iems
intelligente eingebettete mikrosysteme

Bioinformatics Group
Prof. Dr. Rolf Backofen
Florian Eggenhofer
Michael Uhl
Rick Gelhausen

Entwurf, Analyse und Umsetzung von Algorithmen

UNI FREIBURG

Exercise sheet 13

In this exercise you will compute the largest connected component and implement Dijkstra's Algorithm to find the fastest (shortest) routes in the street graphs of Freiburg. **Read the exercise sheet first before starting to write code.**

Exercise 1 (5 points) *Largest Connected Component*
On the website you can find the *graph.py* file containing the `Graph` class and some already implemented methods (e.g. for reading in the *.graph* file). Have a look at the already supplied functionalities. Based on these, implement the method `compute_lcc` which calculates all connected components and marks the nodes in the largest connected component (= those with the highest count of marked nodes). For this you can utilize the already implemented method `compute_reachable_nodes` inside the class for the computation of reachable nodes for a given node. Be aware that you might need to adapt the output of the method for this exercise. Also make sure to pass `False` as second parameter to `read_graph_from_file` to load the graph with undirected edges.

Exercise 2 (10 points) *Dijkstra's Algorithm*
Extend the `Graph` class with the method `compute_shortest_paths()`. This method will use Dijkstra's algorithm to calculate the cost of all shortest paths from a given node to all reachable nodes. For this make use of the variable `costs` that is saved in the class `Arc`. To calculate the shortest path extend your class `Node` with the member variable `traceback_arc`, which stores the edge that was used to calculate the lowest costs for this particular node (i.e., to reconstruct the shortest path).

Exercise 3 (5 points) *Route Planner*
4.1) Write a short program *route_planner.py*, which calculates the route between the Faculty of Engineering (node ID 95466) and the Schwarzwald-Stadion Freiburg (node ID 136096) regarding the following aspects:

1. Shortest path (max. speed 300 km/h)
2. Fastest traveling time by car (max. speed 130 km/h)
3. Fastest traveling time by moped (max. speed 50 km/h)

The necessary graph data *freiburg.graph* can found on the course website. Load the data by passing `True` as second parameter to `read_graph_from_file` to load the graph with directed edges. To switch between shortest path and fastest traveling time, set the edge costs by using the methods `set_arc_costs_to_distance()` and `set_arc_costs_to_travel_time()` included in `graph.py`. Provide the distance in km, the travel time in hours and minutes, and the computation time in seconds and milliseconds (excluding reading in the graph).

In addition, you can convert the 3 routes into the *MapBBCode* format to visualize them (example see below).

4.2) Find the node in the graph which is furthest away from the technical faculty in regard of the travel time per car and moped. Please provide again the distance in km, the travel time in hours and minutes, and the computation time (and **optionally** the *MapBBCode*).

Example *MapBBCode*:

```
[map]
  <lat11>,<lon11> <lat12>,<lon12> ... <lat1n>,<lon1n>(blue|label1);
  <lat21>,<lon21> <lat22>,<lon22> ... <lat2n>,<lon2n>(red|label2);
  <lat31>,<lon31> <lat32>,<lon32> ... <lat3n>,<lon3n>(green|label3)
[/map]

Example:
[map]47.7811,8.34618 47.7812,8.34682(blue|Some route)[/map]
```

Every route consists of a pairs of coordinates *<latitude>,<longitude>* with each pair separated by a comma. Coordinate pairs are separated by a space character. Routes are separated by a semicolon (not after the last or only one). After the last coordinate pair, optional parameters can be given, e.g. in the form of *(color|label)*, which provides colors and labels to the routes. Mark route 1 as blue, route 2 as red and route 3 as green. The generated code can be visualized on *http://share.mapbbcode.org/*.