The Algorithm for Ranking the Segments of the River Network for Geographic Information Analysis Based on Graphs

Introduction

Mikhail Sarafanov Follow
Aug 7, 2020 · 11 min read ★

The topic of this article is the application of information technologies in environmental science, namely, in hydrology. Below is a description of the algorithm for ranking rivers and the plugin we implemented for the open-source geographic information system QGIS.

An important aspect of hydrological surveys is not only the collection of information received from research expeditions and automatic devices but also the analysis of all the obtained data, including the use of GIS (geoinformation systems). However, exploration of the spatial structure of hydrological systems can be difficult due to a large amount of data. In such cases, we cannot do research without using additional tools that allow us to automate the process.

Visualization plays an important role when working with spatial data. Correct visual representation of the results of the analysis helps to better understand the structure of spatial objects and to know something new. For the image of rivers in classical cartography, the following method is used: rivers are represented as a solid line with a gradual thickening (depending on the number of tributaries that flow into the river) from the source to the mouth of the river. Moreover, segments of the river network often need to be ranked by the degree of distance from the source. This type of

information is important not only for visualization, but also for a more complete perception of the data structure, its spatial distribution, and subsequent processing.

The problem of ranking rivers can be illustrated as follows (Fig. 1):

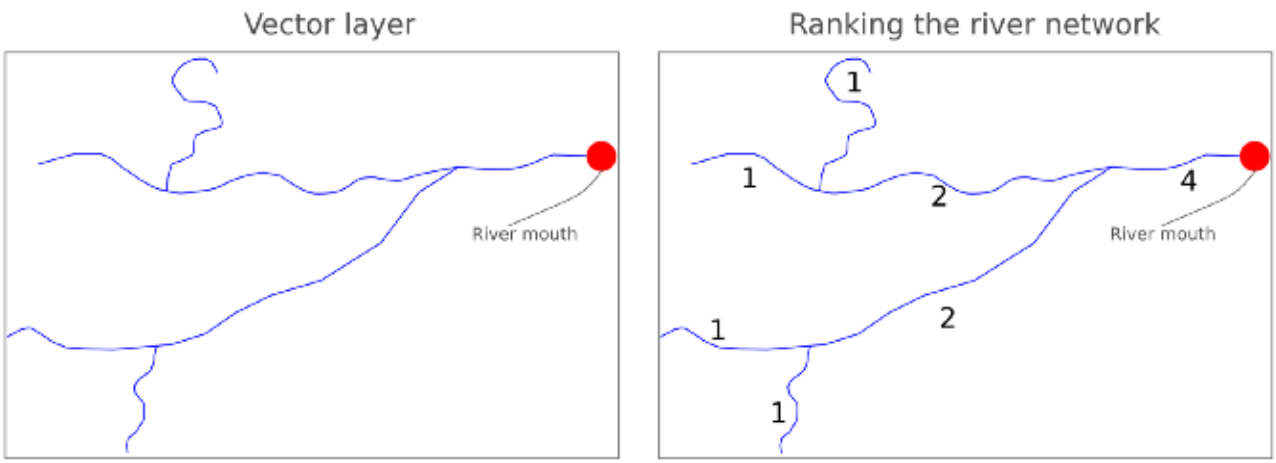


Figure 1. Ranking rivers task, numbers indicate the attribute assigned to each segment of the river network for the total number of tributaries flowing in

Thus, each segment of the river needs to be mapped to a value that shows how many segments flow into this section.

Modern GIS, such as ArcGIS or its open-source competitor QGIS, have tools for working with river networks. However, the river ranking tool requires a large number of additional auxiliary materials and, as it seems to us, unnecessary transformations. For example, for an existing GIS tool to start working with river networks you need to prepare a digital elevation model. A significant disadvantage, in addition to complex and multi-stage data preparation, is the inability to use already prepared vector layers with a river network for analysis, which limits the possibility of using digital bases from open sources (OpenStreetMap or Natural Earth).

Of course, you can assign attribute values to segments without using algorithms, but this approach is no longer relevant if you need to rank the network with several thousand segments.

We decided to automate this procedure by representing the river network as a graph and then applying graph traversal algorithms. To simplify the user's work with the implemented algorithm, a plugin for the QGIS geoinformation system — "Lines Ranking" was written. The code is distributed freely and is available in the QGIS repository, as well as on GitHub.

Installation

The plugin requires QGIS version >= 3.14, as well as the following dependencies: python libraries — networkx, pandas.

