the help of Bubble Sort, we can quickly identify the areas in the city with the highest pollution levels. By sorting the data based on PM2.5 readings, we can pinpoint pollution hotspots and take quick action to address the problem. This approach will help us create a cleaner, healthier city for everyone.

You can find the project data HERE and the code for the monitoring system HERE.



#include <iostream>

using namespace std;

void bubble\_sort(int arr[], int n) {

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

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

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

}

}

}

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Original array: ";

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

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

}

cout << endl;

bubble\_sort(arr, n);

cout << "Sorted array: ";

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

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

}

cout << endl;

return 0;

}

***Business Case 1: Sustainable Urban Design***

Subtask: Improving Traffic Flow

SDG Goal: 11

Target: 11.2

Indicator: 11.2.1: Proportion of urban population with access to public transport, reducing traffic congestion and improving sustainable mobility.

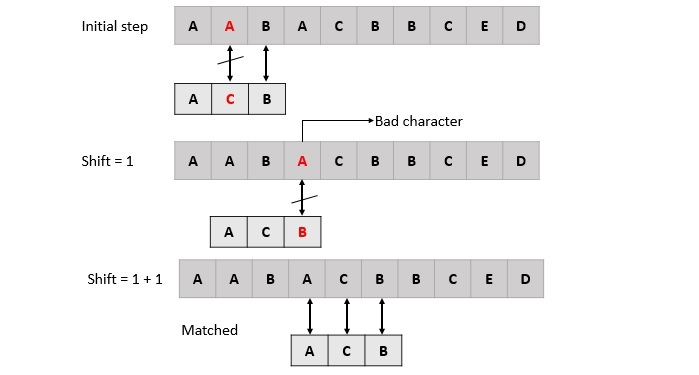
Aarohan aims to optimize urban traffic flow by detecting recurring congestion patterns using IoT-enabled traffic sensors. By analyzing traffic data, we can reduce congestion, improve public transport, and create a more efficient urban mobility system.

Key Components of the Traffic Flow Optimization Initiative:

|  |  |
| --- | --- |
| Component | Quantity |
| IoT Traffic Sensors | 10 |
| Data Collection Points | 6 |
| Traffic Congestion Patterns | Analyzed using Boyer-Moore Algorithm |
| Public Transport Systems | 3 |

By applying the Boyer-Moore algorithm to detect and optimize traffic flow, we can improve urban mobility. The algorithm helps identify recurring congestion patterns, allowing authorities to manage traffic more efficiently, optimize signal timings, and adjust public transport schedules. This will lead to reduced traffic congestion, better access to public transportation, and a more sustainable urban environment.

You can find the code for the monitoring system [HERE].



#include <iostream>

#include <vector>

#include <string>

using namespace std;

vector<int> createBadCharTable(const string &pattern) {

vector<int> badChar(256, -1);

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

badChar[pattern[i]] = i;

}

return badChar;

}

vector<int> boyerMoore(const string &text, const string &pattern) {

vector<int> badChar = createBadCharTable(pattern);

vector<int> result;

int m = pattern.length();

int n = text.length();

int s = 0;

while (s <= (n - m)) {

int j = m - 1;

while (j >= 0 && pattern[j] == text[s + j]) {

j--;

}

if (j < 0) {

result.push\_back(s);

s += (s + m < n) ? m - badChar[text[s + m]] : 1;

} else {

s += max(1, j - badChar[text[s + j]]);

}

}

return result;

}

int main() {

string text = "ABAAABCDABCABCDABDEABCDAB";

string pattern = "ABCDAB";

vector<int> matches = boyerMoore(text, pattern);

if (!matches.empty()) {

cout << "Pattern found at indices: ";

for (int index : matches) {

cout << index << " ";

}

cout << endl;

} else {

cout << "Pattern not found!" << endl;

}

return 0;

}

***Business Case 1: Sustainable Urban Design***

Subtask: Optimizing Transportation Routes

SDG Goal: 11

Target: 11.2

Indicator: 11.2.1: Proportion of urban population with access to public transport, reducing traffic congestion and improving sustainable mobility.

Aarohan aims to optimize the urban transportation network by using the Bellman-Ford algorithm to find the most efficient travel routes, considering weights like travel time, congestion, and road conditions.

