Chapter 15: Graph algorithms II. - minimum spanning tree

Read the `hungary_cities.shp` shapefile located in the `data` folder. This dataset contains both scalar and spatial data of the Hungarian cities:

1. City Id
2. City Name
3. County Name
4. Status (town, city, independent city, national capital, capital district)
5. KSH code (unique statistical code for the city)

Source: *ELTE FI, Institute of Cartography and Geoinformatics*

In [1]:

```
import geopandas as gpd

cities = gpd.read_file('../data/hungary_cities.shp')
display(cities)
```

	Id	County	City	Status	KSH	geometry
0	1	FEJÉR	Aba	town	17376	POINT (610046.800 187639.000)
1	2	BARANYA	Abaliget	town	12548	POINT (577946.100 89280.800)
2	3	HEVES	Abasár	town	24554	POINT (721963.700 273880.300)
3	4	BORSOD-ABAUJ-ZEMPLÉN	Abaújalpár	town	15662	POINT (812129.200 331508.200)
4	5	BORSOD-ABAUJ-ZEMPLÉN	Abaújkér	town	26718	POINT (809795.600 331138.300)
...	...	...	...	...	...	...
3142	3143	GYÕR-MOSON-SOPRON	Zsira	town	04622	POINT (471324.200 237577.200)
3143	3144	CSONGRÁD	Zsombó	town	17765	POINT (721098.100 109690.000)
3144	3145	BORSOD-ABAUJ-ZEMPLÉN	Zsujta	town	11022	POINT (815027.400 353143.100)
3145	3146	SZABOLCS-SZATMÁR-BEREG	Zsurk	town	13037	POINT (884847.700 344952.800)
3146	3147	BORSOD-ABAUJ-ZEMPLÉN	Zubogy	town	19105	POINT (763123.300 338338.600)

3147 rows × 6 columns

The correct encoding of the file should be automatically detected. In case the Hungarian characters are displayed incorrectly, you may specify the encoding manually:

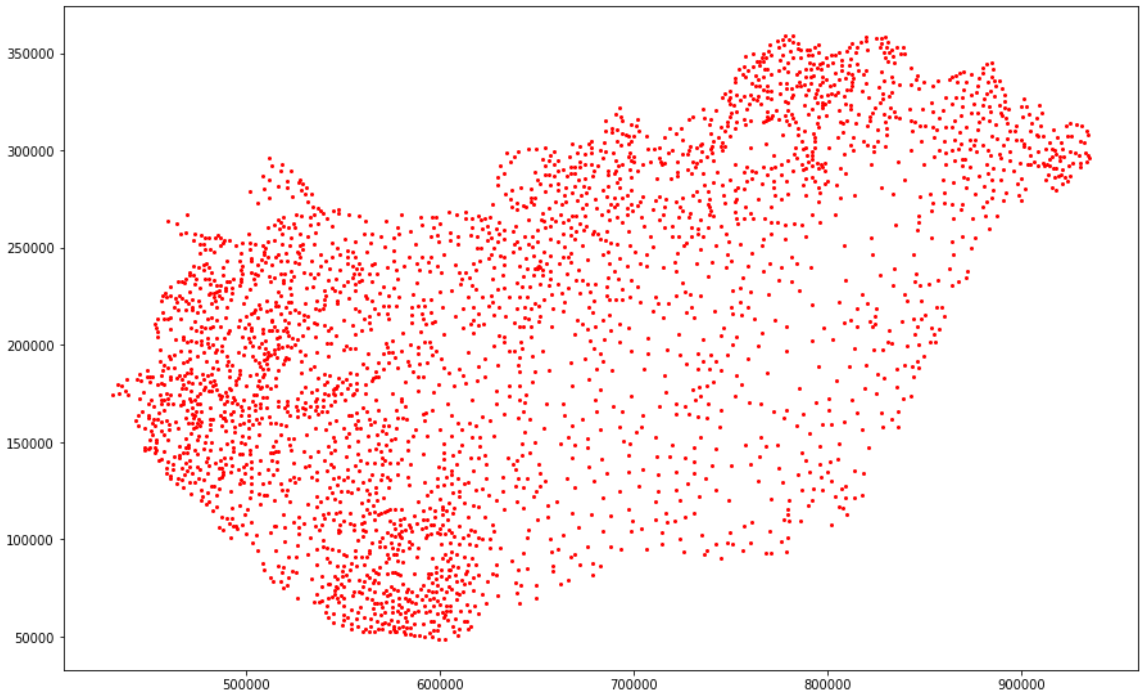
```
cities = gpd.read_file('../data/hungary_cities.shp', encoding='latin1')
```

Visualize the *GeoDataFrame*

Plot the location of all Hungarian cities:

In [2]:

```
import matplotlib.pyplot as plt
%matplotlib inline

cities.plot(figsize=[15,10], color='red', markersize=4)
plt.show()
```

