Dijkstra's Algorithm Implementation

=====================================

Introduction

----

jkstra's algorithm is a well-known graph search algorithm used to find the shortest

Implementation

----

aph Representati

will use an adjacency list to represent the graph, where each node is associated w

de and Edge Manageme

will use the NetworkX library to add, remove, and update nodes and edges in the g

ortest Path Calculati

will use Dijkstra's algorithm to calculate the shortest path between two node

th Visualizati

will use Matplotlib to visualize the shortest path between two node

Code

--

`pyth

port networkx as

port matplotlib.pyplot as p

port hea

ass Grap

f __init__(self

lf.G = nx.DiGraph

f add_node(self, node_id

lf.G.add_node(node_i

f add_edge(self, node1, node2, weight

lf.G.add_edge(node1, node2, weight=weigh

f dijkstra(self, start_node, end_node

itialize distances and previous nod

stances = {node: float('inf') for node in self.G.nodes(

evious_nodes = {node: None for node in self.G.nodes(

stances[start_node] =

iority que

iority_queue = [(0, start_node

ile priority_queu

rrent_distance, current_node = heapq.heappop(priority_queu

ip if current distance is greater than already known distan

current_distance > distances[current_node

ntin

erate over neighbo

r neighbor in self.G.neighbors(current_node

ight = self.G[current_node][neighbor]['weight

stance = current_distance + weig

date distance and previous node if shorter path is fou

distance < distances[neighbor

stances[neighbor] = distan

evious_nodes[neighbor] = current_no

apq.heappush(priority_queue, (distance, neighbor

ild shortest pa

th =

rrent_node = end_no

ile current_node is not Non

th.append(current_nod

rrent_node = previous_nodes[current_nod

th.reverse

turn distances[end_node], pa

f visualize(self, path

s = nx.spring_layout(self.

.draw(self.G, pos, with_labels=True, node_color='lightblue

.draw_networkx_edges(self.G, pos, edgelist=[(path[i], path[i+1]) for i in range(len(p

t.show

ample usa

aph = Graph

aph.add_node(

aph.add_node(

aph.add_node(

aph.add_edge(1, 2,

aph.add_edge(1, 3,

aph.add_edge(2, 3,

stance, path = graph.dijkstra(1,

int(f"Shortest distance: {distance}

int(f"Shortest path: {path}

aph.visualize(pat

Algorithm

Initialize distances and previous nodes for all nodes in the grap

Set the distance of the start node to 0 and add it to the priority queu

While the priority queue is not empt

Dequeue the node with the smallest distanc

If the current distance is greater than the already known distance, skip this nod

Iterate over the neighbors of the current nod

Calculate the tentative distance to the neighbo

If the calculated distance is smaller than the already known distance, update the d

Add the neighbor to the priority queu

Build the shortest path by backtracking from the end node to the start nod

Time Complexity

----

e time complexity of Dijkstra's algorithm is O(|E|log|V|) in the worst case, where |E|

Space Complexity

----

e space complexity of Dijkstra's algorithm is O(|V| + |E|), where |V| is the number of

te: This implementation assumes that the graph is represented as an adjacency list

Generated on 2025-04-06 at 12:41:43