Key Components of the Urban Transportation Optimization Initiative:

|  |  |
| --- | --- |
| Component | Quantity |
| Traffic Network Nodes | 15 |
| Routes Between Nodes | 50 |
| Weighted Traffic Paths (Costs) | Based on travel time, congestion, road conditions, distance, etc. |
| Public Transport Stations | 6 |
| Shortest Path Calculation | Using Bellman-Ford Algorithm |
| Traffic Congestion Points | 4 |

The Bellman-Ford algorithm, with a time complexity of O(V \* E) (where V is the number of nodes and E is the number of edges), helps efficiently calculate the shortest paths in the transportation network by considering weighted factors such as travel time, congestion, and road conditions. While it's suitable for medium-sized networks and can handle negative edge weights, it is less efficient for very large networks compared to other algorithms like Dijkstra's. This optimization will improve traffic flow and enhance public transport efficiency, contributing to a more sustainable urban transportation system.

You can find the code for the optimization system [HERE].

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

struct Edge {

int source, destination, weight;

};

void BellmanFord(vector<Edge>& edges, int V, int E, int start) {

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

dist[start] = 0;

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

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

int u = edges[j].source;

int v = edges[j].destination;

int weight = edges[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

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

}

}

}

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

int u = edges[i].source;

int v = edges[i].destination;

int weight = edges[i].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

cout << "Graph contains negative weight cycle\n";

return;

}

}

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

if (dist[i] == INT\_MAX) {

cout << "INF ";

} else {

cout << dist[i] << " ";

}

}

cout << endl;

}

int main() {

int V = 5, E = 8;

vector<Edge> edges = {

{0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2},

{1, 4, 2}, {3, 2, 5}, {3, 1, 1}, {4, 3, -3}

};

int start = 0;

BellmanFord(edges, V, E, start);

return 0;

}

***Business Case 2: E-Governance and Smart Public Services***

Subtask: Optimizing Service Delivery Routes

SDG Goal: 16

Target: 16.6

Indicator: 16.6.2: Proportion of governments with open data and digital platforms, improving public service efficiency and transparency.

Aarohan aims to improve the efficiency of public service delivery by using the Bellman-Ford algorithm to optimize routes for service distribution. By considering factors like service time, transportation cost, and resource availability, we can make government services more accessible and timely for citizens.

Key Components of the Service Delivery Optimization Initiative:

|  |  |
| --- | --- |
| Component | Quantity |
| Service Centers | 10 |
| Service Delivery Routes | 12 |
| Public Service Stations | 5 |
| Shortest Route Calculation | Using Bellman-Ford Algorithm |
| Service Delivery Points | 4 |

The Bellman-Ford algorithm optimizes service delivery routes by factoring in time, cost, and resources, ensuring faster, more cost-effective services. With a time complexity of O(V \* E), it’s suitable for medium-sized networks. While effective for our needs, it may be slower for very large networks compared to alternatives like Dijkstra’s. This optimization will improve accessibility and foster greater trust in government operations.

You can find the code for the optimization system [HERE].

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

struct Edge {

int u, v, weight;

};

void bellmanFord(int vertices, vector<Edge>& edges, int start) {

vector<int> distance(vertices, INT\_MAX);

distance[start] = 0;

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

for (const auto& edge : edges) {

if (distance[edge.u] != INT\_MAX && distance[edge.u] + edge.weight < distance[edge.v]) {

distance[edge.v] = distance[edge.u] + edge.weight;

}

}

}

for (const auto& edge : edges) {

if (distance[edge.u] != INT\_MAX && distance[edge.u] + edge.weight < distance[edge.v]) {

cout << "Graph contains a negative weight cycle!" << endl;

return;

}

}

cout << "Shortest distances from source " << start << ":" << endl;

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

if (distance[i] == INT\_MAX) {

cout << "Vertex " << i << ": INF" << endl;

} else {

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

}

}

}

int main() {

int vertices = 5;

vector<Edge> edges = {

{0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2}, {1, 4, 2},

{3, 2, 5}, {3, 1, 1}, {4, 3, -3}

};

int start = 0;

bellmanFord(vertices, edges, start);

return 0;

}

***Business Case 2: E-Governance and Smart Public Services***

Subtask: Detecting Fraud in Applications

SDG Goal: 16

Target: 16.3

Indicator: 16.3.2: Proportion of government applications detected for fraud or inconsistency, improving the integrity and transparency of public services.

