Dijkstra Algorithm Visualization

Aisana Sembek, Assan Assemov, Bolashak Kulmukhambetov, Timur Chuiko, Yerzhan Amangeldin

1. Project Description

This project provides a visual and interactive simulation of Dijkstra's algorithm for finding the shortest paths in a weighted graph. It is developed using the Pygame library in Python, offering a user-friendly graphical interface where users can manually construct graphs by adding nodes and edges, assign weights to edges, and run the algorithm to observe how it calculates the shortest path from a selected starting node to all other nodes.

The visualization is designed to run step-by-step, highlighting which nodes are being visited, how distances are updated, and which edges form the final shortest path tree. It uses a priority queue to efficiently select the next node with the minimum tentative distance, following the standard Dijkstra algorithm logic.

The program includes various user controls for adding, deleting, and connecting nodes, starting and pausing the algorithm, and viewing real-time updates to the distance table. These features collectively make the tool suitable for classroom demonstrations, individual exploration, and algorithm debugging or learning.

2. Project Goals

The main objective of this project is to provide an educational tool that simplifies the understanding of Dijkstra's algorithm through interactive visualization. The specific goals include:

- **Enhance Conceptual Understanding:** Enable users to visually grasp how Dijkstra's algorithm processes nodes and updates shortest paths, bridging the gap between theoretical knowledge and practical application.
- **Encourage Interactive Learning:** Allow users to actively build graphs, assign weights, and follow the algorithm's decision-making process step by step, fostering engagement and deeper comprehension.
- **Support Algorithm Education:** Serve as a supportive tool for educators and students in computer science courses to demonstrate and explore graph algorithms in real time.
- **Provide Clear Visual Feedback:** Use distinct colors and animations to differentiate visited nodes, current nodes, finalized paths, and distance updates, ensuring clarity during algorithm execution.

- **Offer Practical Implementation Experience:** Introduce users to how graph-based algorithms are implemented in a programming environment, particularly using Python and Pygame, encouraging further exploration of data structures and algorithm development.

3. Technologies Used

- Python 3.10
- Pygame Library
- heapq Module (for priority queue)
- Data Structures:
  - Dictionaries
  - Lists
  - Sets
  - Tuples
- RGB Color Coding for Visualization
- Event-Driven Programming
- Custom Implementation of Dijkstra's Algorithm
- Modular Code Structure

4. Features

- **Interactive Graph Creation:** Users can dynamically create nodes and connect them with weighted edges using mouse clicks, enabling full control over the graph structure.
- **Edge Weight Assignment:** When connecting nodes, users can input custom weights for each edge, which are then displayed on the screen for clarity.
- **Dijkstra's Algorithm Visualization:** The shortest path calculation is animated step by step, showing visited nodes, current processing node, updated distances, and finalized paths.
- **Color-Coded Visualization:** Different colors represent the state of nodes and edges:
  - Grey – Default/unvisited nodes
  - Blue – Current node being processed
  - Pink – Node added to the shortest path tree
  - Orange – Start node
- **Distance Table Display:** Real-time updates of each node's current shortest distance from the source node are shown during execution.
- **Reset and Clear Options:** Users can reset the algorithm state or clear the entire graph to start fresh without restarting the application.
- **Keyboard Shortcuts:** Various functions are triggered by keyboard commands, such as starting the algorithm, clearing nodes, or deleting edges.
- **Error Handling:** The application includes basic error checks, such as preventing duplicate edges or operations on incomplete nodes.
- **Simple and Intuitive UI:** The layout is designed to be user-friendly and minimalistic, focusing on usability and clarity for educational purposes.

5. How to Use

- **Launching the Application:** Run the script to open the graphical window. The user interface will display a canvas along with interactive control buttons.
- **Creating Nodes:** Left-click anywhere on the canvas to place a new node. Each node is automatically numbered for identification.
- **Adding Edges:** Click on two different nodes in sequence. A popup will prompt the user to input a weight for the edge connecting them.
- **Editing the Graph:** Press the **"Edit Graph"** button to switch into edit mode, allowing you to add or remove nodes and edges.
- **Setting the Start Node:** Use the **"Change Start"** button, then click on the node you want to designate as the starting point for Dijkstra's algorithm.
- **Visualizing the Algorithm:** Click the **"Visual"** button to initiate the step-by-step visualization of Dijkstra's algorithm from the selected start node.
- **Controlling the Animation:**
  - **"Next"** – Proceed to the next step in the algorithm.
  - **"Previous"** – Go back one step to review the previous state.
  - **"Pause"** – Temporarily halt the visualization.
- **Exiting the Program:** Click the window's close button to exit the application safely.

6. Code Structure

The project consists of multiple functions and methods responsible for visualizing and interacting with a graph during the execution of Dijkstra's algorithm. Below is a detailed overview of the core components:

- `main` — Entry point of the application. Initializes the environment, creates the main loop, and manages the application's runtime behavior.
- `GraphVisualizer` — Main class managing state, drawing, interactions, and control flow of the graph and algorithm visualization.
- `reset_graph()` — Resets the graph state, clearing all distances, paths, and visited information to allow a fresh run of the algorithm.
- `update_steps()` — Advances the visualization by one step, updating node and edge states based on algorithm progression.
- `dijkstra_steps(graph, start)` — Generator function implementing Dijkstra's algorithm. Yields each step to enable animated visualization.
- `add_node(pos)` — Adds a new node at the specified position.
- `add_edge(start_node, end_node, weight)` — Adds a new weighted edge between two nodes.
- `update_edge_weight(edge, new_weight)` — Modifies the weight of a given edge.
- `draw_graph()` — Renders all graph elements, including nodes, edges, weights, and highlights.

- `draw_edges()` — (if present) Used internally to render edges with weight labels and directional arrows.
- `draw_instructions_table()` — Draws a side panel with user instructions and usage tips.
- `draw_dijkstra_table()` — Visualizes Dijkstra's distance table, showing current shortest distances and paths from the start node.
- `draw_adjacency_list()` — Displays the graph's adjacency list in text format.
- `draw_controls()` — Renders interactive buttons like "Edit Graph," "Change Start," "Visual," "Pause," "Next," and "Previous."
- `draw_legend()` — Renders a color legend to help users interpret node/edge states (e.g., visited, processing, start, end).
- `draw_message(text)` — Displays temporary messages or status updates to guide the user.
- `handle_click(pos)` — Processes mouse click events for node/edge interaction, selection, and graph editing.
- `handle_edit_click()` — Activates edge weight editing mode and processes input accordingly.
- `handle_start_node_selection()` — Allows user to change the start node before running the algorithm.
- `edit_edge_weight(edge)` — Triggers an input mechanism for modifying the selected edge's weight.
- `point_near_line(p, a, b)` — Utility function to detect whether a click is close to a line segment (edge), used for edge selection.
- `update_visited_sets()` — Updates visual states of visited and unvisited nodes as the algorithm progresses.
- `update()` — Core update logic for refreshing the application state between frames.
- `draw()` — Draws the entire interface, including graph, UI elements, tables, and background.
- `draw_button()` — Helper function to draw individual UI buttons with labels and states.
- `draw_background_gradient()` — Draws a gradient background to improve visual appeal.