For Linux:

$ locate pip3

$ cd <your system pip3 path from previous step>

$ pip3 install pandas

$ pip3 install networkx

For Windows:

In command line OSGeo4W:

$ pip install pandas

$ pip install networkx

Using

Input data is a vector layer consisting of objects with a linear geometry type (Line, MultiLine). Custom attributes are stored in the input layer to the output layer.

> We do not recommend using a field named “fid” for the input layer. At the stage of connecting gaps in the river network, using the built — in module of the GRASS package- v.clean where this field name is the “system” one.

Also, an obligatory input parameter is a point (Start Point Coordinates) that determines the position of the mouth of the river network. It can be set from the map, from a file, or from a layer uploaded to QGIS. The position of the river mouth can be approximate. The calculation is based on the segment of the river network closest to the point (the closing vertex of the future graph).

Optional input data:

- the threshold for “tightening” gaps in the river network (Spline Threshold). This operation involves the use of a package of GRASS for QGIS. If the specified

package is missing, you should fix the gaps in another way and leave this field empty;

- custom field names for the output layer. Allows the user to assign a field name to record the rank of each segment (Rank fieldname), the number of tributaries (Flow field name), and the distance from the mouth in meters (Distance field name). If parameters are not set , the default names of the fields are Rank, Value, and Distance;
- location of the output file. If this parameter is omitted, a temporary layer will be created and added to the QGIS layer stack;

We can define the task performed by the algorithm as follows: compare the total number of tributaries flowing into each segment of the river, calculate the number of tributaries for each segment, as well as the distance of the farthest point of the segment from the mouth.

Algorithm description

In GIS, data can be presented in two main formats: raster and vector. A raster is a matrix where a certain parameter value is stored in each pixel. Satellite images, reanalysis grids, various output layers from climate models, and others are often represented in raster format in environmental science. Vector data is represented as simple geometric objects, such as points, lines, and polygons. Each object in a vector format can be associated with some information in the form of attributes. All the actions described below will be performed on the vector layer of the river network.

As a result, the algorithm returns a vector layer in which each object is assigned attributes that determine the distance of segments from the river mouth and the total number of tributaries that flow into this segment.

Preprocessing input data

Note that the topology of the original vector layer may be corrupted. The reason for this may be export / import between different GIS, incorrect file creation, etc. Corrupted layer topology can be expressed in the absence of connections between objects, i.e. the formation of various breaks (Fig. 2), creating additional closures, intersections, etc.