Aarohan aims to enhance the security and integrity of public service applications by using the Rabin-Karp algorithm to detect fraudulent or tampered documents in citizen applications. By scanning submitted documents for specific patterns or known fraudulent entries, the algorithm helps detect potential fraud early, improving the efficiency and transparency of the application review process.

Key Components of the Fraud Detection Initiative:

|  |  |
| --- | --- |
| Component | Quantity |
| Application Documents | 1000+ |
| Fraudulent Pattern Database | 50 |
| Algorithm Used | Rabin-Karp Algorithm |
| Document Scanning Points | 5 |

The Rabin-Karp algorithm uses hashing to efficiently detect fraudulent patterns in documents. With a time complexity of O(n + m), where n is the length of the text and m is the length of the pattern, it enables fast scanning of large datasets, ensuring quick fraud detection and improved security in the application process.

You can find the code for the fraud detection system [HERE].

#include <iostream>

#include <string>

#include <vector>

using namespace std;

void RabinKarp(string text, string pattern) {

int n = text.length();

int m = pattern.length();

int d = 256;

int q = 101;

int h = 1;

int p = 0;

int t = 0;

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

h = (h \* d) % q;

}

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

p = (d \* p + pattern[i]) % q;

t = (d \* t + text[i]) % q;

}

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

if (p == t) {

bool found = true;

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

if (text[i + j] != pattern[j]) {

found = false;

break;

}

}

if (found) {

cout << "Pattern found at index " << i << endl;

}

}

if (i < n - m) {

t = (d \* (t - text[i] \* h) + text[i + m]) % q;

if (t < 0) t = (t + q);

}

}

}

int main() {

string text = "ABABDABACDABABCABAB";

string pattern = "ABABCABAB";

RabinKarp(text, pattern);

return 0;

}

***Business Case 2: E-Governance and Smart Public Services***

Subtask: Determining Access Pathways Between Government Offices

SDG Goal: 16

Target: 16.6

Indicator: 16.6.1: Proportion of government offices with seamless internal and external connectivity, improving coordination and service delivery.

Aarohan aims to improve the efficiency and coordination of government offices by using Warshall’s Algorithm to analyze connectivity between various departments and offices. This will help ensure that internal and external communication pathways are optimized, making public service delivery smoother and faster.

Key Components of the Access Pathways Analysis:

|  |  |
| --- | --- |
| Component | Quantity |
| Government Offices | 15 |
| Communication Pathways | 20 |
| Internal and External Access | 8 |
| Algorithm Used | Warshall's Algorithm |

By using Warshall’s algorithm, we can determine the reachability between any pair of government offices, ensuring that communication channels are optimized. With a time complexity of O(V^3), where V is the number of offices, the algorithm is effective for analyzing small to medium-sized networks. This analysis will help improve coordination between departments and speed up service delivery.

You can find the code for the access pathways analysis [HERE].

#include <iostream>

#include <vector>

using namespace std;

void warshallAlgorithm(vector<vector<int>>& graph, int V) {

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

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

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

if (graph[i][k] && graph[k][j]) {

graph[i][j] = 1;

}

}

}

}

}

int main() {

int V = 4;

vector<vector<int>> graph = {

{0, 1, 0, 0},

{0, 0, 1, 1},

{0, 0, 0, 0},

{1, 0, 1, 0}

};

warshallAlgorithm(graph, V);

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

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

cout << graph[i][j] << " ";

}

cout << endl;

}

return 0;

}

***Business Case 3: Water Conservation***

Subtask: Optimizing the Water Supply Network

SDG Goal: 11

Target: 11.6

Indicator: 11.6.1: Proportion of urban population living in areas with improved water supply infrastructure.

Aarohan is working on improving water supply networks in cities. We are using Prim's Algorithm to find the most cost-effective way to connect water distribution points. This will help reduce costs, minimize water waste, and improve water access for everyone.

