**NATUREPULSE – AN ENVIRONMENT IMPACT ANALZER**

Submitted by

Hansa A S - 71762133015

Giridhara Vijay R A - 71762133013

Submitted to

Dr. N. Yamuna Devi

Dr. S. Chandia

Department of Decision and Computing Sciences



**DEPARTMENT OF COMPUTING**

**COIMBATORE INSTITUTE OF TECHNOLOGY**

**(Autonomous Institution affiliated to Anna University)**

**COIMBATORE – 641014**

**April 2025**

**ABSTRACT**

NaturePulse is a full-fledged web-based Environmental Impact Analyzer designed to visualize, compare, and analyze environmental metrics across various cities. The platform empowers user to make informed decisions based on real-time and historical data on air quality, water sources, soil conditions, and climate indicators. It provides an interactive dashboard, advanced algorithmic insights (Kruskals algorithm and Multstage Graph), and data export features. Users can explore single-city environmental summaries, compare multiple cities, or simulate infrastructure plans and policy decisions through integrated algorithmic modules. Built using HTML, CSS, and JavaScript on the frontend and Flask on the backend, the application uses structured environmental datasets stored in Hash table. Advanced algorithms such as Kruskal's MST and Multi-stage Graph models have been implemented to optimize pipeline networks and policy-making strategies, respectively. The system is not only educational but also practical for real-world environmental analysis.

**SUITABILITY OF APPLICATION TO IMPLEMENT DESIGN TECHNIQUES AND HASHING**

The **NATUREPULSE – AN ENVIRONMENTAL IMPACT ANALYZER** is highly suitable for implementing algorithm design techniques and hashing for the following reasons:

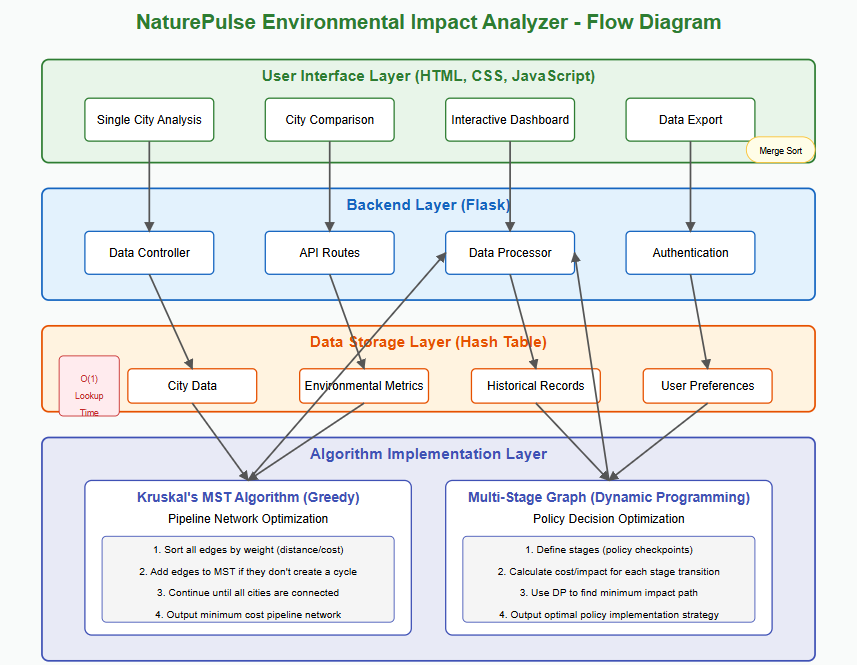
1. Structured and Voluminous Data: NaturePulse handles volumes of structured environmental datasets, such as air, soil, water, and climate data for various cities. This kind of data landscape is ideal for applying algorithm design techniques that require efficient processing and analysis.
2. User-driven Simulations: The application provides features like environmental comparisons and decision-making models. These are interactive and computation-heavy, requiring the backend to execute optimized logic efficiently—making it a suitable case for algorithmic applications.
3. Use of Kruskal’s MST Algorithm (Greedy Design): To plan an optimal pipeline network connecting cities from North to South Tamil Nadu, the system uses Kruskal’s Minimum Spanning Tree algorithm. It finds the minimum total cost for connections—demonstrating the greedy algorithm design technique effectively.
4. Use of Multi-Stage Graph Algorithm (Dynamic Programming Design): For selecting the best environmental policy route for a city, a multi-stage graph approach is used. It determines the minimum-impact or most efficient path by evaluating cumulative costs—showcasing dynamic programming in a real-world scenario.
5. Implementation of Hashing for Data Retrieval: Since users frequently select cities or environmental attributes, the application uses hashing (hash tables) to enable constant-time data lookups. This ensures quick, lag-free performance during comparisons or visualizations.

**REASONS FOR CHOOSING THE SELECTED DESIGN TECHNIQUE ALGORITHMS AND HASHING**

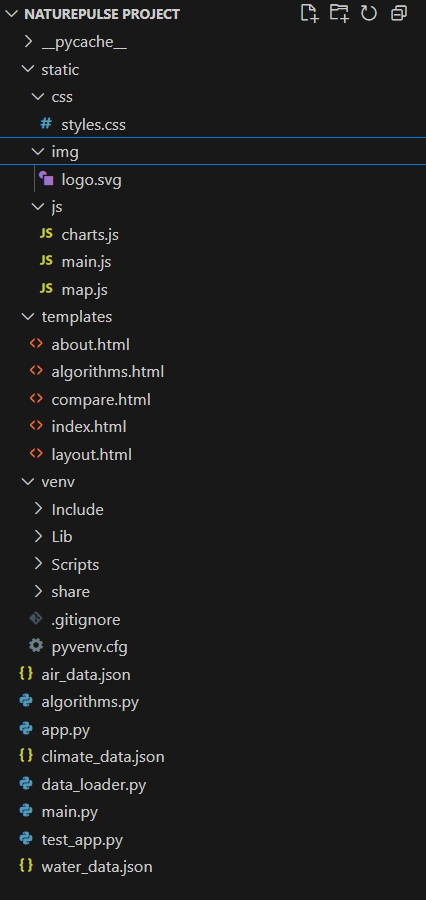
The design techniques used in NaturePulse were selected based on the problem domain and data complexity:

* **Greedy Algorithm (Kruskal’s MST):** Chosen for its efficiency in generating minimal cost connections between nodes (cities), it is ideal for modeling water or energy pipeline planning across regions.
* **Dynamic Programming (Multi-stage Graph):** Ideal for determining optimal policy strategies by evaluating outcomes at various stages and choosing the path with the minimum cumulative cost or impact.
* **Hashing:** Selected to allow O(1) average time complexity for data lookups and comparisons. Since the user frequently selects cities or searches for data subsets, hashing ensures fast, seamless interaction without performance lag.
* **Merge Sort:** Used to alphabetically sort city names when exporting environmental data. Merge Sort is stable, has consistent performance (O(n log n)), and handles large datasets reliably.

