DAT171 - Computer assignment 1

(2023-01-17)

Note on collaboration: These assignments should be done in groups of two. Naturally, you are encouraged to discuss the problem with your classmates, but it is important that you **do not send or receive entire chunks of code between groups**.

The goal of the first computer assignment is to get you comfortable in reading in data from a text file and using the NumPy, SciPy, and Matplotlib libraries. The task is to construct a graph of neighbouring cities and to find the shortest path between two given cities, through the neighbouring cities. **See the example figure at the end of this document.**

Three files are supplied to test your program, with one very small file, `SampleCoordinates.txt`, suitable for testing your functions. In task 3 you need a radius that limits what is considered neighbouring cities, in task 6 you need to enter start and end cities. You can find these values in the table below (the city no. corresponds to the line in the respective city file).

Filename	Radius	Start city	End city
`SampleCoordinates.txt`	0.08	0	5
`HungaryCities.txt`	0.005	311	702
`GermanyCities.txt`	0.0025	31	2

For convenience, the task is divided into steps as listed below. You are required to stick to the given function prototypes, you are of course allowed to write more sub-functions if you want. *Functions must be documented using PyDoc.*

1. Create the function `read_coordinate_file(filename)` that reads the given coordinate file and parses the results into an array of coordinates. The coordinates are expressed in the format: `{a, b}` where `a` is the latitude and `b` is the longitude (in degrees). Convert these using the Mercator projection to obtain the coordinates in xy-format:

   $$x = R\frac{\pi b}{180}, \quad y = R\ln\left(\tan\left(\frac{\pi}{4} + \frac{\pi a}{360}\right)\right)$$

   The radii in the table are given for the normalised $R = 1$. Return a 2D `ndarray` with the x and y values.

   Hints: Use the function `split` for the strings. Convert a string to a number with the function `float`.

2. Create the function `plot_points(coord_list)` that plots the data points read from the file. Make sure you use a correct aspect ratio so that the countries look similar in your plot as they do on a map.

   Hints: Remember to call `plt.show()` after plot commands are finished.

3. Create the function `construct_graph_connections(coord_list, radius)` that computes all the connections between all the points in `coord_list` that are within the radius given. Simply check each coordinate against all other coordinates to see if they are within the given radius using two nested loops. Return one `ndarray` with the indices of each two connected cities and another `ndarray` with the distance between them.

4. Create the function `construct_graph(indices, distance, N)` that constructs a sparse graph. The graph should be represented as a compressed sparse row matrix (`csr_matrix` in `scipy.sparse`). Each element at index `[i, j]` in the matrix should be the distance between city `i` and `j`. Construct the matrix with `csr_matrix((data, ij), shape=(M, N))`.

   Hints: You need to provide a size N as input to this function. What is N? The SciPy manual contains examples on how to create the matrices: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

5. Extend `plot_points(coord_list, indices)` to also include the graph connections from task 3.

   Hints: Use Matplotlib:s `LineCollection` instead of plotting the lines individually, since it is much faster.

6. Create the function `find_shortest_path(graph, start_node, end_node)` that returns the shortest paths from `start_node` to `end_node` as well as its total distance. Use the functions in `scipy.sparse.csgraph` (https://docs.scipy.org/doc/scipy/reference/sparse.csgraph.html). One of the outputs from the shortest path function in SciPy is a “predecessor matrix”. The columns represent the predecessor when taking the shortest

path to the given column index (this seems complicated, but is actually a clever way to store the shortest paths to every possible end node). Please make sure you properly document this function!

Hints: The function `shortest_path` finds the shortest path, it lets the user input what indices (starting nodes) the path should be computed for. This saves significant computational effort! The file `SampleCoordinatex.txt` should generate the sequence `[0, 4, 3, 5]`, the total distance for this path is `0.16446989717973517`.

7. Extend `plot_points(coord_list, indices, path)` to also include the shortest path. Make the shortest path more visible by making it stand out (thicker line width and a noticeable colour for example).

8. Record the total time for each of the functions (see the table below for reference). Which routine consumes the most time? Do not add anything to your functions in this step, just time things at the places where you call the functions.