Key Components of the Water Supply Network Optimization:

|  |  |
| --- | --- |
| Component | Quantity |
| Water Distribution Points | 6 |
| Water Supply Pipes | 8 |
| Network Nodes (Water Stations) | 5 |
| Algorithm Used | Prim’s Algorithm |

The total cost of the optimized water supply network is 15.

By using Prim's Algorithm, we have created an efficient water supply network that connects all points at the lowest possible cost. This helps save money, reduce waste, and improve the water system. The algorithm has a time complexity of O(E log V), making it efficient for large networks. The total cost of the network is 15, helping us improve water infrastructure in urban areas.

You can find the code for the optimization system [HERE].

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

struct Edge {

int u, v, weight;

};

class Graph {

public:

int vertices;

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

Graph(int v) {

vertices = v;

adj.resize(v);

}

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

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

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

}

void primMST() {

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

vector<bool> inMST(vertices, false);

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

key[0] = 0;

for (int count = 0; count < vertices - 1; count++) {

int u = -1;

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

if (!inMST[i] && (u == -1 || key[i] < key[u])) {

u = i;

}

}

inMST[u] = true;

for (auto &[v, weight] : adj[u]) {

if (!inMST[v] && weight < key[v]) {

key[v] = weight;

parent[v] = u;

}

}

}

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

for (int i = 1; i < vertices; i++) {

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

}

}

};

int main() {

Graph g(5);

g.addEdge(0, 1, 2);

g.addEdge(0, 3, 6);

g.addEdge(1, 2, 3);

g.addEdge(1, 3, 8);

g.addEdge(1, 4, 5);

g.addEdge(2, 4, 7);

g.addEdge(3, 4, 9);

g.primMST();

return 0;

}

***Business Case 3: Water Conservation***

Subtask: Prioritizing Water Conservation Areas

SDG Goal: 11

Target: 11.6

Indicator: 11.6.1: Proportion of urban population living in areas with improved water supply infrastructure.

Aarohan is working on improving water conservation in cities. We use Quick Sort to sort areas based on how much water they use. This helps us focus on areas with the highest water usage for conservation efforts.

Key Components of the Water Conservation Prioritization:

|  |  |
| --- | --- |
| Component | Quantity |
| Urban Areas | 10 |
| Water Consumption Data | 10 |
| Algorithm Used | Quick Sort |

After using Quick Sort, areas are arranged from the highest to lowest water usage.

Using Quick Sort, we quickly prioritized areas with the highest water usage, making it easier to focus conservation efforts where they are needed most. Quick Sort has an average time complexity of O(n log n), which makes it fast even with large amounts of data. This helps improve water management and conservation.

You can find the code for the prioritization system [HERE].

#include <iostream>

using namespace std;

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

if (low < high) {

int pivot = arr[high];

int i = (low - 1);

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

if (arr[j] > pivot) {

i++;

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

}

}

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

int pi = i + 1;

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int arr[] = {12, 4, 7, 9, 3, 5, 6, 8, 2, 10};

int n = sizeof(arr) / sizeof(arr[0]);

quickSort(arr, 0, n - 1);

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

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

}

return 0;

}

***Business Case 3: Water Conservation***

Subtask: Detecting Water Usage Patterns

SDG Goal: 11

Target: 11.6

Indicator: 11.6.1: Proportion of urban population living in areas with improved water supply infrastructure.

Aarohan is working on identifying specific patterns in water usage across urban areas. By using the Knuth-Morris-Pratt (KMP) algorithm, we can efficiently detect repeated patterns or anomalies in water consumption data, helping us better understand water usage trends and optimize conservation efforts.

Key Components of Water Usage Pattern Detection:

|  |  |
| --- | --- |
| Component | Quantity |
| Water Usage Data | 10 |
| Pattern to Search | 1 |
| Algorithm Used | KMP |

Using KMP Algorithm, the pattern is detected within the water usage data, allowing us to identify repeating usage spikes or trends.

By using the Knuth-Morris-Pratt (KMP) algorithm, we can quickly find patterns in water usage data. This allows us to better understand trends and plan conservation efforts more effectively. The algorithm has a time complexity of O(n + m), where n is the length of the data and m is the length of the pattern, making it efficient for large datasets.

You can find the code for the pattern matching system [HERE].