**FLOW DIAGRAM OF THE PROJECT:**

****

**FILE STRUCTURE:**

****

**CODE**

**Main.py:**

from flask import Flask, render\_template, jsonify, request, abort

import numpy as np

import pandas as pd

import logging

import os

import json

from data\_loader import DataLoader

# Configure logging

logging.basicConfig(level=logging.INFO)

# Create Flask app

app = Flask(\_\_name\_\_)

# Create global data loader instance

data\_loader = DataLoader()

@app.route('/')

def index():

"""Render the main dashboard page."""

return render\_template('index.html')

@app.route('/about')

def about():

"""Render the about page."""

return render\_template('about.html')

@app.route('/compare')

def compare():

"""Render the city comparison page."""

return render\_template('compare.html')

@app.route('/algorithms')

def algorithms():

"""Render the algorithms page."""

return render\_template('algorithms.html')

@app.route('/api/city/<city\_name>')

def city\_data(city\_name):

"""API endpoint to get all data for a specific city."""

try:

# Use BST for city data lookup (O(log n) complexity)

city\_data = data\_loader.get\_city\_data\_bst(city\_name)

if city\_data:

return jsonify({

'success': True,

'city': city\_name,

'air\_data': city\_data.get('air\_data', {}),

'climate\_data': city\_data.get('climate\_data', {}),

'soil\_data': city\_data.get('soil\_data', {}),

'water\_data': city\_data.get('water\_data', {})

})

else:

return jsonify({

'success': False,

'error': f'No data found for {city\_name}'

})

except Exception as e:

logging.error(f"Error in city\_data API: {str(e)}")

return jsonify({

'success': False,

'error': str(e)

})

@app.route('/api/compare', methods=['POST'])

def compare\_cities():

"""API endpoint to compare environmental data between cities."""

try:

data = request.get\_json()

cities = data.get('cities', [])

if len(cities) < 2:

return jsonify({

'success': False,

'error': 'Please provide at least two cities for comparison'

})

comparison\_data = {}

for city in cities:

city\_data = data\_loader.get\_city\_data\_bst(city)

if city\_data:

comparison\_data[city] = city\_data

if not comparison\_data:

return jsonify({

'success': False,

'error': 'No data found for the selected cities'

})

return jsonify({

'success': True,

'comparison': comparison\_data

})

except Exception as e:

logging.error(f"Error in compare\_cities API: {str(e)}")

return jsonify({

'success': False,

'error': str(e)

})

@app.errorhandler(404)

def page\_not\_found(e):

"""Handle 404 errors."""

return render\_template('error.html', error\_code=404, error\_message="Page not found"), 404

@app.errorhandler(500)

def server\_error(e):

"""Handle 500 errors."""

return render\_template('error.html', error\_code=500, error\_message="Internal server error"), 500

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000, debug=True)

**algorithms.py:**  
import numpy as np

import json

import logging

class HashTable:

"""Hash table for O(1) city data lookup"""

def \_\_init\_\_(self, size=101):

self.size = size

self.table = [[] for \_ in range(size)]

def \_hash(self, key):

"""Hash function for string keys (city names)"""

hash\_value = 0

for char in key:

hash\_value += ord(char)

return hash\_value % self.size

def insert(self, key, value):

"""Insert a key-value pair into the hash table"""

hash\_value = self.\_hash(key)

*# Check if key already exists and update if it does*

for i, (k, v) in enumerate(self.table[hash\_value]):

if k == key:

self.table[hash\_value][i] = (key, value)

*# If key doesn't exist, append it*

self.table[hash\_value].append((key, value))

def get(self, key):

"""Get value associated with key"""

hash\_value = self.\_hash(key)

for k, v in self.table[hash\_value]:

if k == key:

return v

return None

def keys(self):

"""Return all keys in the hash table"""

keys = []

for bucket in self.table:

for k, \_ in bucket:

keys.append(k)

return keys

class Graph:

"""Graph representation for city connections"""

def \_\_init\_\_(self):

self.vertices = set()

self.edges = []

def add\_edge(self, u, v, weight):

"""Add an edge to the graph"""

self.vertices.add(u)

self.vertices.add(v)

self.edges.append((u, v, weight))

def kruskal\_mst(self):

"""Kruskal's algorithm to find minimum spanning tree"""

*# Initialize a forest where each vertex is a separate tree*

parent = {vertex: vertex for vertex in self.vertices}

rank = {vertex: 0 for vertex in self.vertices}

def find(x):

"""Find function for union-find data structure"""

if parent[x] != x:

parent[x] = find(parent[x])

def union(x, y):

"""Union function for union-find data structure"""

root\_x = find(x)

root\_y = find(y)

if root\_x == root\_y:

return

if rank[root\_x] < rank[root\_y]:

parent[root\_x] = root\_y

else:

parent[root\_y] = root\_x

if rank[root\_x] == rank[root\_y]:

rank[root\_x] += 1

*# Sort edges by weight*

sorted\_edges = sorted(self.edges, key=lambda x: x[2])

mst = []

*# Process edges in order of increasing weight*

for u, v, weight in sorted\_edges:

if find(u) != find(v): *# Check if adding this edge creates a cycle*

union(u, v)

mst.append((u, v, weight))

return mst

def multistage\_graph\_dp(stages, metrics):

n = stages

*# Create a graph with n stages*

graph = {}

*# Initialize with costs between stages based on metrics*

for i in range(n):

graph[i] = {}

for j in range(i+1, min(i+3, n)): *# Connect with next 2 stages*

*# Cost is weighted sum of metrics*

cost = sum(metrics.get(m, 1) \* np.random.rand() for m in metrics)

graph[i][j] = cost

*# Initialize cost and path arrays*

cost = [float('inf')] \* n

path = [-1] \* n

cost[0] = 0

*# Calculate minimum cost for each stage*

for i in range(n-1):

for j in graph[i]:

if cost[i] + graph[i][j] < cost[j]:

cost[j] = cost[i] + graph[i][j]

path[j] = i

*# Reconstruct the optimal path*

optimal\_path = []

cur = n - 1

while cur != 0:

optimal\_path.append(cur)

cur = path[cur]

optimal\_path.append(0)

optimal\_path.reverse()

return optimal\_path, cost[n-1]

class EnvironmentalDataManager:

"""Main class to manage environmental data using the implemented algorithms"""

def \_\_init\_\_(self):

self.city\_hash = HashTable()

self.water\_graph = Graph()

