A* Algorithm Visualization

name: Su Hyun Kim
SJSU ID: 018219422
class #: CMPE-252 Sec 02
source code link: github link

Introduction

This report details the implementation and visualization of Dijkstra's algorithm and multiple versions of the A* algorithm for finding the shortest path between two nodes in a graph. The visualization component generates images of each step of the algorithms and can optionally create a video showcasing the execution of all algorithms simultaneously. The output includes the shortest paths, distances, and visual representations for each algorithm.

Implementation Overview

Language and Tools

- **Programming Language**: Python 3.10
- **Libraries Used**:
  - `matplotlib` for visualization
  - `opencv-python` for video generation
  - `argparse` for command-line argument parsing
  - `collections`, `heapq`, and `os` for data handling
  - `numpy` for numerical operations
  - `tqdm` for progress bar

Files Structure

- **Main Script**: `main.py`
- **Input Files**:
  - `input.txt`: Contains graph edges and nodes
  - `coords.txt`: Contains coordinates for each node
- **Output Files**:
  - `<SJSU_ID>.txt`: Contains the shortest path and distances
  - `<SJSU_ID>.mp4`: Video of the algorithm's execution

Algorithm Implementation

Data Processing

- **Graph Representation**:
    - The graph is represented using a dictionary (`defaultdict(list)`) where each key is a node, and the value is a list of tuples representing connected nodes (neighbors) and the distance to them.
- **Node Information**:
    - Node coordinates are stored in a list (`node_info`), where the index corresponds to the node number. (0th index have no meaning)

Algorithms

The script implements Dijkstra's algorithm and multiple versions of the A* algorithm with different epsilon values (1 to 5).

Common Components

- **Priority Queue**:
    - Uses a min-heap (`heapq`) as a priority queue to always select the node with the smallest cumulative distance.
- **Distance Tracking**:
    - Maintains a dictionary (`min_distance_from_start`) to keep track of the minimum distance to each node from the start node.
- **Path Reconstruction**:
    - Keeps track of the path and distance history to reconstruct the shortest path at the end of the algorithm.

Dijkstra's Algorithm

- **Heuristic Function:**
    - Dijkstra's algorithm does not use a heuristic function; it relies solely on the actual distances from the start node.

A* Algorithm Variants

- **Heuristic Function**:
    - Uses the Euclidean distance between the current node and the end node as the heuristic function.
- **Epsilon Values:**
    - The script runs A* algorithm with epsilon value from 1 to 5.
    - The epsilon value scales the heuristic to influence the search behavior.

Visualization

- **Image Generation**:

    - Generates images at each significant step of algorithms, showing the nodes being processed.
    - Uses Matplotlib to plot the graph, nodes, and edges.
    - Nodes are color-coded:
        - **Green**: Start node
        - **Red**: End node
        - **Blue**: Nodes currently being processed
        - **Gray**: Nodes that have been fully processed
- **Final Path Visualization**:
    - After finding the shortest path for each algorithm, generates a final image highlighting the path in red for each algorithm.

Video Generation

- **Using OpenCV**:
    - Frames are created in memory using `BytesIO`, avoiding the need to save images to local disk.
    - Images from each algorithm are combined into frames and compiled into a video.
    - Video is saved in MP4 format using the `mp4v` codec.
- **Command-Line Control**:
    - Video generation can be toggled on or off using command-line arguments.

Instructions to Compile and Run the Code

Prerequisites

- **Python 3.10** or higher (Python **3.10.5** preferred)
- **Required Python Libraries**:
    - `matplotlib`
    - `opencv-python`
    - `numpy` (dependency of `matplotlib` and `opencv-python`)
    - `tqdm`

Setting Up the Environment

Using `requirements.txt`

1. **Create a Virtual Environment** (optional but recommended):
   `python -m venv venv`
2. **Activate the Virtual Environment**:
    - **Windows**: `venv\Scripts\activate`

    - **macOS/Linux**: `source venv/bin/activate`
3. **Install Dependencies**: `pip install -r requirements.txt`

Using Poetry (Optional)

1. **Install Poetry**: `pip install poetry`
2. **Install Dependencies**: `poetry install`
3. **Activate the Poetry Shell**: `poetry shell`

Preparing Input Files

Ensure that `input.txt` and `coords.txt` are placed in the same directory as `main.py`.

Running the Script

Open a terminal in the directory containing `main.py` and the input files.

Command-Line Arguments

- **`--video`**: Control video generation. (default = 1)
    - `1` to generate the video (default).
    - `0` to skip video generation.
- **`--steps_per_frame`**: Adjust the number of algorithm steps included in each video frame. (default = 3)
    - Higher values speed up the video by including more steps per frame.

Examples

- `python main.py --video 1 --steps_per_frame 5`

Code Explanation

Structure

Algorithm Classes

- **Base Class**: `Algorithm`
    - Contains common properties and methods used by both Dijkstra's algorithm and A* algorithms.
- **DijkstraAlgorithm Class**:
    - Inherits from `Algorithm`.

    - Implements Dijkstra's algorithm without a heuristic function.
- **AStarAlgorithm Class**:
    - Inherits from `Algorithm`.
    - Implements the A* algorithm with varying epsilon values.
    - The heuristic function used is the Euclidean distance between the current node and the end node.

DataProcessor Class

- **Initialization**:
    - Accepts parameters to control video generation and steps per frame.
    - Initializes variables for graph and node information.
- **Methods**:
    - `process_input_files`: Reads and processes input files.
    - `initialize_algorithms`: Sets up instances of the algorithms to be run.
    - `run_algorithms`: Executes all algorithms concurrently, step by step.
    - `generate_base_graph_image`: Creates the base graph image used in visualizations for each algorithm.
    - `generate_combined_image_for_step`: Generates combined images for all algorithms at each step.
    - `generate_final_images`: Generates the final images highlighting the shortest paths for each algorithm.
    - `generate_output_file`: Writes the shortest paths and distances for each algorithm to a text file.
    - `generate_video_from_frames`: Compiles frames into a video using OpenCV.
    - `main`: Orchestrates the execution of all methods.

Command-Line Argument Parsing:

- Uses `argparse` to handle command-line arguments for video generation and frame rate control.

Key Components

- **Visualization Logic**:
    - Nodes are plotted using Matplotlib.
    - Colors indicate the state of nodes during the algorithm.
    - Images are saved at specified intervals based on `steps_per_frame`.
- **Algorithm Logic**:
    - Uses a priority queue to select the next node with the smallest cumulative distance.
    - Keep track of the minimum distance to each node to avoid unnecessary processing.

Conclusion

The implemented script successfully calculates the shortest paths using Dijkstra's algorithm and multiple versions of the A* algorithm. It provides visual insights into the execution of each algorithm and allows for direct comparison through combined visualizations.