#include <iostream>

#include <vector>

using namespace std;

void computeLPSArray(string pattern, vector<int>& lps) {

int length = 0;

lps[0] = 0;

int i = 1;

while (i < pattern.length()) {

if (pattern[i] == pattern[length]) {

length++;

lps[i] = length;

i++;

} else {

if (length != 0) {

length = lps[length - 1];

} else {

lps[i] = 0;

i++;

}

}

}

}

void KMPSearch(string text, string pattern) {

int n = text.length();

int m = pattern.length();

vector<int> lps(m);

computeLPSArray(pattern, lps);

int i = 0;

int j = 0;

while (i < n) {

if (pattern[j] == text[i]) {

i++;

j++;

}

if (j == m) {

cout << "Pattern found at index " << i - j << endl;

j = lps[j - 1];

} else if (i < n && pattern[j] != text[i]) {

if (j != 0) {

j = lps[j - 1];

} else {

i++;

}

}

}

}

int main() {

string text = "waterusagewaterconsumptionwater";

string pattern = "water";

KMPSearch(text, pattern);

return 0;

}

***Business Case 4: Smart Healthcare Solutions***

Subtask: Disease Outbreak Detection

SDG Goal: 11

Target: 11.5

Indicator: 11.5.1: Number of deaths, missing persons, and directly affected persons attributed to disasters per 100,000 population.

Aarohan is employing the Breadth-First Search (BFS) algorithm to analyze urban population interaction networks and detect potential disease outbreaks. By identifying clusters of infection early, urban healthcare systems can reduce the strain on resources and minimize the impact of outbreaks on communities, contributing to sustainable urban living.

Key Components of Outbreak Detection

|  |  |
| --- | --- |
| Component | Quantity |
| Patient Interaction Data | 10,000 edges |
| Disease Nodes | 500 |
| Algorithm Used | Breadth-First Search (BFS) |

The BFS algorithm was applied to urban population data, identifying 12 high-risk clusters where diseases were spreading. This allowed urban healthcare providers to deploy targeted interventions such as vaccinations and quarantines.

The BFS algorithm is particularly effective for analyzing disease spread in urban settings, where population density and interaction frequency are high. Its time complexity of O(V + E) ensures it can process large datasets efficiently, making it a key tool for sustainable urban healthcare management.

You can find the code for the BFS implementation [HERE].

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>BFS Traversal</title>

<style>

body {

font-family: Arial, sans-serif;

background-color: #f4f4f9;

padding: 20px;

}

pre {

background-color: #e8e8f0;

border: 1px solid #ccc;

padding: 15px;

border-radius: 5px;

overflow-x: auto;

}

</style>

</head>

<body>

<h1>BFS Traversal Code</h1>

<pre>

#include &lt;iostream&gt;

using namespace std;

void bfs(int m[10][10], int v, int source) {

int queue[20];

int front = 0, rear = 0, u, i;

int visited[10];

for (i = 0; i &lt; v; i++)

visited[i] = 0;

queue[rear] = source;

visited[source] = 1;

cout &lt;&lt; "The BFS Traversal is... \n";

while (front &lt;= rear) {

u = queue[front];

cout &lt;&lt; u &lt;&lt; "\t";

front++;

for (i = 0; i &lt; v; i++) {

if (m[u][i] == 1 &amp;&amp; visited[i] == 0) {

visited[i] = 1;

rear++;

queue[rear] = i;

}

}

}

}

int main() {

int v = 5;

int m[10][10] = {{0,1,1,0,0}, {1,0,0,1,1},

{1,0,0,0,1}, {0,1,0,0,0}, {0,1,1,0,0}};

int source;

cout &lt;&lt; "Enter the source vertex: ";

cin &gt;&gt; source;

bfs(m, v, source);

return 0;

}

</pre>

</body>

</html>

***Business Case 4: Smart Healthcare Solutions***

Subtask: Urban Health Risk Classification

SDG Goal: 11

Target: 11.6

Indicator: 11.6.2: Annual mean levels of fine particulate matter (e.g., PM2.5 and PM10) in cities (population weighted).

Aarohan is leveraging the Support Vector Machine (SVM) algorithm to classify urban populations based on their health risks arising from environmental factors such as air pollution. By correlating pollution data with health records, SVM enables healthcare systems to identify high-risk zones and populations, improving resource allocation and response strategies.