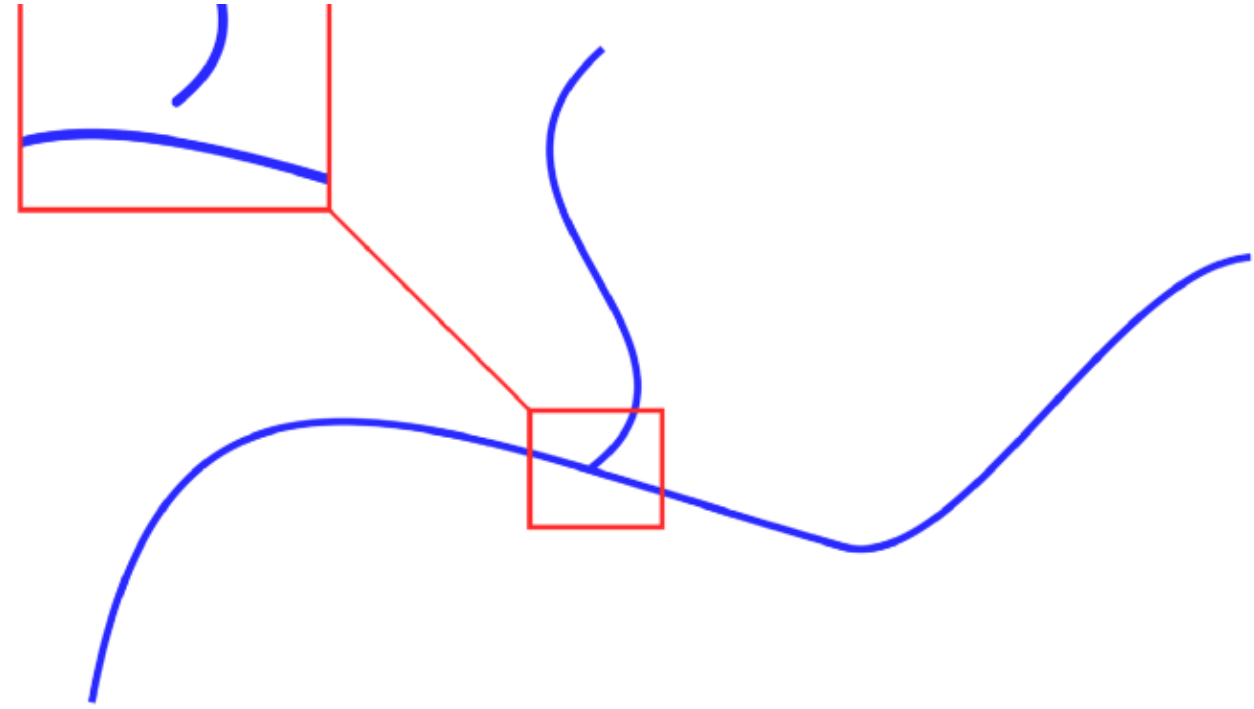


Figure 2. Corrupted topology of vector objects

Therefore, the first stage of preprocessing is to correct the topology of objects: “tighten” the nodes, make the original vector layer consistent. To do this, use the tools from the data analysis panel in QGIS — “Fix geometries” (fixgeometries built-in) and v.clean (from the GRASS package).

After the topology is fixed, the layer is divided into segments at the points where the lines have intersections. The result after splitting is illustrated below (Fig. 3).

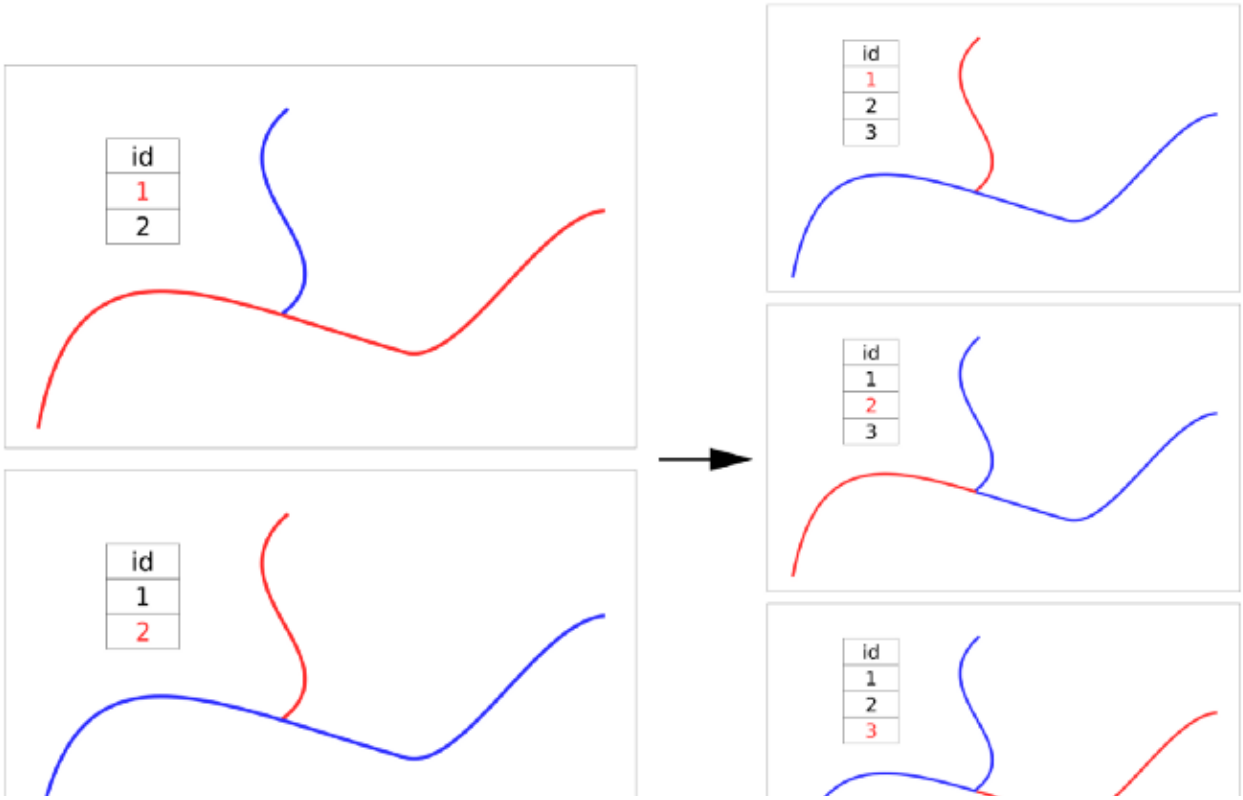


Figure 3. Split linear objects into segments

Thus, using the “splitwithlines” tool in QGIS, we divide the source layer into segments.