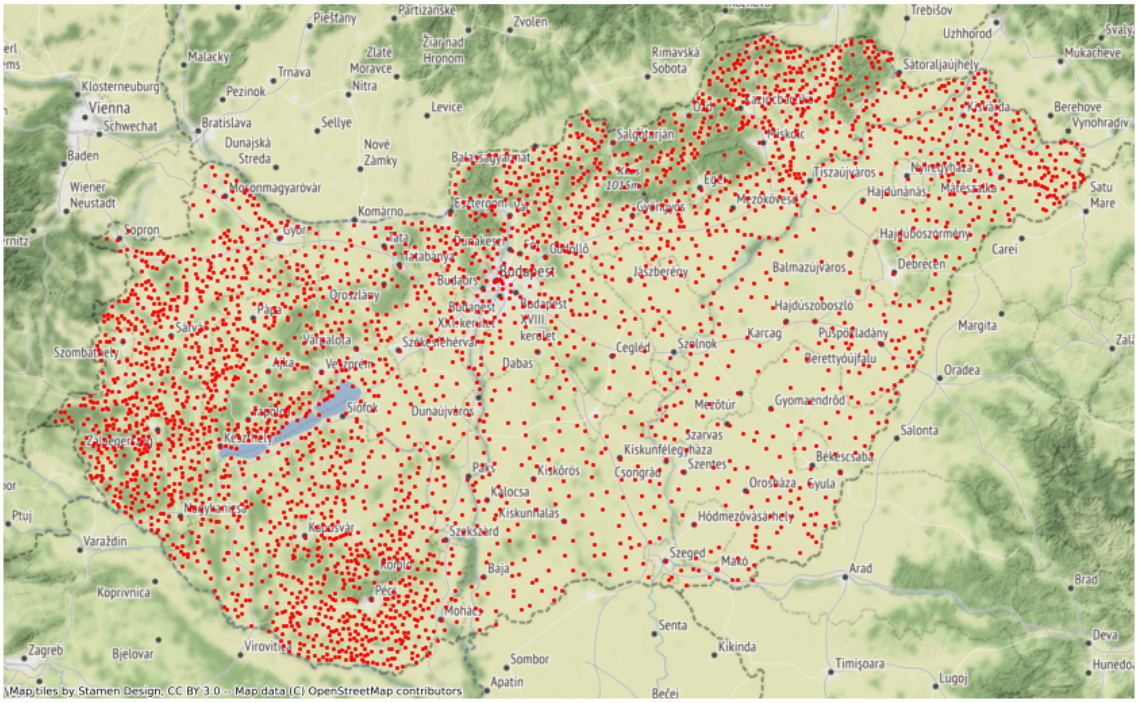


Add a raster base map to it with the *contextily* package:

In [3]:

```
import contextily as ctx

# Display the CRS
print(cities.crs)

# Set the CRS to EOV projection (EPSG:23700) if None
if(cities.crs == None):
    cities.set_crs('epsg:23700', inplace=True)

# Display the CRS
print(cities.crs)

# Transform the GeoDataFrame to Web Mercator projection (EPSG:3857) to display c
orrectly with the base map
ax = cities.to_crs('epsg:3857').plot(figsize=[15,10], color='red', markersize=4)
ax.set_axis_off()
ctx.add_basemap(ax)
plt.show()
```

```
None
epsg:23700
```



Create a minimum spanning tree

NetworkX supports both the *Prim* and the *Kruskal* algorithm for building a minimum / maximum spanning tree (https://networkx.org/documentation/stable/reference/algorithms/tree.html#module-networkx.algorithms.tree.mst), with a uniform interface. The default is *Kruskal*.

```
nx.minimum_spanning_tree(graph, weight, algorithm)
```

Example

Step 1: Create an undirected graph with the towns as the nodes.

In [4]:

```
import networkx as nx

# Create empty, undirected graph
graph = nx.Graph()

for index, row in cities.iterrows():
     graph.add_node(row['City'],
        county = row['County'],
        status = row['Status'],
        ksh_code = row['KSH'],
        location = row['geometry']
    )

# Check results
print(graph.nodes['Esztergom'])
```

```
{'county': 'KOMÁROM-ESZTERGOM', 'status': 'city', 'ksh_code': '2513
1', 'location': <shapely.geometry.point.Point object at 0x7f45966344
c0>}
```

Display the location in WKT format:

In [5]:

```
print(graph.nodes['Esztergom']['location'].wkt)
```

```
POINT (627140 272097.8)
```

Fetch the *(X,Y)* coordinates form the location:

In [6]:

```
print(graph.nodes['Esztergom']['location'].x)
print(graph.nodes['Esztergom']['location'].y)
```

```
627140.0
272097.8
```

Calculate the location between 2 cities with the *Pythagoras theorem*:

In [7]:

```
import math

def dist(loc_a, loc_b):
    return math.sqrt(math.pow(loc_a.x - loc_b.x, 2) +
                     math.pow(loc_a.y - loc_b.y, 2))


print(dist(graph.nodes['Esztergom']['location'], graph.nodes['Budapest']['locati
on']))
```

```
39476.19752399673
```

The *Point* type has a built-in `distance()` method to do that:

In [8]:

```
print(graph.nodes['Esztergom']['location'].distance(graph.nodes['Budapest']['loc
ation']))
```

```
39476.19752399673
```

Step 2: Create a complete graph (add all possible edges).

