A* Pathfinding Algorithm for City Navigation

Introduction

This document presents a Python implementation of the A* algorithm, tailored for finding the shortest path between cities in Pakistan. This algorithm combines geographical coordinates and road distances to efficiently navigate between cities. It leverages heuristic evaluation using Euclidean distance and visualizes paths using network graphs.

Libraries Used

```
import csv
import math
import heapq
import matplotlib.pyplot as plt
import networkx as nx
```

- **csv**: This module provides functions for reading and writing CSV files. It's used here to load city and distance data from CSV files.
- **math**: Provides mathematical functions. In this code, it's used to compute the Euclidean distance in the heuristic function.
- **heapq**: Implements a heap queue algorithm (priority queue). It's used for efficient priority queue operations in the A* search algorithm.
- **matplotlib.pyplot**: Provides a MATLAB-like plotting framework. It's used to visualize the network graph of cities and paths.
- **networkx**: A library for the creation, manipulation, and study of complex networks (graphs). It's used here to represent cities and their connections as a graph.

Function Definitions

load_cities(file_path)

Purpose: Reads a CSV file containing cities and their coordinates, constructs a dictionary cities where keys are city names and values are tuples of (latitude, longitude).

Explanation:

```
# Load the datasets
def load_cities(file_path):
    cities = {}
    with open(file_path, 'r', encoding='utf-8-sig') as file:
        reader = csv.DictReader(file)
        headers = reader.fieldnames
        print(f"Cities CSV headers: {headers}")  # Debugging line
        for row in reader:
            cities[row['City']] = (float(row['Latitude']),
float(row['Longitude']))
    return cities
```

- Opens the CSV file specified by file_path using open() function with utf-8-sig encoding to handle UTF-8 with BOM (Byte Order Mark).
- Uses csv.DictReader to read each row as a dictionary with column headers as keys ('City', 'Latitude', 'Longitude').
- Iterates through each row and populates the cities dictionary where keys are city names (row['City']) and values are tuples of latitude and longitude coordinates.

load_distances(file_path)

Purpose: Reads a CSV file containing distances between pairs of cities, constructs a nested dictionary distances where distances[city1][city2] gives the distance between city1 and city2.

Explanation:

```
def load_distances(file_path):
    distances = {}
    with open(file_path, 'r', encoding='utf-8-sig') as file:
        reader = csv.DictReader(file)
        headers = reader.fieldnames
        print(f"Distances CSV headers: {headers}")  # Debugging line
        for row in reader:
            if row['City1'] not in distances:
                distances[row['City1']] = {}
            if row['City2'] not in distances:
                distances[row['City2']] = {}
            distances[row['City1']][row['City2']] = float(row['Distance'])
            distances[row['City2']][row['City1']] = float(row['Distance'])
    return distances
```

- Opens the CSV file specified by file_path similarly using csv.DictReader to read each row.
- Initializes an empty dictionary distance to store distances between cities.
- Populates the distances dictionary ensuring bidirectional connectivity by adding distances for both city1 to city2 and city2 to city1.

heuristic(city1, city2, cities)

Purpose: Calculates the Euclidean distance between two cities based on their coordinates.

Explanation:

```
# Heuristic function: Euclidean distance
def heuristic(city1, city2, cities):
    lat1, lon1 = cities[city1]
    lat2, lon2 = cities[city2]
    return math.sqrt((lat1 - lat2) ** 2 + (lon1 - lon2) ** 2)
```

- Retrieves latitude (lat1, lat2) and longitude (lon1, lon2) for city1 and city2 from the cities dictionary.
- Computes and returns the Euclidean distance using the formula math.sqrt((lat1 - lat2) ** 2 + (lon1 - lon2) ** 2).

a_star_search(start, goal, cities, distances)

Purpose: Implements the A* search algorithm to find the shortest path from start city to goal city using provided cities and distances data.

Explanation:

```
# A* Algorithm
def a_star_search(start, goal, cities, distances):
    open_list = []
    heapq.heappush(open_list, (0 + heuristic(start, goal, cities), 0, start,
[]))
    closed_list = set()

    while open_list:
        _, cost, current_city, path = heapq.heappop(open_list)

        if current_city in closed_list:
            continue

        path = path + [current_city]

        if current_city == goal:
            return path, cost

        closed_list.add(current_city)

        for neighbor, distance in distances[current_city].items():
            if neighbor not in closed_list:
```

```
                new_cost = cost + distance
                heapq.heappush(open_list, (new_cost + heuristic(neighbor,
goal, cities), new_cost, neighbor, path))

    return None, float('inf')
```

- Initializes an empty priority queue (open_list) using heapq to store tuples of (total_cost, cost_so_far, current_city, path).
- Pushes the starting city (start) onto open_list with a priority based on heuristic and initial cost.
- Uses a closed_list set to track visited cities to avoid revisiting and potential loops.
- Iterates through open_list, expanding the path by exploring neighbors (distances[current_city].items()).
- Computes new costs and updates open_list with new paths if a lower cost path to a neighbor is found.
- Terminates and returns the optimal path (path) and its cost when the goal city (goal) is reached.

plot_graph(cities, distances, path)

Purpose: Visualizes the cities and their connections as a network graph using networkx and matplotlib.pyplot. It highlights the optimal path (path) found by A* in red.