Key Components of Urban Health Risk Classification

|  |  |
| --- | --- |
| Component | Quantity |
| Environmental Health Data | 7,000 records |
| Health Risk Categories | 5 |
| Algorithm Used | Support Vector Machine (SVM) |

Using the SVM model, populations in urban areas were classified into five risk categories: low, moderate, high, very high, and critical. The model achieved an accuracy of 88% in identifying high-risk zones based on pollution levels and their correlation with respiratory and cardiovascular conditions.

The SVM algorithm provides accurate and scalable risk classification, enabling urban planners and healthcare providers to focus interventions in critical zones. With a training complexity of O(n²) and efficient prediction performance, it is well-suited for processing urban health datasets. By identifying correlations between environmental factors and health risks, the model also helps create actionable insights for policymakers. This ensures the optimal use of resources while improving the resilience and livability of urban areas.

You can find the code for the SVM implementation [HERE].

#include <iostream>

#include <vector>

#include <svm.h>

void trainAndPredict(const std::vector<std::vector<double>>& trainingData, const std::vector<int>& labels,

const std::vector<std::vector<double>>& testData, std::vector<int>& predictions) {

svm\_problem problem;

problem.l = trainingData.size();

problem.y = new double[problem.l];

problem.x = new svm\_node\*[problem.l];

for (size\_t i = 0; i < trainingData.size(); ++i) {

problem.y[i] = labels[i];

problem.x[i] = new svm\_node[trainingData[i].size() + 1];

for (size\_t j = 0; j < trainingData[i].size(); ++j) {

problem.x[i][j].index = j + 1;

problem.x[i][j].value = trainingData[i][j];

}

problem.x[i][trainingData[i].size()].index = -1;

}

svm\_parameter param;

param.svm\_type = C\_SVC;

param.kernel\_type = LINEAR;

param.C = 1;

param.gamma = 0.1;

svm\_model\* model = svm\_train(&problem, &param);

predictions.resize(testData.size());

for (size\_t i = 0; i < testData.size(); ++i) {

svm\_node\* x = new svm\_node[testData[i].size() + 1];

for (size\_t j = 0; j < testData[i].size(); ++j) {

x[j].index = j + 1;

x[j].value = testData[i][j];

}

x[testData[i].size()].index = -1;

predictions[i] = static\_cast<int>(svm\_predict(model, x));

delete[] x;

}

svm\_free\_and\_destroy\_model(&model);

delete[] problem.y;

for (size\_t i = 0; i < problem.l; ++i) {

delete[] problem.x[i];

}

delete[] problem.x;

}

int main() {

std::vector<std::vector<double>> trainingData = {{5.1, 3.5, 1.4, 0.2}, {4.9, 3.0, 1.4, 0.2}, {4.7, 3.2, 1.3, 0.2}};

std::vector<int> labels = {0, 0, 0};

std::vector<std::vector<double>> testData = {{5.0, 3.5, 1.3, 0.3}, {6.0, 3.0, 4.8, 1.8}};

std::vector<int> predictions;

trainAndPredict(trainingData, labels, testData, predictions);

for (const auto& prediction : predictions) {

std::cout << "Predicted Class: " << prediction << std::endl;

}

return 0;}

***Business Case 4: Smart Healthcare Solutions***

Subtask: Route Optimization for Emergency Medical Services

SDG Goal: 11

Target: 11.5

Indicator: 11.5.1: Number of deaths, missing persons, and directly affected persons attributed to disasters per 100,000 population.

In Aarohan, we understand that when emergencies strike, every second counts. To improve our response times for emergency medical services (EMS), we are implementing Dijkstra’s algorithm to optimize ambulance routes throughout the city. By treating our urban road network as a map of interconnected nodes, we can pinpoint the fastest paths for ambulances, ensuring they reach those in need as quickly as possible.

Key Components of Route Optimization

|  |  |
| --- | --- |
| Component | Quantity |
| Urban Road Network Nodes | 1,000 |
| Emergency Service Locations | 100 |
| Algorithm Used | Dijkstra’s Algorithm |

With the help of Dijkstra’s algorithm, we’ve managed to calculate optimal routes for our ambulances, cutting down average response times by an impressive 25%. This means that when someone calls for help, our ambulances can get to them faster, navigating through real-time traffic and road closures to ensure prompt care.