In [9]:

```
import math

for city_from in graph.nodes:
    location_from = graph.nodes[city_from]['location']
    for city_to in graph.nodes:
        location_to = graph.nodes[city_to]['location']
        if city_from < city_to: # we do not need to add all edges twice
            # Add edge to the graph with distance as its cost
            graph.add_edge(city_from, city_to,
                distance = graph.nodes[city_from]['location'].distance(graph.nod
es[city_to]['location']))

# Check results
print(graph['Esztergom']['Debrecen'])
```

```
{'distance': 218626.45554703576}
```

Step 3: Calculate the minimum spanning tree as a new graph.

In [10]:

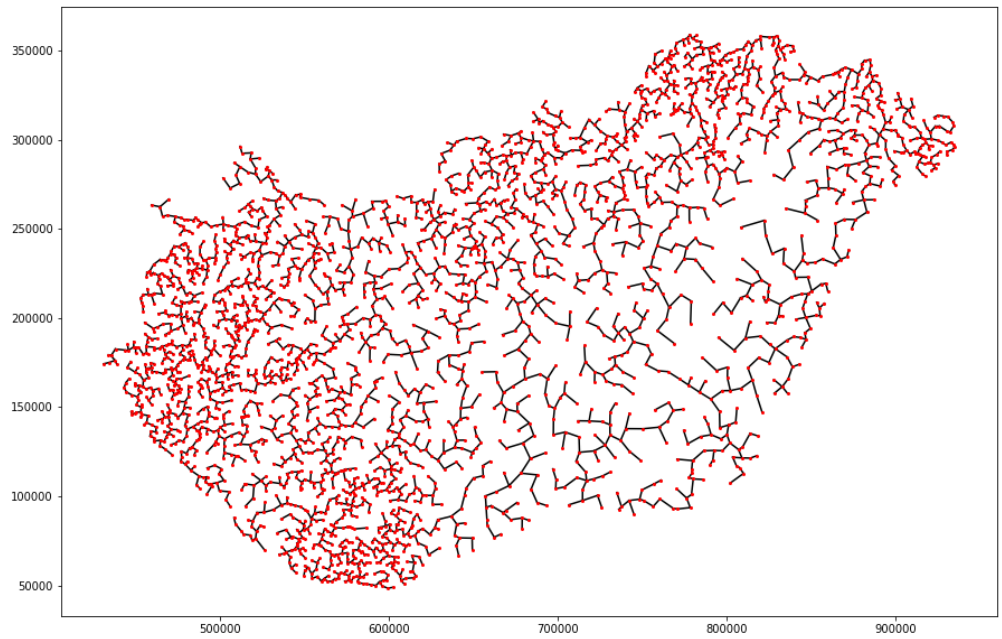
```
print('Number of nodes in original graph: {0}'.format(graph.order()))
print('Number of edges in original graph: {0}'.format(graph.size()))

spanning_tree = nx.minimum_spanning_tree(graph, weight = 'distance')

print('Number of nodes in spanning tree: {0}'.format(spanning_tree.order()))
print('Number of edges in spanning tree: {0}'.format(spanning_tree.size()))
```

```
Number of nodes in original graph: 3147
Number of edges in original graph: 4950231
Number of nodes in spanning tree: 3147
Number of edges in spanning tree: 3146
```

Step 4: Visualize results.

In [11]:

```
# Start new plot figure
plt.figure(figsize=[15,10])

# Plot all edges as black lines in the MST
for edge in spanning_tree.edges:
    city_from = edge[0]
    city_to   = edge[1]

    location_from = spanning_tree.nodes[city_from]['location']
    location_to   = spanning_tree.nodes[city_to]['location']
    plt.plot([location_from.x, location_to.x], [location_from.y, location_to.y],
 color='black')

# Plot all cities as red dots
for city in spanning_tree.nodes:
    location = spanning_tree.nodes[city]['location']
    plt.plot(location.x, location.y, color='red', marker='o', markersize=2)

# Display plot
plt.show()
```



Alternative approach: use NetworkX to draw the plot.

In [12]:

```
# Add all city coordinates a tuples to the nodes of the graph.
for node in spanning_tree.nodes:
    spanning_tree.nodes[node]['coords'] = spanning_tree.nodes[node]['location'].
coords[0]

# Visualize the spanning tree, using the positions in the coords field.
plt.figure(figsize=[15,10])
nx.draw_networkx(spanning_tree, nx.get_node_attributes(spanning_tree, 'coords'),
 with_labels=False, node_size=0)
plt.show()
```