Route planning between Indian cities

Madhav Ganesan

125156066

* Problem Statement:

“*The problem of finding the shortest path between any two cities is a cumbersome task. This problem is of supreme importance in various applications like logistics, transportation and real estate.*”

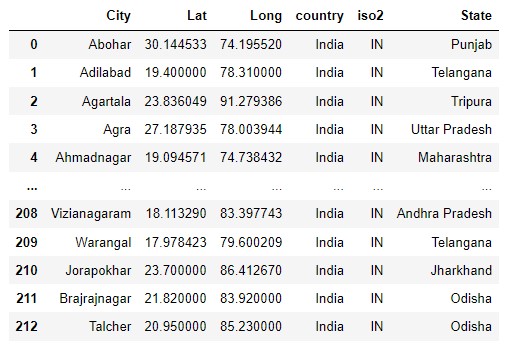
* Aim:

The aim of this project is to develop a highly efficient and scalable solution for the challenging problem of finding the shortest path between any two cities.

* Dataset used:

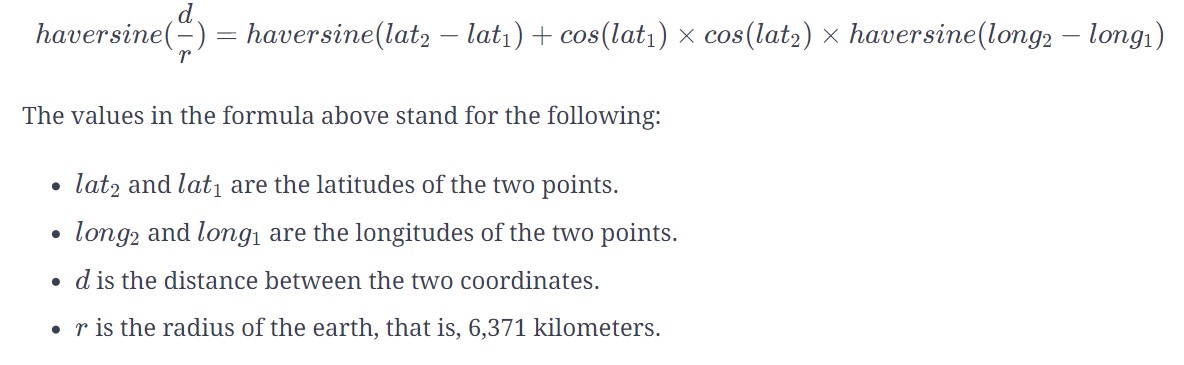
Indian cities dataset from Kaggle

https://www.kaggle.com/datasets/parulpandey/indian-cities-database



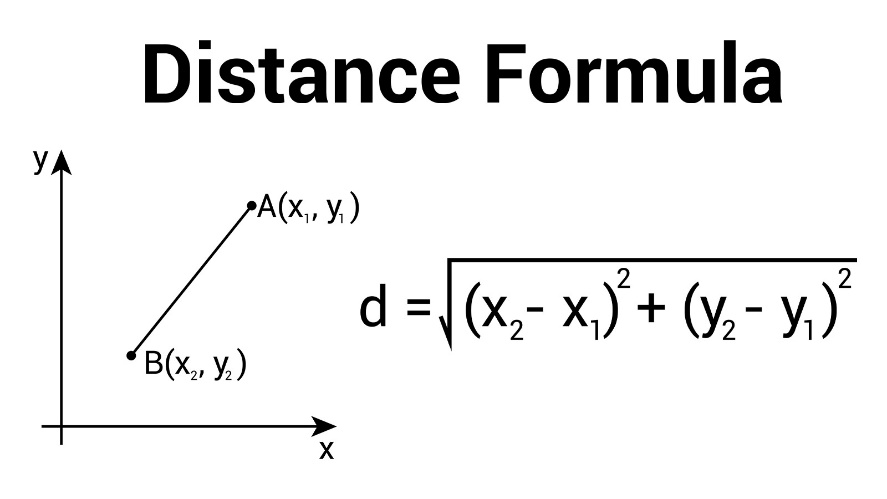
* Haversine Formula:

The Haversine formula is a mathematical formula used to calculate the distance between any two points on the surface of a sphere, given their latitude and longitude coordinates. It is commonly used in navigation and geographical applications, such as calculating distances between locations on the Earth.