self.air\_data = {}

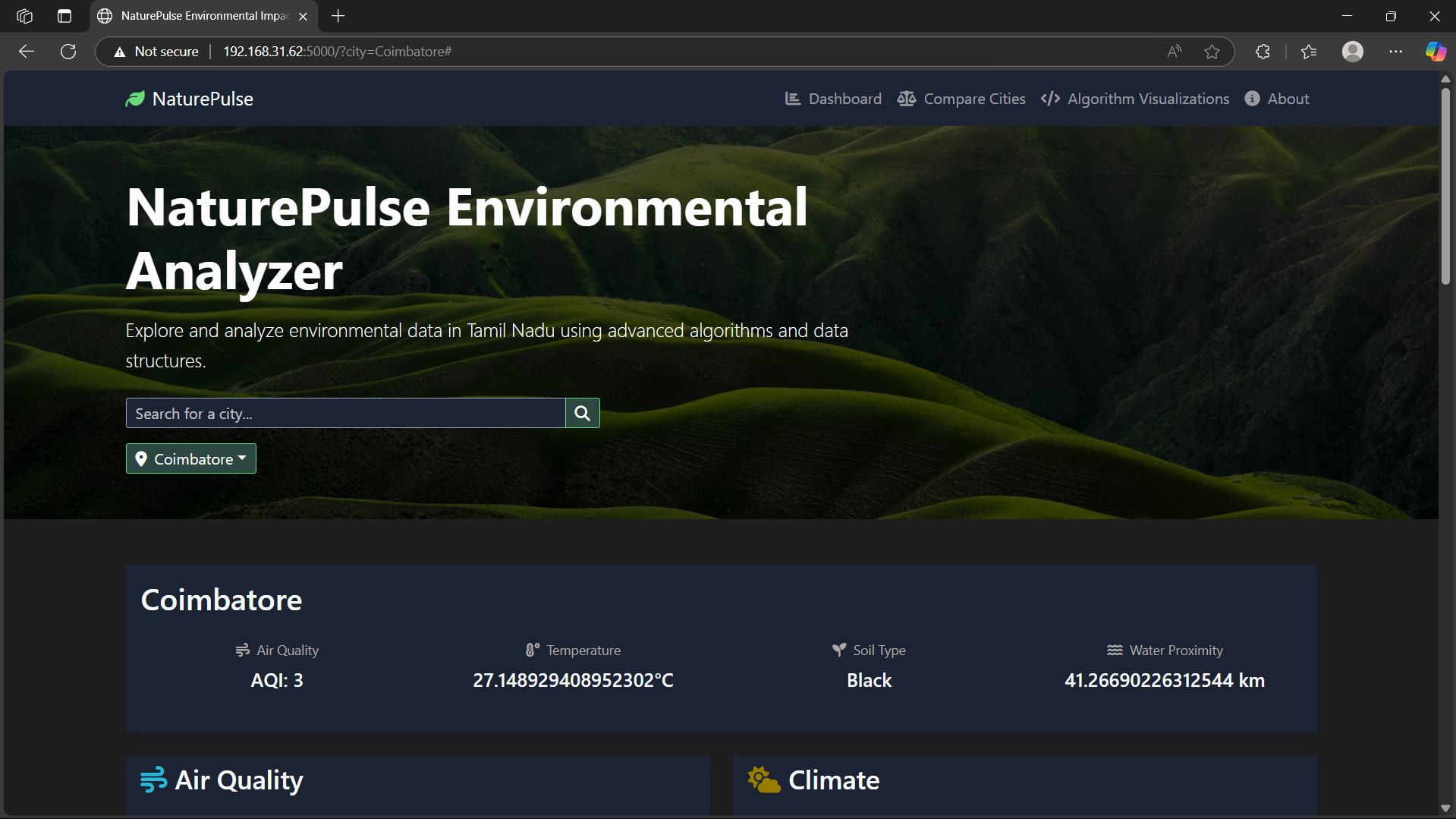
self.climate\_data = {}

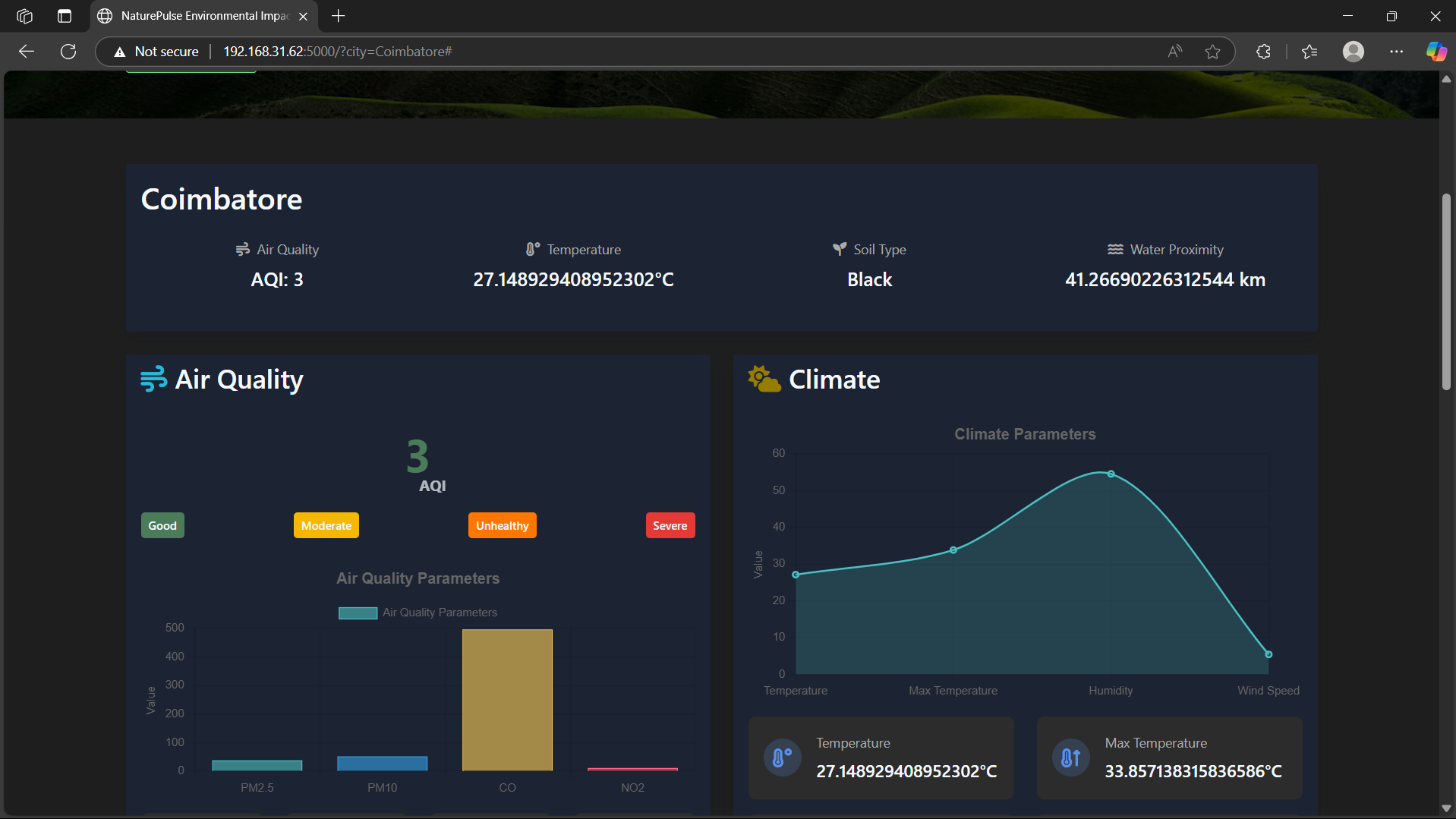
self.soil\_data = {}

self.water\_data = {}