Explanation:

```
# Plotting the graph
def plot_graph(cities, distances, path):
    G = nx.Graph()

    for city, coords in cities.items():
        G.add_node(city, pos=coords)

    for city1, neighbors in distances.items():
        for city2, distance in neighbors.items():
            G.add_edge(city1, city2, weight=distance)

    pos = nx.get_node_attributes(G, 'pos')

    # Draw nodes with specific properties
    nx.draw_networkx_nodes(G, pos, node_size=500, node_color='skyblue',
edgecolors='black')

    # Draw edges with weights
    nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=1.5,
edge_color='gray')
```

```
    # Highlight the path edges
    if path:
        path_edges = [(path[i], path[i + 1]) for i in range(len(path) - 1)]
        nx.draw_networkx_edges(G, pos, edgelist=path_edges, width=3,
edge_color='red')

    # Draw node labels
    nx.draw_networkx_labels(G, pos, font_size=10, font_color='black')

    # Draw edge labels (distances)
    edge_labels = {(city1, city2): f'{int(distances[city1][city2])} km' for
city1 in distances for city2 in distances[city1]}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_size=8, label_pos=0.5)

    plt.title("City Map with A* Pathfinding")
    plt.show()
```

- Creates an empty graph (G) using networkx (nx.Graph()).
- Adds nodes for each city with their coordinates (cities) and edges with weights corresponding to distances (distances).
- Retrieves positions (pos) of nodes for graph layout using nx.get_node_attributes.
- Uses matplotlib.pyplot to draw nodes (nx.draw_networkx_nodes), edges (nx.draw_networkx_edges), labels (nx.draw_networkx_labels), and highlights the optimal path (path) in red if provided.

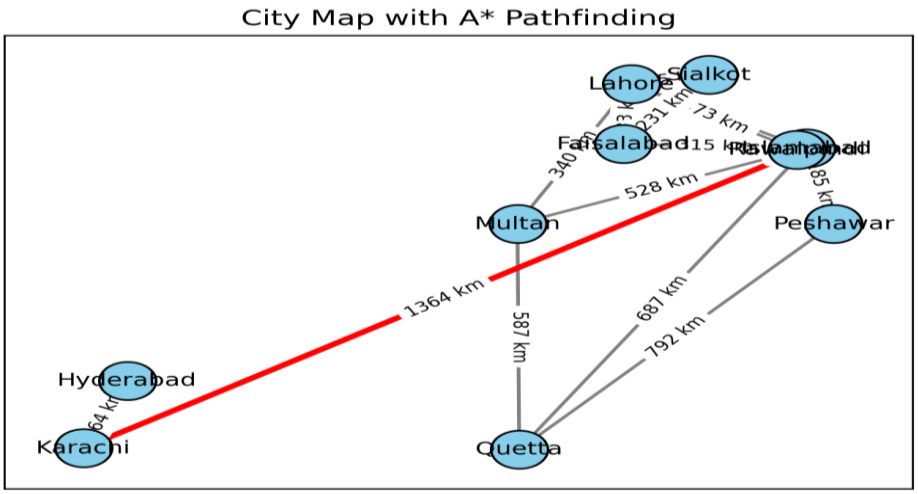
Main Program Execution

```
# File paths
cities_file_path = r'C:\Users\Imtiaz Ahmed\OneDrive\Desktop\Astar
Search\Pakistn_cities.csv'
distances_file_path = r'C:\Users\Imtiaz Ahmed\OneDrive\Desktop\Astar
Search\Pakistan_distances2.csv'

# Load data
cities = load_cities(cities_file_path)
distances = load_distances(distances_file_path)

# Main loop for continuous input
while True:
    # Take user input for start and goal cities
    start_city = input("Enter the start city (or type 'exit' to quit): ")
    if start_city.lower() == 'exit':
        break
```

```python
    goal_city = input("Enter the goal city: ")

    # Perform A* search
    path, cost = a_star_search(start_city, goal_city, cities, distances)

    print(f"Path from {start_city} to {goal_city}: {path}")
    print(f"Total cost: {cost}")

    # Plot the graph with the path
    plot_graph(cities, distances, path)
```

Output

```
PS C:\Users\Imtiaz Ahmed\OneDrive\Desktop\Astar Search> & "C:/Users/Imtiaz
/Desktop/Astar Search/A_star_search.py"
Cities CSV headers: ['City', 'Latitude', 'Longitude']
Distances CSV headers: ['City1', 'City2', 'Distance']
Enter the start city (or type 'exit' to quit): Rawalpindi
Enter the goal city: Karachi
Path from Rawalpindi to Karachi: ['Rawalpindi', 'Islamabad', 'Karachi']
Total cost: 1378.0
```



Conclusion

This document demonstrates a Python implementation of the A* algorithm for city navigation using geographical coordinates and road distances in Pakistan. It emphasizes modularity, efficiency, and visualization to aid in understanding and implementing pathfinding algorithms in geographic contexts. The provided code can be extended and adapted for various applications requiring optimal route planning between locations based on distance and heuristic estimates.