Peer-review rapport Lab3

Fortfattaren : Othman Belal
Laborationen : Rasmus Bäckström

Section 1: Core assignment

Q1: Does the application run?
Yes, the application runs successfully.

Q2: Does the application display the complete map of tram lines?
Yes, the application displays the complete map of tram lines.

Q3: Is it possible to query the shortest path between any two points?
Yes, it is possible to query the shortest path between any two points.

Section 2: Optional tasks

B1: Is the submission successfully accounting for Bonus Part 1?
Yes/No (Provide relevant details)
Rasmus has not addressed bonuses in his lab.

B2: Is the submission successfully accounting for Bonus Part 2?
Yes/No (Provide relevant details)
Rasmus has not addressed bonuses in his lab.

Section 3: Code quality

Let's analyze the provided code in terms of code quality and address the specified points:

1. Reuse of Code from Lab 2:
   - The code from Lab 2 seems to be reused effectively. The `**Graph**` and `**WeightedGraph**` classes are defined in the `graph` code, and they are imported and extended in the `**tramvis**` code.
   - The `**Graph**` class is used as a base class for the `**WeightedGraph**` class, demonstrating proper inheritance.
   - The code appears to avoid boilerplate and redundancies, making efficient use of the previously defined graph-related functionality.

2. Implementation and Usage of **Dijkstra's** Algorithm:
   - Dijkstra's algorithm is implemented in the `**dijkstra**` function within the `graph` code.
   - The function takes into account different cost functions through the use of a lambda function, allowing flexibility in calculating distances based on different criteria (e.g., time, distance).
   - There is only one definition of the `**dijkstra**` function, and different distances are obtained by changing the cost function.
#Exampel from graph code (graph.py)

```
def dijkstra(source, graph=None, cost=None):
        # dijkstra function implementation …..
```
 # Example from tramvis code (tramvis.py)
```
network = TramGraph(start=[("A", "B"), ("B", "C")], directed=True)
shortest_path_dist = dijkstra("A", graph=network, cost=lambda u, v:
distance_between_stops(network.stops, u, v))
print("Shortest Path based on Distance:", shortest_path_dist["B"]["path"])

# Calculate shortest path based on transition time
shortest_path_time = dijkstra("A", graph=network, cost=lambda u, v:
network.transition_time(u, v))
print("Shortest Path based on Transition Time:", shortest_path_time["B"]["path"])
```

- The function correctly calculates the shortest path and associated distances for both weighted and unweighted graphs.

examples from the code regarding unweighted graph:
```
 # Assuming we have an instance of the Graph class
graph_instance = Graph(start=[(1, 2), (2, 3), (1, 3)], directed=True)
shortest_path_unweighted = dijkstra(1, graph=graph_instance)
print("Shortest Path in Unweighted Graph:", shortest_path_unweighted[3]["path"])
```
examples from the code regarding weighted graph:

 # Example from graph code (graph.py)
```
# Assuming we have an instance of the WeightedGraph class
weighted_graph_instance = WeightedGraph()

# Adding edges and weights to the graph
weighted_graph_instance.add_edge(1, 2)
weighted_graph_instance.set_weight(1, 2, 5)  # Assigning a weight of 5 to the edge (1, 2)

# Calculate shortest path in the weighted graph
shortest_path_weighted = dijkstra(1, graph=weighted_graph_instance, cost=lambda u, v:
weighted_graph_instance.get_weight(u, v))
print("Shortest Path in Weighted Graph:", shortest_path_weighted[2]["path"])
```

Overall Code Quality:

- The code is organized and modular, with clear class definitions and function implementations.
- Meaningful variable and function names enhance code making it eazy to read, understand and analyze, which i did not put much attention on it in my lab.
- Proper use of comments helps to explain the purpose and functionality of different sections of the code.