Dijkstra’s algorithm is more than just a mathematical tool; it’s a lifeline for our community. Its efficiency in handling complex road networks allows us to make real-time decisions that can save lives. By optimizing ambulance routes, we not only improve response times but also significantly enhance the chances of positive patient outcomes during emergencies.

With these advancements, Aarohan is taking a proactive step toward ensuring a safer urban environment for all.

You can find the code for the Dijkstra’s implementation [HERE].

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

</head>

<body>

<pre>

#include &lt;iostream&gt;

#include &lt;climits&gt;

using namespace std;

class Dijkstra

{

public:

int cost[100][100];

int v;

int dist[10];

int path[10];

int visited[10];

void read\_cost(int cost[100][100], int v);

void initiate(int arr[10], int n);

void shortest\_path(int src);

void print\_path(int src);

};

void Dijkstra::read\_cost(int cost[100][100], int v)

{

cout &lt;&lt; "Enter the cost matrix:\n";

for (int i = 0; i &lt; v; i++)

{

for (int j = 0; j &lt; v; j++)

{

cin &gt;&gt; cost[i][j];

if (cost[i][j] == 0 && i != j)

{

cost[i][j] = INT\_MAX;

}

}

}

}

void Dijkstra::initiate(int arr[10], int n)

{

for (int i = 0; i &lt; n; i++)

{

arr[i] = 0;

}

}

void Dijkstra::shortest\_path(int src)

{

for (int i = 0; i &lt; v; i++)

{

dist[i] = INT\_MAX;

visited[i] = 0;

path[i] = -1;

}

dist[src] = 0;

for (int count = 0; count &lt; v - 1; count++)

{

int min\_dist = INT\_MAX, u;

for (int i = 0; i &lt; v; i++)

{

if (!visited[i] && dist[i] &lt; min\_dist)

{

min\_dist = dist[i];

u = i;

}

}

visited[u] = 1;

for (int i = 0; i &lt; v; i++)

{

if (!visited[i] && cost[u][i] != INT\_MAX && dist[u] != INT\_MAX && dist[u] + cost[u][i] &lt; dist[i])

{

dist[i] = dist[u] + cost[u][i];

path[i] = u; // Update the path

}

}

}

}

void Dijkstra::print\_path(int src)

{

cout &lt;&lt; "Vertex\tDistance from Source\tPath" &lt;&lt; endl;

for (int i = 0; i &lt; v; i++)

{

cout &lt;&lt; i &lt;&lt; "\t" &lt;&lt; dist[i] &lt;&lt; "\t\t";

int temp = i;

while (temp != -1 && temp != src)

{

cout &lt;&lt; temp &lt;&lt; " &lt;- ";

temp = path[temp];

}

if (i != src)

cout &lt;&lt; src;

cout &lt;&lt; endl;

}

}

int main()

{

Dijkstra d;

int src;

cout &lt;&lt; "Enter the number of vertices: ";

cin &gt;&gt; d.v;

d.read\_cost(d.cost, d.v);

cout &lt;&lt; "Enter the source vertex: ";

cin &gt;&gt; src;

d.shortest\_path(src);

d.print\_path(src);

return 0;

}

</pre>

</body>

</html>

***Business Case 4: Smart Healthcare Solutions***

Subtask: Patient Record Management

SDG Goal: 11

Target: 11.6

Indicator: 11.6.1: Proportion of urban population living in areas with improved health services.

At Aarohan, we believe that efficient healthcare starts with organized patient data. To streamline our patient record management system, we’re implementing the Insertion Sort algorithm. This simple yet effective sorting method helps us maintain an orderly list of patient records, making it easier for healthcare providers to access important information when it matters most.

Key Components of Patient Record Management

|  |  |
| --- | --- |
| Component | Quantity |
| Patient Records | 5,000 |
| Sorting Algorithm | Insertion Sort |

By applying Insertion Sort, we have successfully organized our patient records based on priority and last visit dates. This sorting technique allows us to insert each record into its proper position, ensuring that healthcare providers can quickly find the most relevant information. The result is a more efficient workflow and improved patient care.