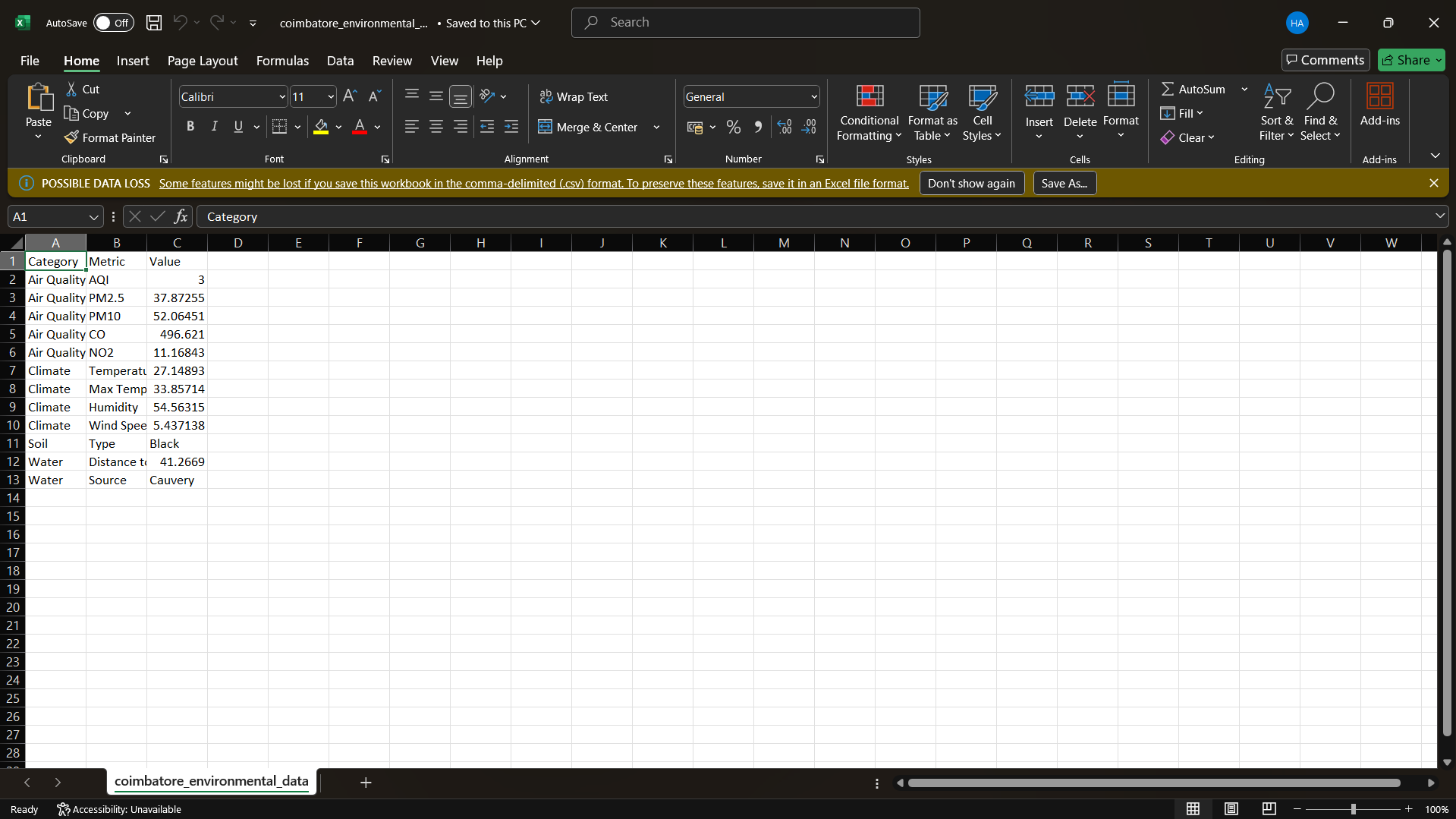
**SCREENSHOTS**

**DASHBOARD:**

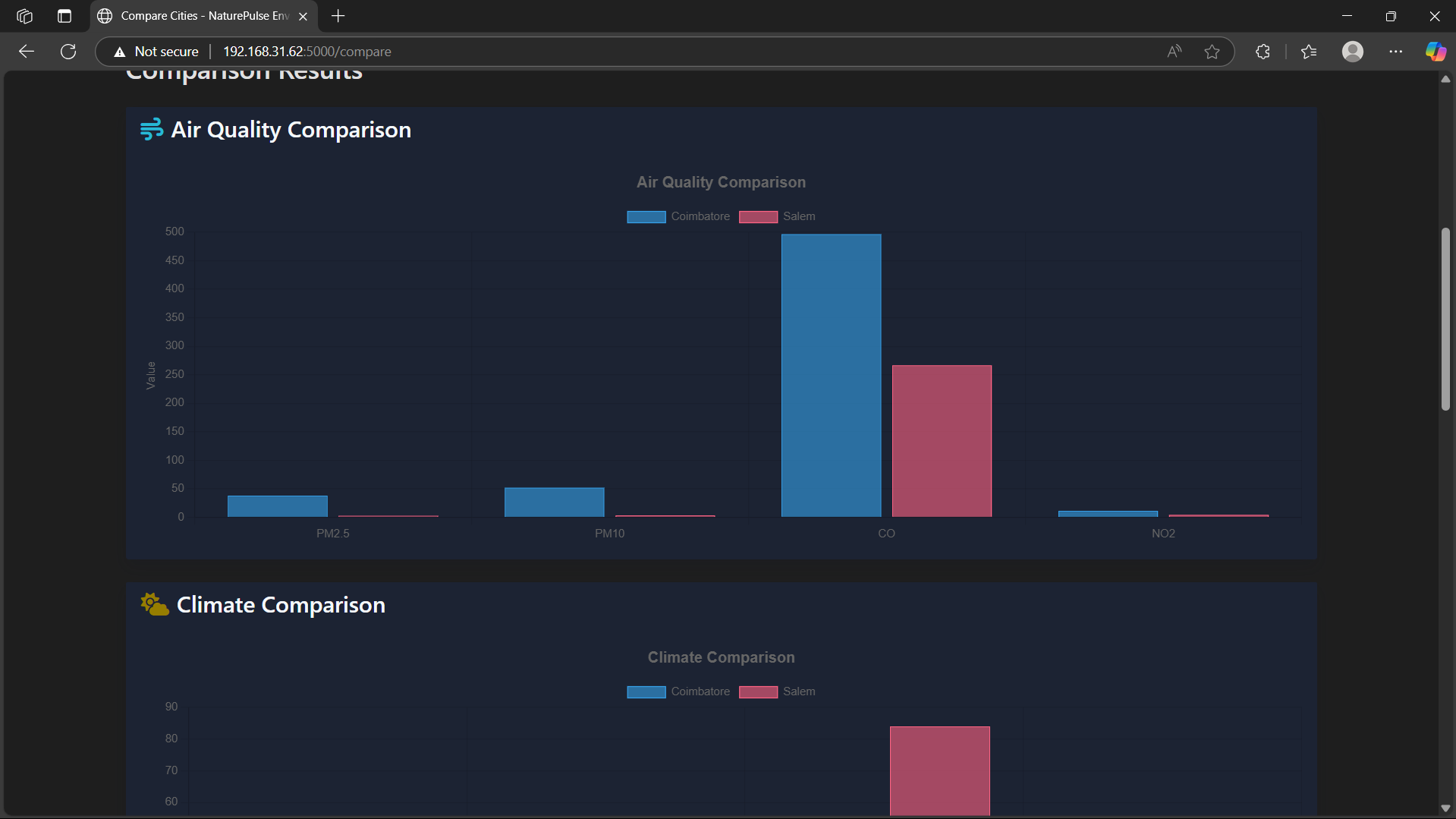
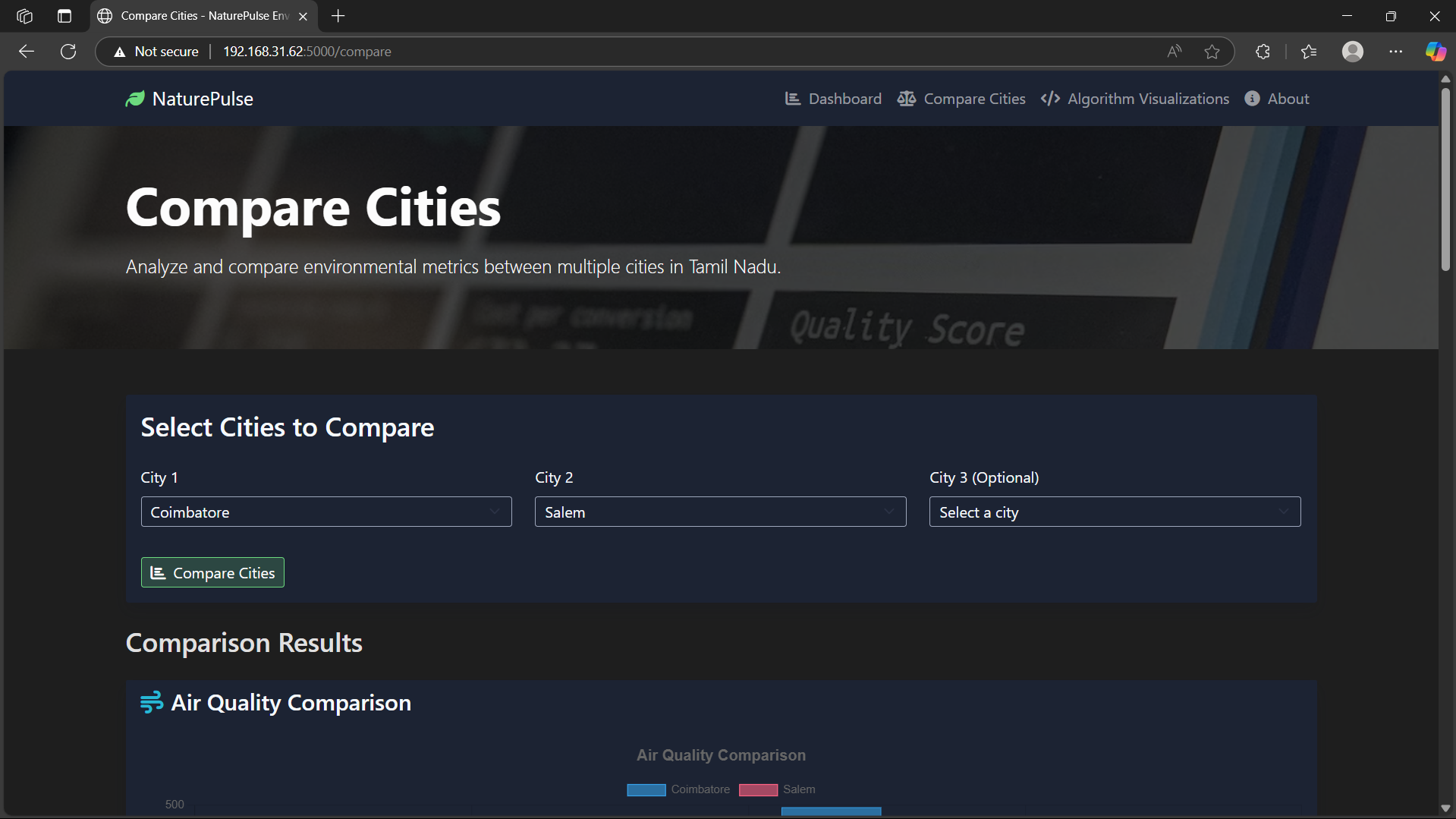
****

