Hw1 - Dijkstra's Algorithm

Student Name : Chen Kai Zhang

Student ID : 014806701

Running Code

In this section we will go over how to setup and run the script.

Requirements

- Python 3.11 +
- `input.txt`
- `coords.txt`

Setup

1. Create a new virtual environment with `python -m venv venv` and activate it via `source venv\bin\activate`.
2. Install the required packages via `pip install -r requirements.txt`

Execution

The script is designed to run as an executable.

In the command line, run `python 014806701.py -i <input.txt> -c <coords.txt> -s <start_node> -e <end_node>`

- For more clearance, run `python 014806701.py -h`

If a visualization of the process is desired, simply add `-v` flag to the end of the executable. Setting the flag will show an animation and save it as an `mp4`.

Executing the script will generate a new file `014806701.txt` which contains two lines

1. The shortest path
2. The total weights along the shortest path

Project Overview

In this section we will breakdown the source code.

All Dijkstra related functionality are organized into a class called `Dijkstra`. This class includes 3 functions:

1. `compute`
2. `visualize`
3. `save`

`compute`

This is the main Dijkstra algorithm. At a high level it follows three steps

1. Start at `start` and explores the graph by picking unvisited nodes with the smallest distance
2. Updates the distances to neighbors based on the current node's distance
3. Stops when all nodes are visited or it reached the end

Below is the breakdown of the source code.

Initialization

```
distances = {node: float("infinity") for node in self.graph}
visited = set()
previous = {node: None for node in self.graph}

distances[start] = 0
```

... where

`distances` : stores the shortest known distance to each node

`visited` : keeps track of visited node to avoid revisiting

`previous` : stores the previous node for each node in the shortest path

- All values in `distances` are set to infinity because we have yet to discover shortest distance to each node.
- `distance[start] = 0` because the distance from `start` to `start` is 0

Exploring Next Node

```
while len(visited) < len(self.graph):
    min_distance = float("infinity")
    current = None

    for node in self.graph:
        if node not in visited and distances[node] < min_distance:
            min_distance = distances[node]
            current = node

    if current is None:
        break

    if current == end:
        break

    visited.add(current)
```

This loop will run until all nodes in the graph have been visited or we break early under

- Hit the target node
- No unvisited nodes with a finite distance remains

On every iteration, we will choose a non visited node with the shortest known distance

Updating Discoveries

```
for neighbor, weight in self.graph[current].items():
    if neighbor not in visited:
        new_distance = distances[current] + weight
        if new_distance < distances[neighbor]:
            distances[neighbor] = new_distance
            previous[neighbor] = current
```

This section updates the shortest distance from the `start` node to each neighbor of the `current` node. The process to make this happen is as follows:

- Calculate a potential new distance from `start` through `current` to a neighbor
- If the new distance is less than the current known distance to the neighbor:
    1. Update the new cost to that neighbor
    2. Track the `current` that led to that low cost in `previous`

Building Results

```
path = []
path_cost = []
current = end

while current is not None:
    path.append(current)
    path_cost.append((current, distances[current]))
    current = previous[current]

return path[::-1], path_cost[::-1]
```

After all necessary nodes have been explored, the code will build a path from `start` to `end` by backtracking its steps.

- Remember, `previous` is a dictionary where the keys are visited nodes and the values are it neighbors, the algorithm traversed discovered with the lowest cost

Building shortest path is simply building a list by going through `previous` from `end` to `start` in reverse and the cost along the path is simply reading the cost from `distances` and return that list in reverse.

`visualize`

This code is a duplicate of my Dijkstra algorithm code, but `networkx` drawing function placed in between every step during the **exploration next node** phase and **building results** phase.

`save`

Saves the results returned by `compute` into `014806701.txt` where lines :

1. The shortest path from `start` to `end`
2. The cost per step from the shortest path