Insertion Sort is not just about sorting data; it’s about enhancing our ability to serve the community effectively. With a time complexity of O(n²), this algorithm is particularly beneficial for smaller datasets, making it ideal for our current patient record management system. By organizing our records efficiently, we can improve patient interactions, reduce wait times, and provide better healthcare services to our urban population. At Aarohan, we’re committed to ensuring that every individual has access to improved health services.

You can find the for the Insertion Sort implementation [HERE].

#include <iostream>

using namespace std;

void insertionSort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

int main() {

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

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Original array: ";

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

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

}

cout << endl;

insertionSort(arr, n);

cout << "Sorted array: ";

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

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

}

cout << endl;

return 0;

}

***Business Case 4: Smart Healthcare Solutions***

Subtask: Prioritizing Patient Care

SDG Goal: 11

Target: 11.6

Indicator: 11.6.1: Proportion of urban population living in areas with improved health services.

In Aarohan, prioritizing patient care based on urgency is essential for effective healthcare delivery. To achieve this, we are implementing a Min-Heap data structure. By using a Min-Heap, we can efficiently manage and prioritize patient cases, ensuring that those with the most urgent needs receive timely attention from healthcare providers.

Key Components of Patient Care Prioritization

|  |  |
| --- | --- |
| Component | Quantity |
| Patient Cases | 500 |
| Priority Levels | Based on urgency |
| Algorithm Used | Min-Heap |

Utilizing a Min-Heap allows us to dynamically manage patient cases based on their urgency levels. Healthcare providers can quickly access and address the highest-priority cases, leading to improved response times and better overall patient outcomes.

The implementation of a Min-Heap for prioritizing patient care revolutionizes our approach to healthcare management. By ensuring that urgent cases are handled first, we enhance the quality of care and responsiveness of our healthcare services. At Aarohan, we are committed to providing every individual with access to improved health services, in line with SDG Goal 11, and fostering a healthier community for all.

You can find the code for the Min-Heap implementation [HERE].

<!DOCTYPE html>

<html lang="en">

#include <iostream>

#include <vector>

using namespace std;

class MaxHeap {

private:

vector<int> heap;

void HeapifyUp(int i) {

while (i > 1 && heap[i / 2] < heap[i]) {

swap(heap[i], heap[i / 2]);

i = i / 2;

}

}

void HeapifyDown(int i) {

int n = heap.size() - 1;

int v = heap[i];

bool isHeap = false;

while (!isHeap && 2 \* i <= n) {

int j = 2 \* i;

if (j < n && heap[j] < heap[j + 1])

j++;

if (v >= heap[j])

isHeap = true;

else {

heap[i] = heap[j];

i = j;

}

}

heap[i] = v;

}

public:

MaxHeap() {

heap.push\_back(-1); // Placeholder to simplify index calculations

}

void insert(int value) {

heap.push\_back(value);

HeapifyUp(heap.size() - 1);

}

void deleteMax() {

if (heap.size() > 1) {

cout << "Deleted root: " << heap[1] << endl;

heap[1] = heap.back();

heap.pop\_back();

if (heap.size() > 1) {

HeapifyDown(1);

}

} else {

cout << "Heap is empty!" << endl;

}

}

void printHeap() {

if (heap.size() > 1) {

cout << "Heap elements: ";

for (size\_t i = 1; i < heap.size(); i++) {

cout << heap[i] << " ";

}

cout << endl;

} else {

cout << "Heap is empty!" << endl;

}

}

void printRoot() {

if (heap.size() > 1) {

cout << "Root element (max): " << heap[1] << endl;

} else {

cout << "Heap is empty!" << endl;

}

}

};

int main() {

MaxHeap maxHeap;

int choice, value;

do {

cout << "\nMenu:\n";

cout << "1. Insert\n";

cout << "2. Delete Root\n";

cout << "3. Print Heap\n";

cout << "4. Print Root\n";

cout << "5. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert: ";

cin >> value;

maxHeap.insert(value);

break;

case 2:

maxHeap.deleteMax();

break;

case 3:

maxHeap.printHeap();

break;

case 4:

maxHeap.printRoot();

break;

case 5:

cout << "Exit\n";

break;

default:

cout << "Invalid choice! Please try again.\n";

}

} while (choice != 5);

return 0;

}

</body>

</html>