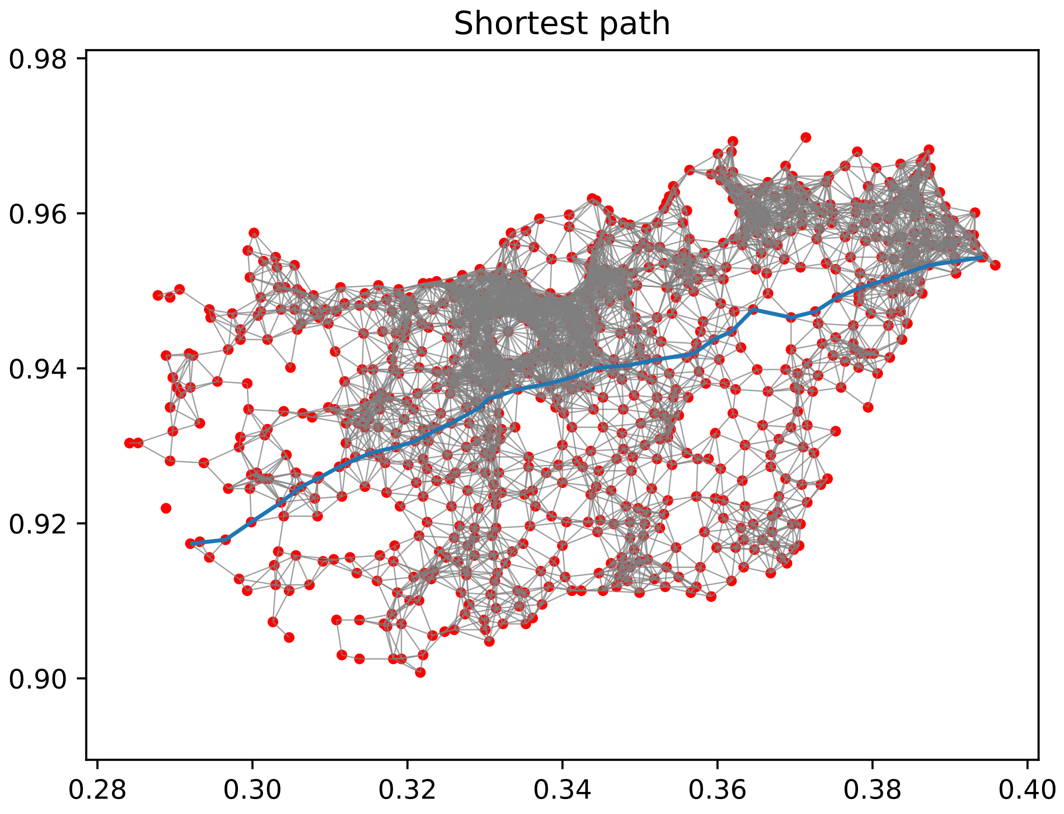
   Hints: Use `time.time()`. When timing `plot_points`, make sure that you time the computational cost of plotting, not the time until the plot is closed.

9. Create the function `construct_fast_graph_connections(coord_list, radius)` that computes all the connections between all the points in `coord_list` that are within the radius given. This time, use the `KDTree` from SciPy to find the closest coordinates quickly. **Note!** You must be able to swap `construct_graph_connections` with `construct_fast_graph_connections` in your code (without any other alterations)!

   Hints: Instead of checking a coordinate against all the other coordinates, we can filter the number of points by for example using the method `query_ball_points(coord, radius)` on the KDTree.

For reference, here is the expected output for `HungaryCities.txt`:

```
Shortest path from 311 to 702 is: [311, 19, 460, 629, 269, 236, 781, 50, 193, 571, 624, 402, 370,
153, 262, 554, 126, 251, 368, 221, 827, 300, 648, 253, 836, 73, 35, 219, 503, 789, 200, 702]
Total distance: 0.1114486118821533
```



Additional instructions

Besides the general *General instructions for computer assignments* please consider the items below for Computer Assignment 1:

1. Write a single main code. For choosing between the different files, define the following variables at the beginning of your main code:

   - `filename`
   - `start_city`
   - `end_city`
   - `radius`

   Parameterise the file for `GermanyCities.txt` when you hand in the code.

2. Reading the input file:

   - Read the file line by line, process each line as it is read.
   - Use `strip`, `split`, and so on for processing each line, not indexing or similar.
   - Remember to close files.

3. Do not append to NumPy-arrays in loops. This is computationally expensive, as it copies the whole array (see https://numpy.org/doc/stable/reference/generated/numpy.append.html). If you really need to append in a loop, use a python `list` instead and convert it to a NumPy-array once it is filled.

4. When using an element of a collection and its index in a loop, do

   ```
   for index, element in enumerate(collection):
       # do stuff
   ```

   instead of looping over the indices.

5. As part of the report, the resulting pictures must be handed in. Make sure the pictures have a correct aspect ratio.

6. The output of the program must include the total distance and the *shortest path* found.

7. Timing information for the major routines must be provided. On a quite old machine (Intel Core i7-2600) the following results (to one digit precision) can be used as a hint on the expected results for `GermanyCities.txt`:

function	time (s)
`read_coordinate_file`	0.03
`construct_graph_connections`	70
`construct_graph`	0.002
`find_shortest_path`	0.007
`plot_points` (from task 7, excluding plt.show)	2
`construct_fast_graph_connections`	0.3
Running the entire program using the fast version, excluding plotting	1

What should be handed in?

Your hand-in should contain the following:

- A working, documented code (preferably in one flat file) for all tasks above.
- Plots for all three input-files.
- The results (i.e. list of cities in the shortest path and total distance) for all three input-files.
- Timing information corresponding to the table above for Germany (at least).

Make sure to follow the "General instructions for Computer Assignments" before handing in.

Good luck!