Implementation of A* Algorithm for Shortest Path Finding in Geospatial Data

Ruhaan Choudhary
Indian Institute of Technology Ropar

June 10, 2024

Contents

1 Introduction

The A* algorithm is a popular pathfinding algorithm used in various fields, including robotics, gaming, and geographic information systems (GIS). It efficiently finds the shortest path between two nodes in a graph by combining the advantages of Dijkstra's algorithm and greedy best-first search. In this paper, we present an implementation of the A* algorithm for finding the shortest path in geospatial data using Python libraries such as OSMnx and NetworkX.

2 Extracting Geospatial Data

Geospatial data is essential for performing pathfinding algorithms on real-world maps. We use the OSMnx library, which allows us to retrieve OpenStreetMap data for any location. First, we define the area of interest, such as a city or a region, using a place name. Then, we download the street network graph representing the road network in that area. Finally, we find the nearest graph nodes to the start and end points of our path.

2.1 Code Snippet

The following Python code snippet demonstrates the process of extracting geospatial data using OSMnx:

```
import osmnx as ox

# Define the place and download the street network
place_name = "San Francisco, California, USA"
G = ox.graph_from_place(place_name, network_type='drive')
```

3 Haversine Distance Calculation

Calculating distances between geographical coordinates is crucial for determining the cost of moving between nodes in the graph. We employ the Haversine distance formula, which accurately computes the great-circle distance between two points on the Earth's surface given their latitude and longitude. This distance metric ensures that our pathfinding algorithm operates correctly on geographic data.

3.1 Code Snippet

The following Python code snippet defines the Haversine distance function:

```
import math

# Define the Haversine distance function
def haversine(lat1, lon1, lat2, lon2):
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    r = 6371  # Radius of Earth in kilometers
    return r * c
```

4 Implementing A* Algorithm with Priority Queues

The A* algorithm efficiently explores paths in the graph by prioritizing nodes based on a combination of the cost to reach them from the start node (the "g-cost") and an estimate of the cost to reach the goal node from them (the "h-cost"). We implement the algorithm using a priority queue data structure, which ensures that nodes with lower total costs are explored first. Additionally, we define a heuristic function based on the Haversine distance to estimate the cost from a node to the goal node.

4.1 Explanation of the A* Algorithm

The A* algorithm is a heuristic search algorithm used for pathfinding in graphs. Here's an explanation of the `a_star` function step by step:

1. **Initialization**:
   - We start by initializing several data structures:
     - `queue`: This is a priority queue used to store nodes to be explored, prioritized by their estimated total cost (`f_cost`).
     - `g_cost`: This dictionary keeps track of the cost of reaching each node from the start node.
     - `f_cost`: This dictionary stores the total estimated cost of reaching each node, which is the sum of the actual cost (`g_cost`) and the heuristic estimate (`heuristic`).
     - `came_from`: This dictionary is used to reconstruct the final path by keeping track of the previous node that led to each node.
2. **Main Loop**:
   - The function enters a loop that continues until the `queue` is empty.
   - In each iteration, the node with the lowest total cost (`f_cost`) is extracted from the `queue`.
   - If the extracted node is the goal node, the function reconstructs and returns the path from the start node to the goal node.
3. **Path Reconstruction**:
   - If the goal node is reached, the function enters a path reconstruction loop.
   - Starting from the goal node, the function iteratively traces back to the start node by following the `came_from` dictionary.
   - Each node encountered during this process is added to the `path` list.
4. **Exploring Neighbors**:

- If the goal node is not reached, the function explores the neighbors of the current node.
- For each neighbor, it calculates a tentative cost (`tentative_g_cost`) to reach that neighbor by adding the cost of the current node to the length of the edge connecting them.
- If the neighbor has not been visited before or if the tentative cost is lower than the previously recorded cost, the function updates the cost and priority of the neighbor node.
- The neighbor node and its updated total cost are then added to the `queue` for further exploration.

5. **No Path Found**:
    - If the `queue` becomes empty without reaching the goal node, it means that no path exists between the start and goal nodes. In this case, the function returns `None`.

5 Example Usage

To demonstrate the effectiveness of our implementation, we present an example scenario in San Francisco, California, USA.

5.1 Define Place and Download Street Network

We define the city of San Francisco as our area of interest and download its street network graph using OSMnx.

5.2 Define Start and End Points

We select the San Francisco City Hall as our starting point and Golden Gate Park as our destination. We find the nearest graph nodes corresponding to these locations.

5.3 Run A* Algorithm

We run our implemented A* algorithm to find the shortest path between the start and end nodes in the street network graph.

5.4 Plot Route on Map

Finally, we visualize the shortest path route on a map using Folium, a Python library for creating interactive maps.

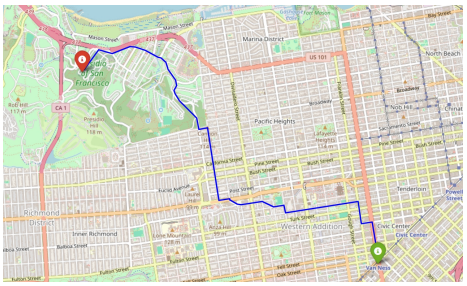

Figure 1: Shortest Path

6 Conclusion

In this paper, we presented a detailed implementation of the A* algorithm for finding the shortest path in geospatial data. By leveraging Python libraries such as OSMnx and NetworkX, we demonstrated how to extract map data, calculate distances, and efficiently find paths between locations on real-world maps. Our implementation provides a powerful tool for various applications requiring pathfinding on geographic datasets.