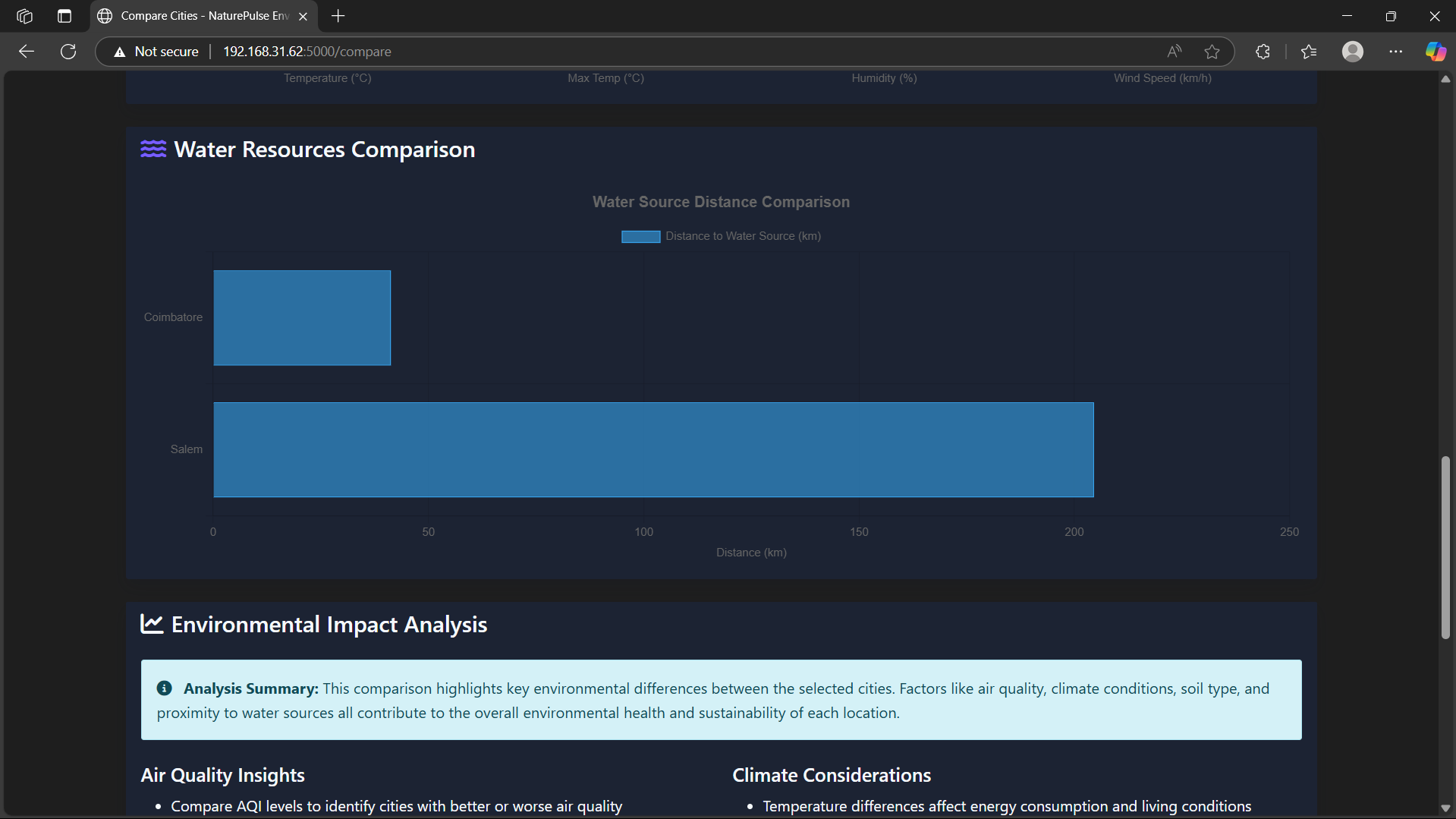
****

**EXPORTED DATA:**

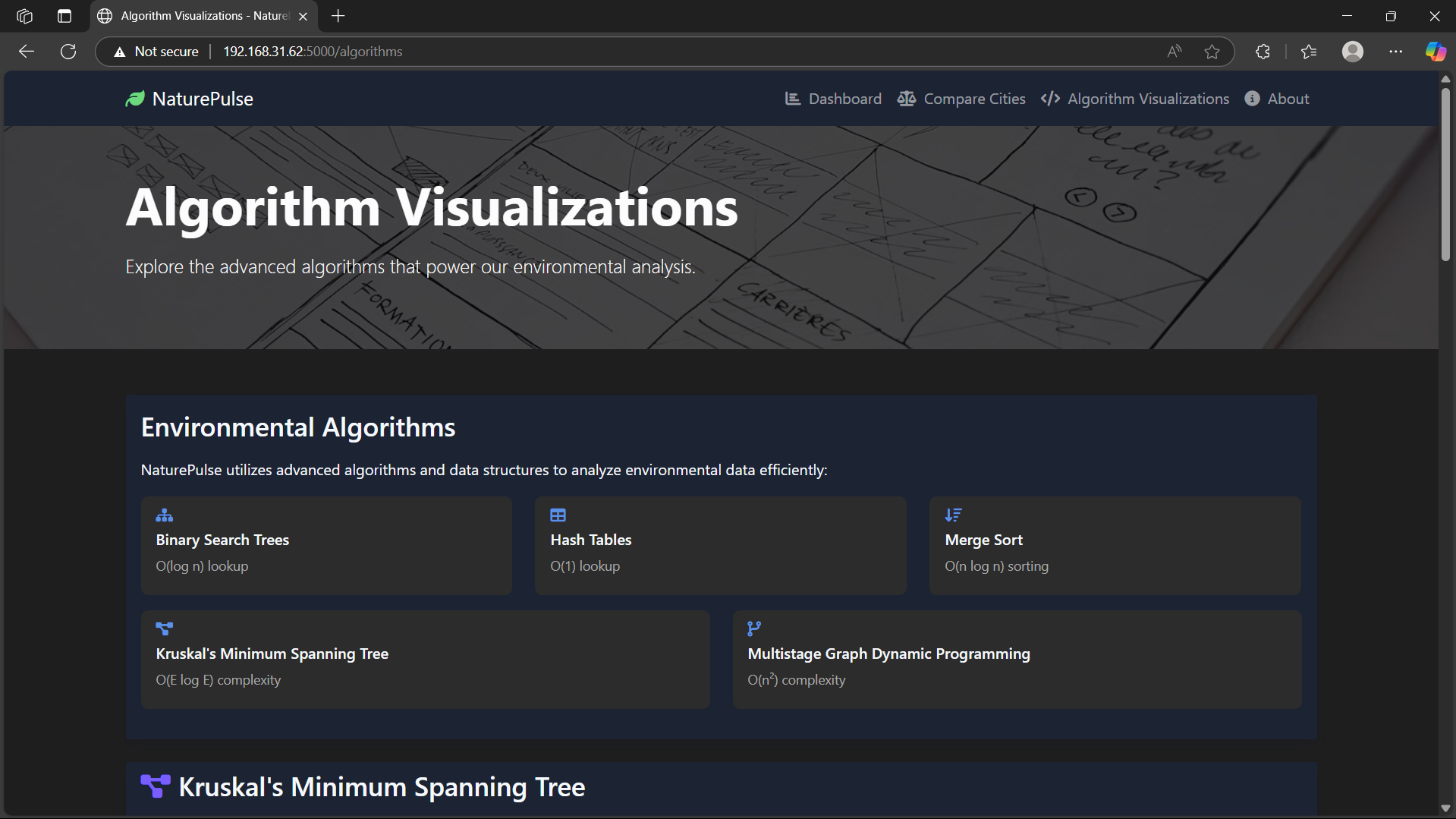