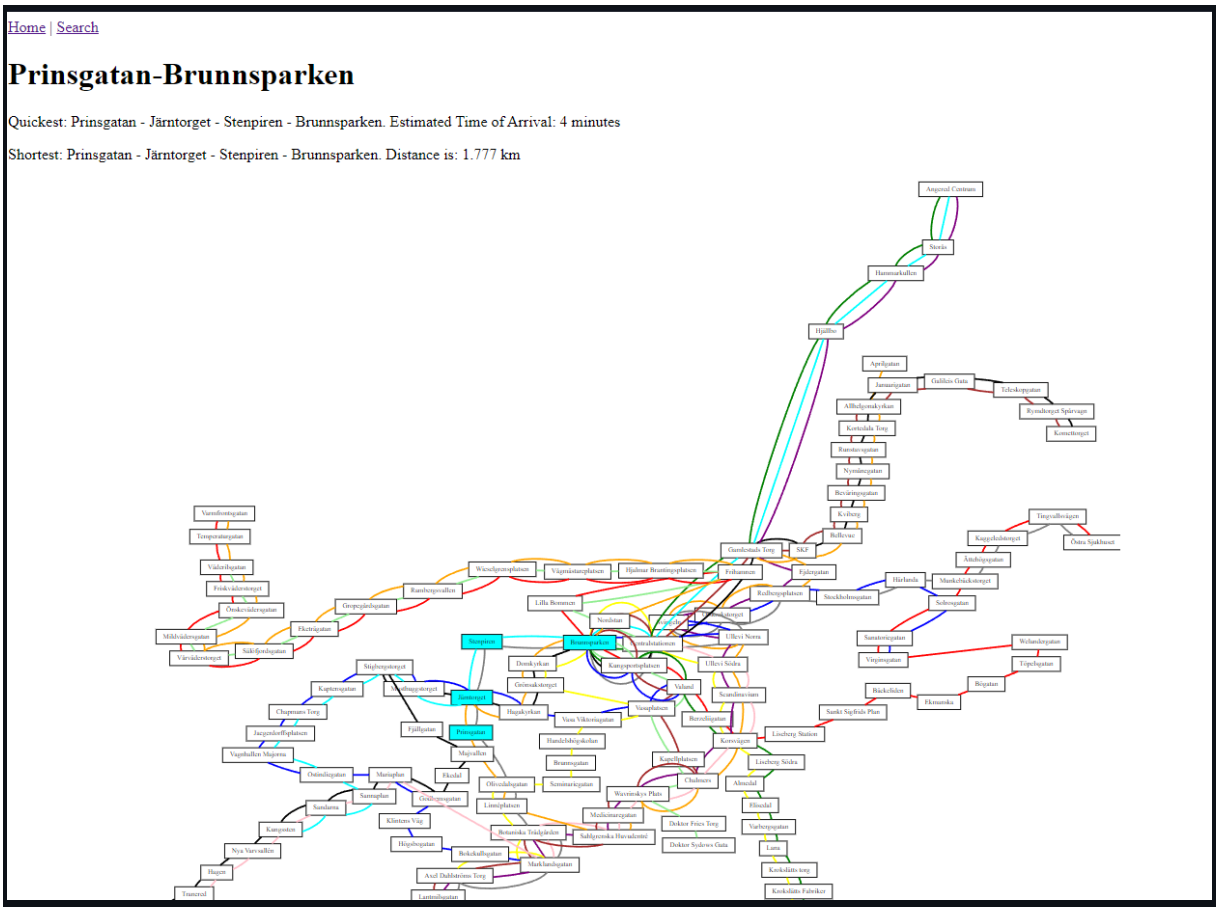
Suggestions for Improvement:

- Consider providing more detailed comments or docstrings, especially for complex functions or methods, to enhance code documentation.

In summary, the code appears to be of good quality, with effective reuse of code from Lab 2 and proper implementation and usage of Dijkstra's algorithm.

Section 4: Screenshots:

Screenshot 1, it shows the Quickest and shortest way between the stations Prinsgatan-Brunnsparken



Second Screenshot present the code of the function show_shortest() (the main function required in the core part of the assignment:

```python
def show_shortest(dep, dest):
    # TODO: uncomment this when it works with your own code
    network = readTramNetwork()


    # TODO: replace this mock-up with actual computation using dijkstra.
    # First you need to calculate the shortest and quickest paths, by using appropriate
    # cost functions in dijkstra().
    # Then you just need to use the lists of stops returned by dijkstra()
    #
    # If you do Bonus 1, you could also tell which tram lines you use and where changes
    # happen. But since this was not mentioned in lab3.md, it is not compulsory.
#----------------------- without bonus

    dijk_dist = dijkstra(dep, network, cost = lambda u,v :distance_between_stops(network.stops, u , v))
    short_distance = round(dijk_dist[dest]["distance"],3)
    short_dist_path = dijk_dist[dest]["path"]
    dijk_time = dijkstra(dep, network,  cost = lambda u,v : network.transition_time( u, v))
    quick_time = dijk_time[dest]["distance"]
    quick_time_path = dijk_time[dest]["path"]

    #----------------------- with bonus
    spec_network = specialize_stops_to_lines(network)
    #short_distance = round(dijkstra(dep, network, cost = lambda u,v : distance_between_stops(network.stops, u , v))[dest]["distance"],2)
    #short_dist_path = dijkstra(dep, network, cost = lambda u,v : distance_between_stops(network.stops, u , v))[dest]["path"]
    #quick_time = dijkstra(dep, network,  cost = lambda u,v : network.get_weight(u, v))[dest]["distance"]
    #quick_time_path = dijkstra(dep, network,  cost = lambda u,v : network.get_weight(u, v))[dest]["path"]

    #quickest = [dep, 'Chalmers', dest]
    #shortest = [dep, 'Chalmers', dest]


    #----------------------------------------------------------------
    timepath = 'Quickest: ' + ' - '.join(quick_time_path) + "."+ " Estimated Time of Arrival:" + f" {quick_time} minutes"   #, 5 minutes'
    geopath = 'Shortest: ' + ' - '.join(short_dist_path) + "." +  f" Distance is: {short_distance} km"  # ', 100 km'
```