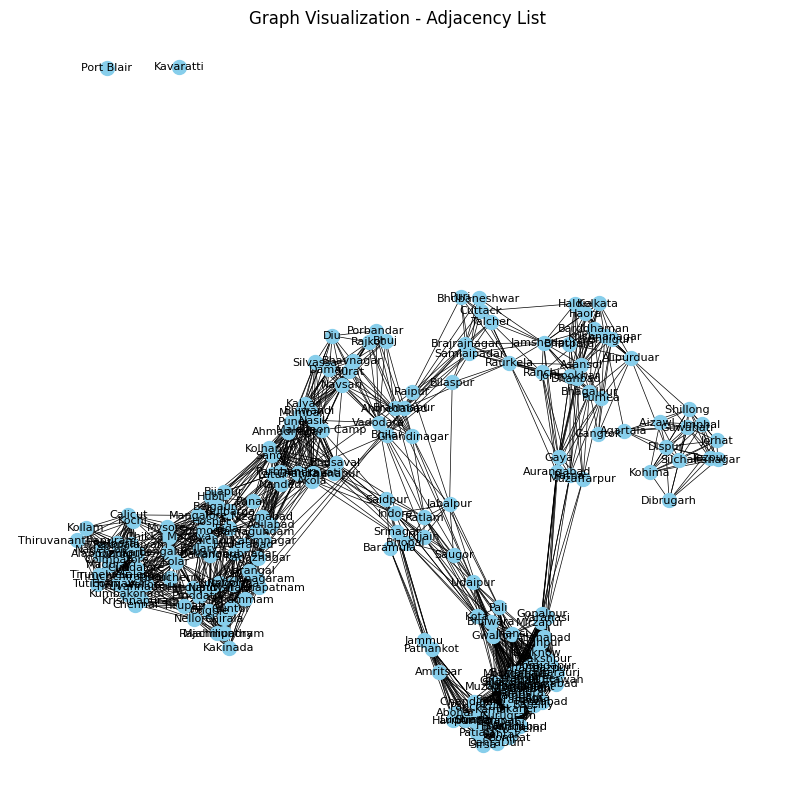


* Euclidean Distance Formula:

Euclidean distance is a measure of the straight-line distance between two points in Euclidean space. I have used **Euclidean distance** between two cities as the **heuristic value** for this informed search algorithm. In two-dimensional space, the Euclidean distance between two points (x1​, y1​) and (x2​, y2​) is calculated using the formula:



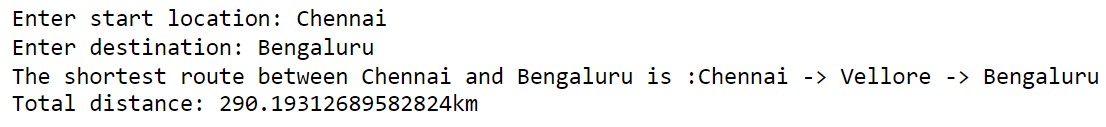
* Graph Visualization:



* A\* Algorithm:

A\* is an **informed search algorithm**, where it uses **heuristics** to guide its search process. The algorithm efficiently finds the path with the lowest cost from a start node to a goal node while exploring the minimum number of nodes. It is widely used for pathfinding and graph traversal algorithm.

* import numpy as np
* import pandas as pd
* data=pd.read\_csv("Indian\_cities.csv")
* #Graph creation
* graph={}
* def graph\_creation():
* global graph
* for index, row in data.iterrows():
* graph[row['City']]=(row['Lat'],row['Long'],row['State'].strip())
* graph\_creation()
* #Calculates distance between adjacent cities
* def calculate\_distance(coord1, coord2):
* lat1, lon1 ,state1= coord1
* lat2, lon2 ,state2= coord2
* R = 6371
* diff\_lat = np.radians(lat2 - lat1)
* diff\_lon = np.radians(lon2 - lon1)
* a = np.sin(diff\_lat/2)\*\*2 + np.cos(np.radians(lat1))\*np.cos(np.radians(lat2))\*np.sin(diff\_lon/2)\*\* 2
* c = 2\*np.arctan2(np.sqrt(a), np.sqrt(1 - a))
* distance = R \* c
* return distance
* adjacency\_list1={}
* def adjacency\_list\_creation(graph):
* global adjacency\_list1
* cities = list(graph.keys())
* for i in range(len(cities)):
* city\_adjacency = []
* for j in range(len(cities)):
* if i != j:
* distance = calculate\_distance(graph[cities[i]], graph[cities[j]])
* if distance < 200 or (graph[cities[i]][-1]==graph[cities[j]][-1]):
* city\_adjacency.append(cities[j])
* adjacency\_list1[cities[i]] = city\_adjacency
* adjacency\_list\_creation(graph)
* from math import sqrt
* import heapq as hq
* def heuristic(point1, point2):
* x1, y1 ,state1= point1
* x2, y2 ,state2= point2
* distance = sqrt((x2 - x1)\*\*2 + (y2 - y1)\*\*2)
* return distance
* def astar(ad\_list, start, goal, graph):
* open\_set = [(0, start)]
* came\_from = {}
* g\_score = {city: float('inf') for city in ad\_list.keys()}
* g\_score[start] = 0
* while open\_set:
* current\_score, current\_city = hq.heappop(open\_set)
* if current\_city == goal:
* path = make\_path(came\_from, start, goal)
* return path, g\_score[goal]
* for neighbor in ad\_list[current\_city]:
* distance = calculate\_distance(graph[current\_city], graph[neighbor])xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
* tentative\_g\_score = g\_score[current\_city] + distance
* if tentative\_g\_score < g\_score[neighbor]:
* g\_score[neighbor] = tentative\_g\_score
* f\_score = tentative\_g\_score + heuristic(graph[neighbor], graph[goal])
* hq.heappush(open\_set, (f\_score, neighbor))
* came\_from[neighbor] = current\_city
* return None, None
* def make\_path(came\_from, start, goal):
* path = [goal]
* while goal != start:
* goal = came\_from[goal]
* path.append(goal)
* return path[::-1]
* start=input("Enter start location: ").strip()
* goal=input("Enter destination: ").strip()
* path, distance = astar(adjacency\_list1, start, goal,graph)
* if path and distance:
* print(f"The shortest route between {start} and {goal} is :{' -> '.join(path)}")
* print(f'Total distance: {distance}km')
* else:
* print(f"No feasible path between {start} and {goal}")
* Output:



* Result:

This project successfully delivers an optimized pathfinding solution for finding the shortest route between any two cities. The implemented algorithm, a variant of A\* (A-star), ensures efficiency, scalability, and practical integration into various applications like logistics and transportation.