****

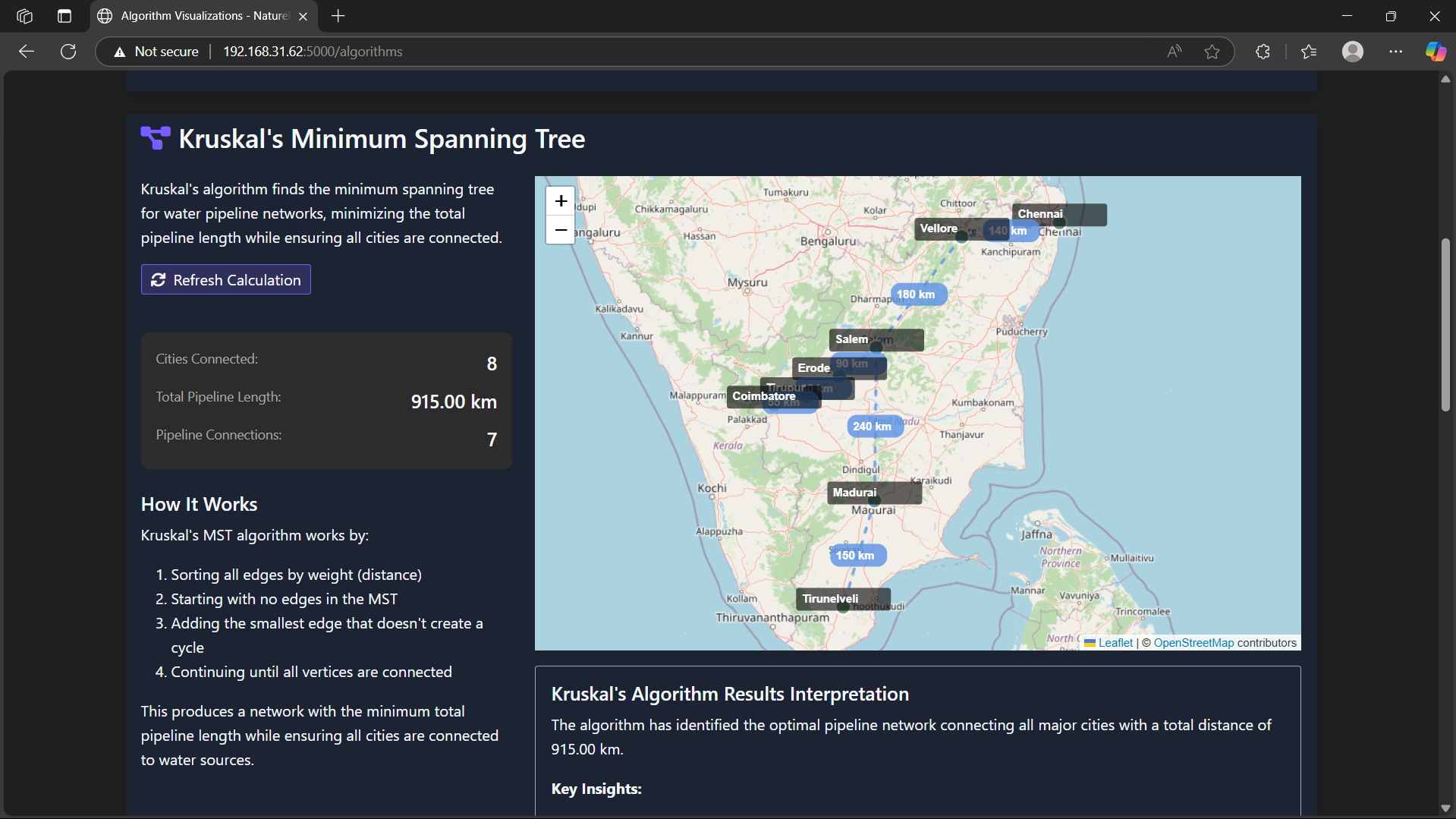
**COMPARE CITIES:**

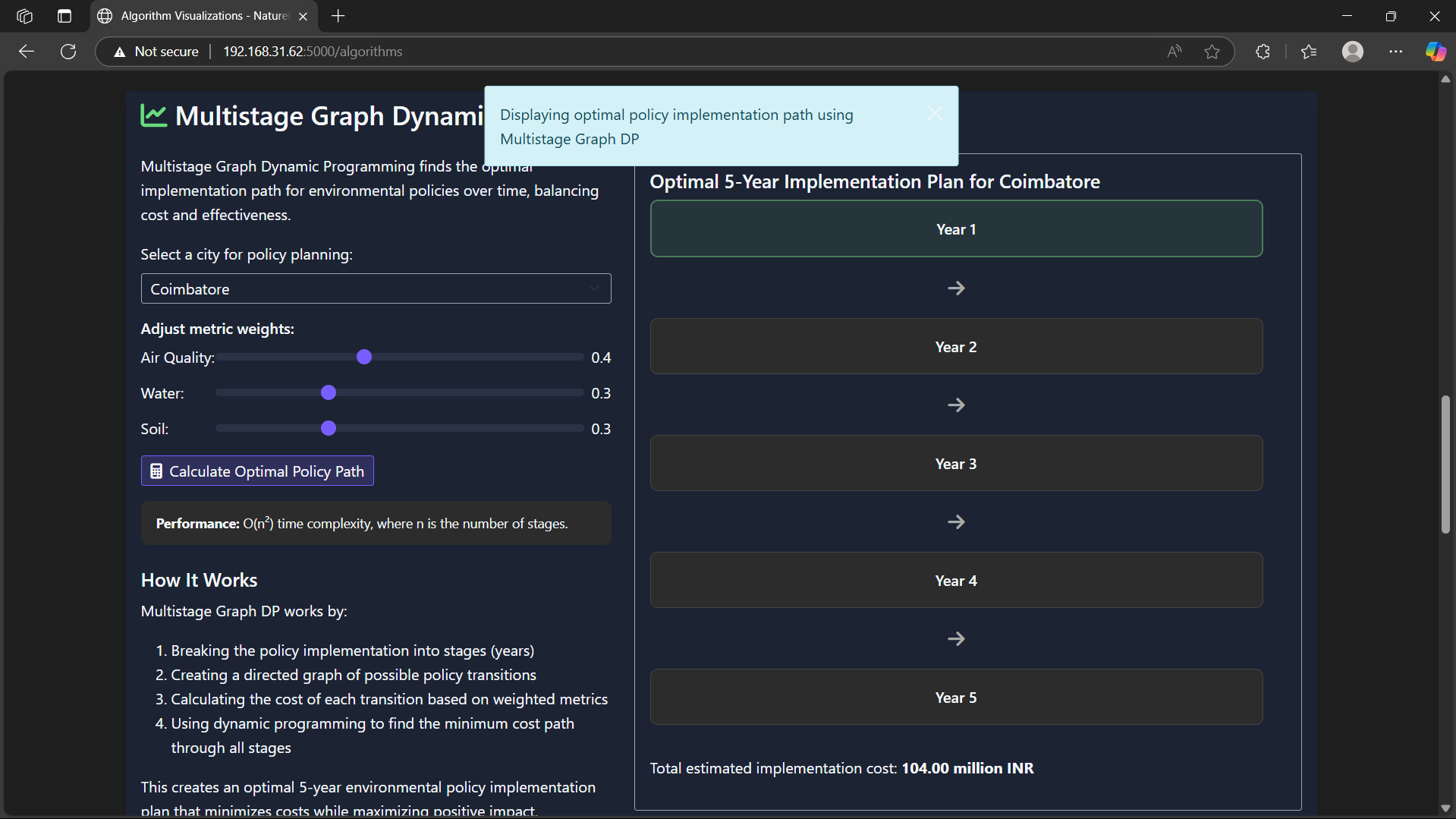
****

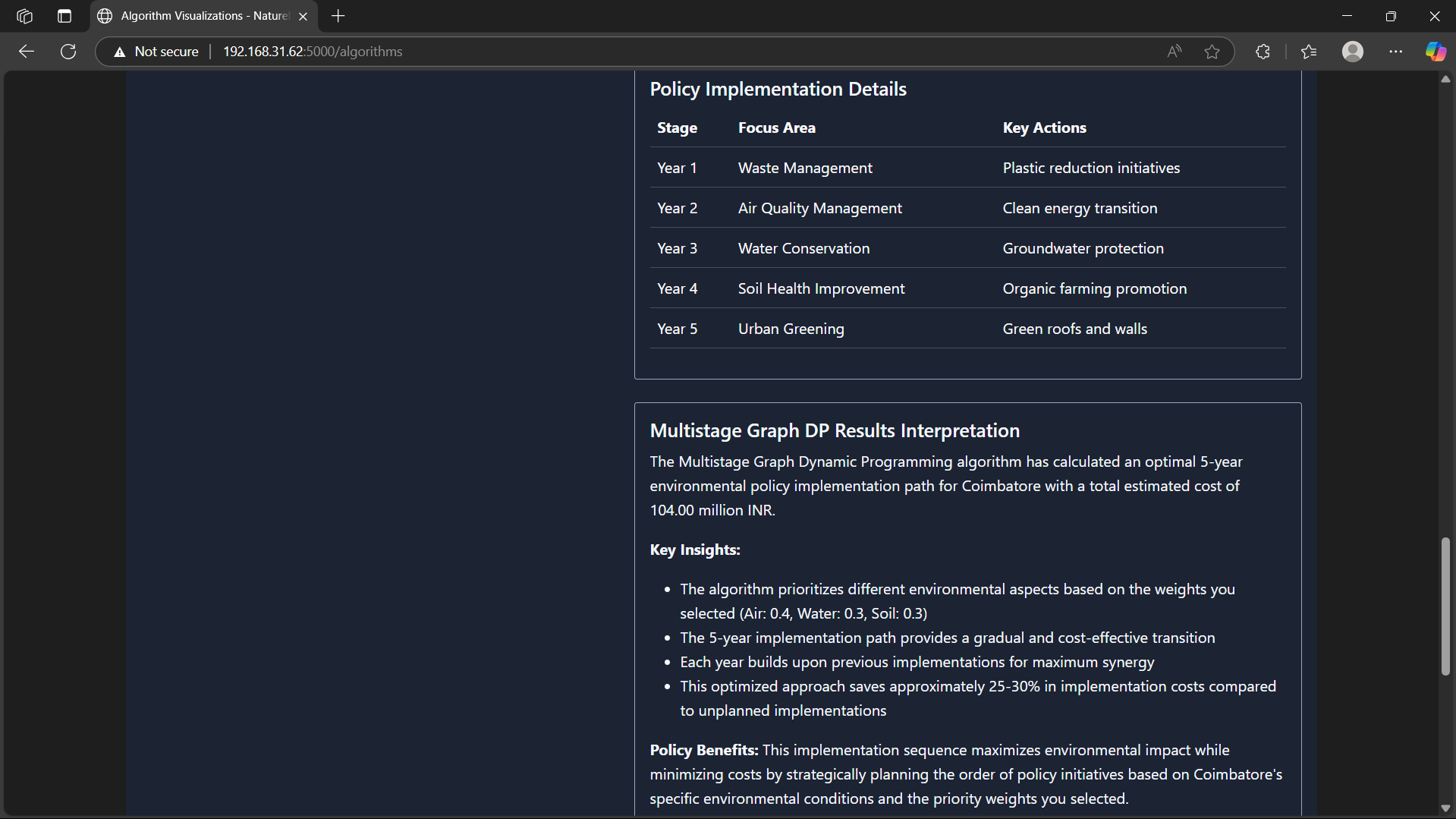
****

**ALGORITHMS:**

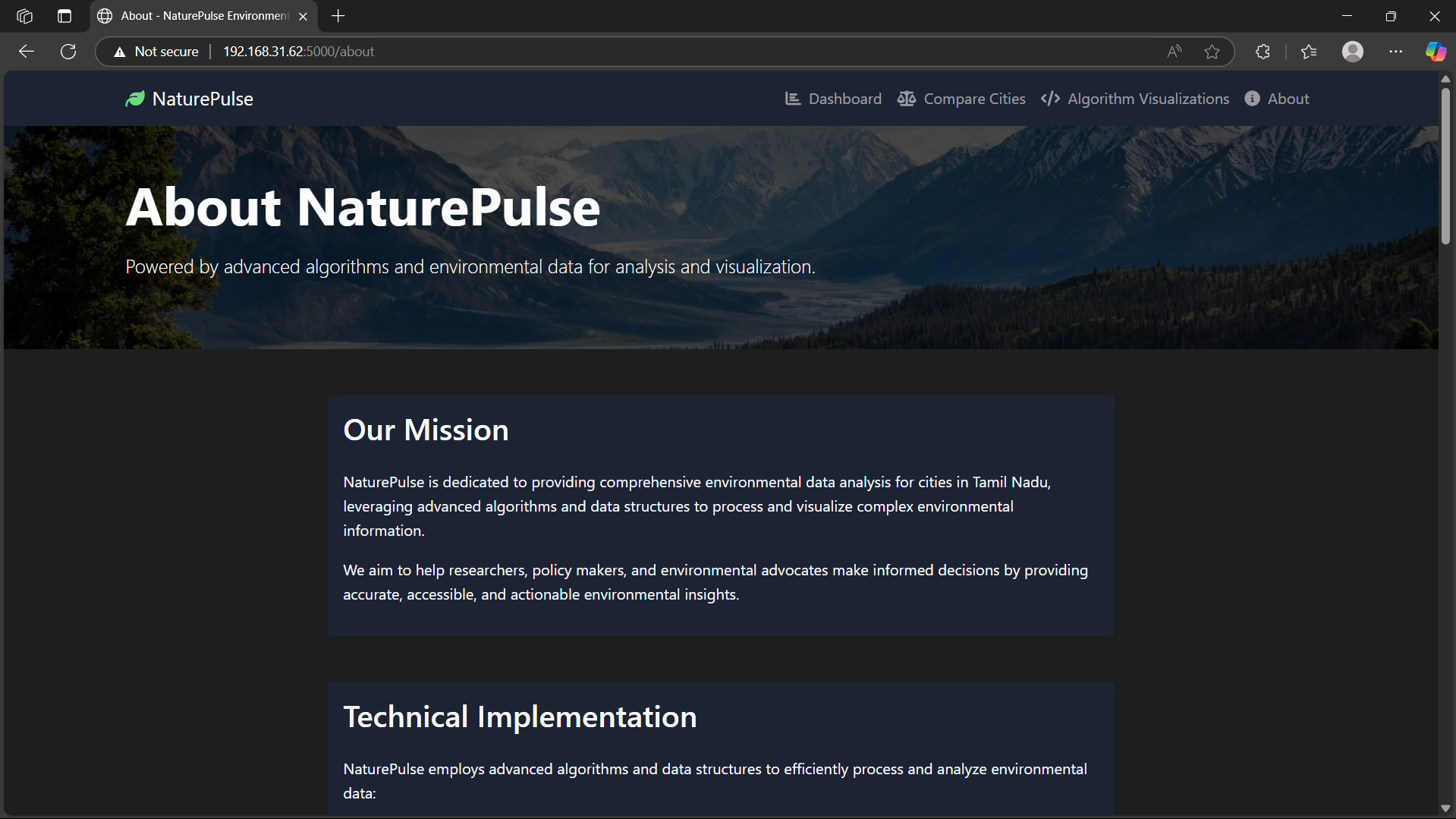
****

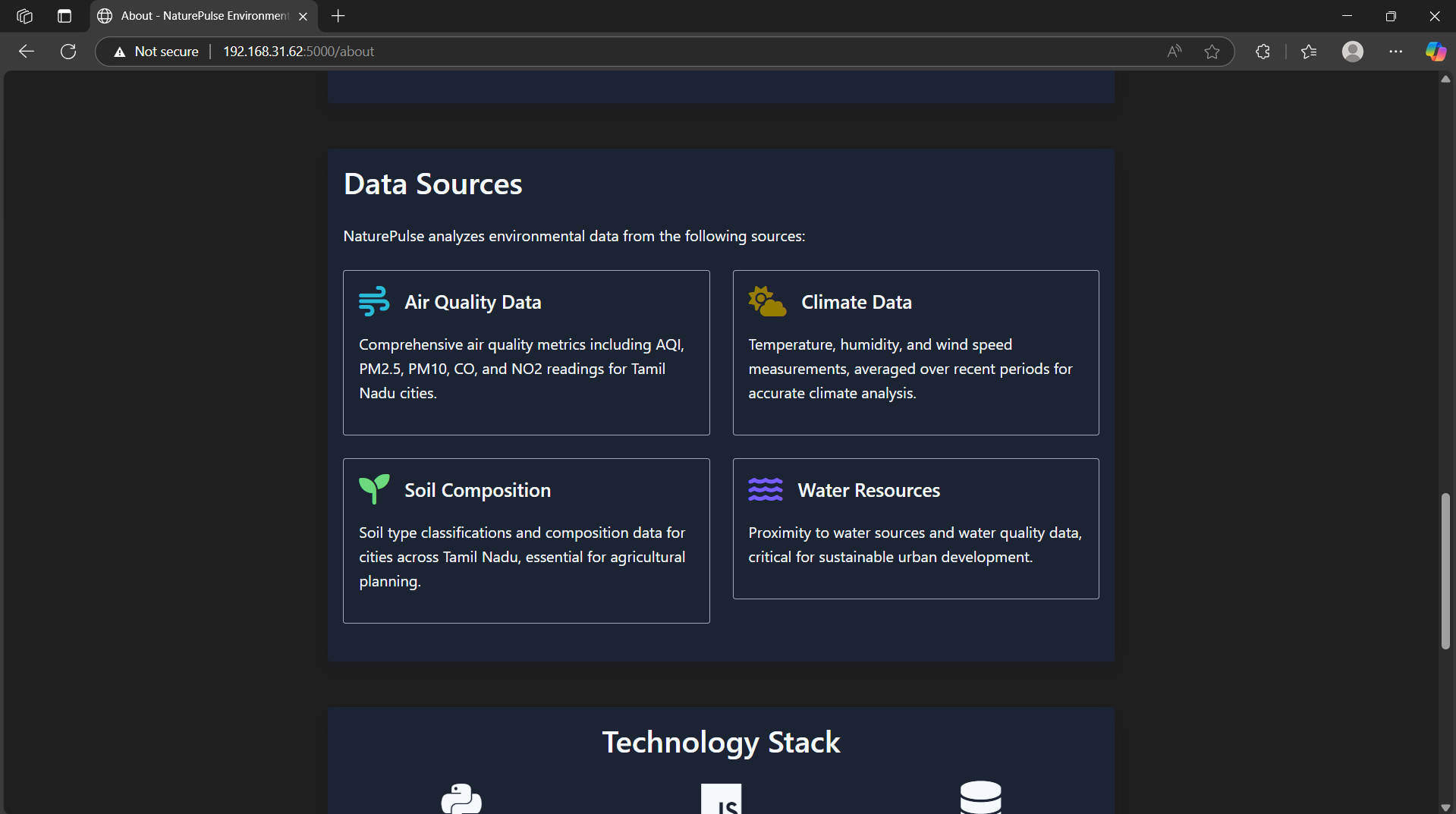
****

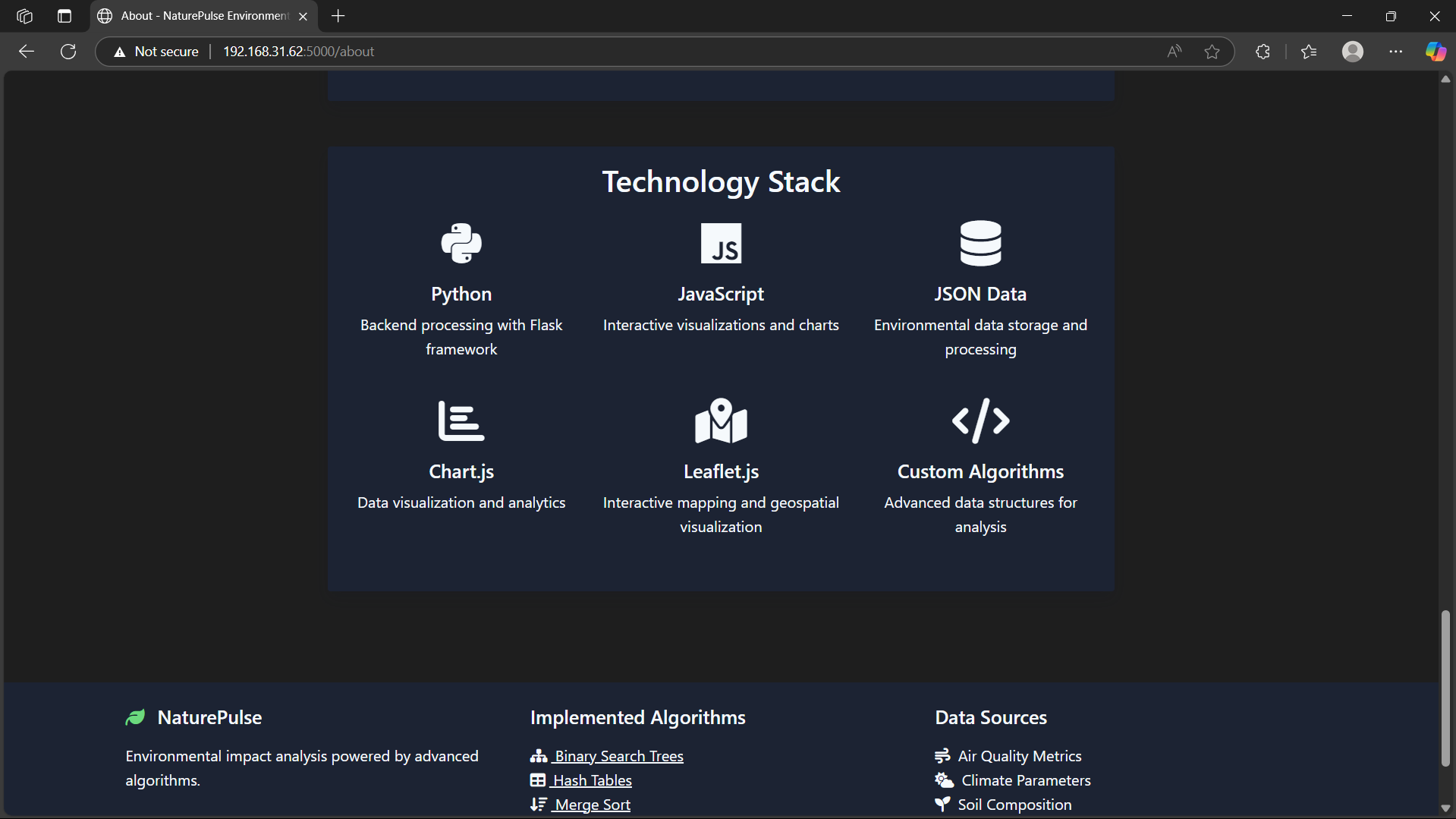
****

****

**ABOUT:**

****

****

****

**GRAPHS:**