For each segment, we use QGIS to calculate the length and enter data in the attribute table of the layer (Fig. 4). the segment Length is calculated according to the user settings of the project (Project -> Properties -> General -> Ellipsoid).

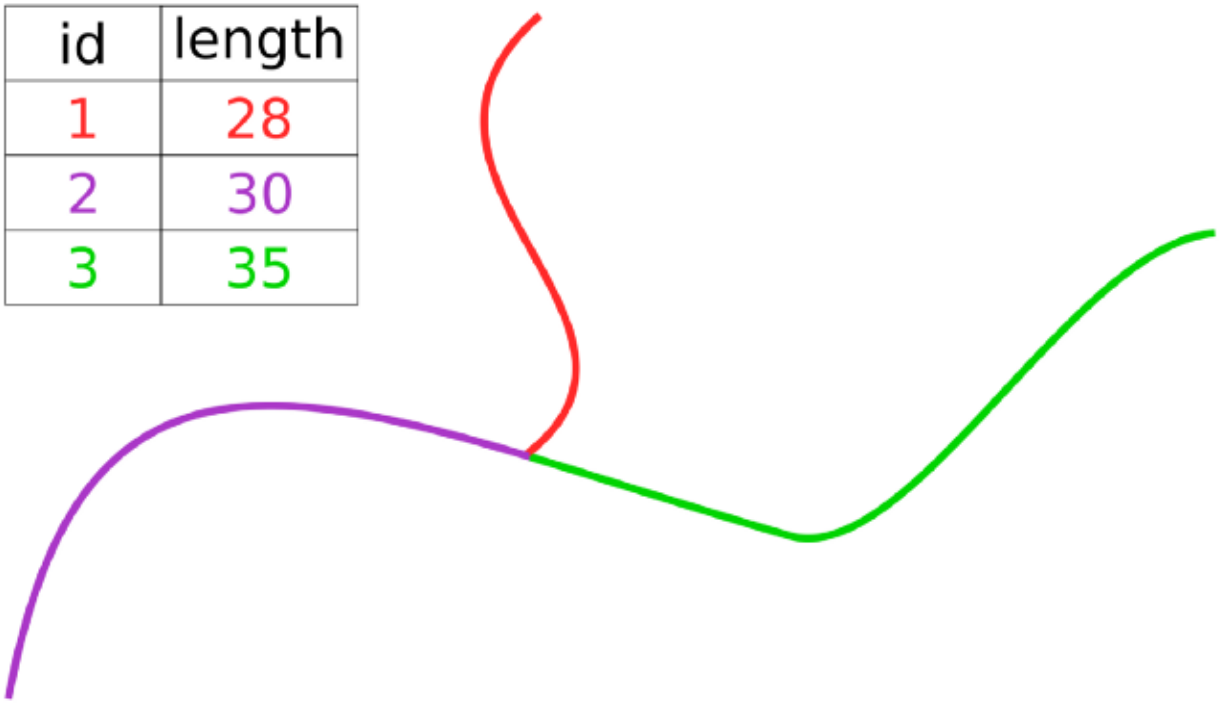
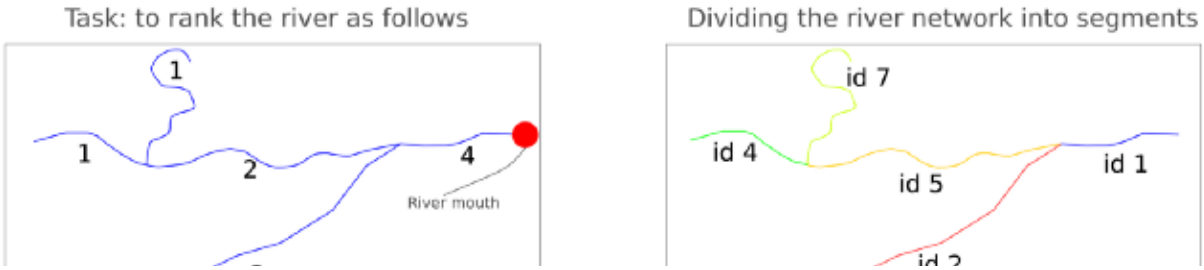


Figure 4. Calculating the length of segments

After that, using the “line intersection” tool (built-in tool), we get a point vector layer, where information about segment intersections is set in the attribute table. This attribute table can be interpreted as an adjacency list.

The preprocessing steps are shown in the following image (Fig. 5).



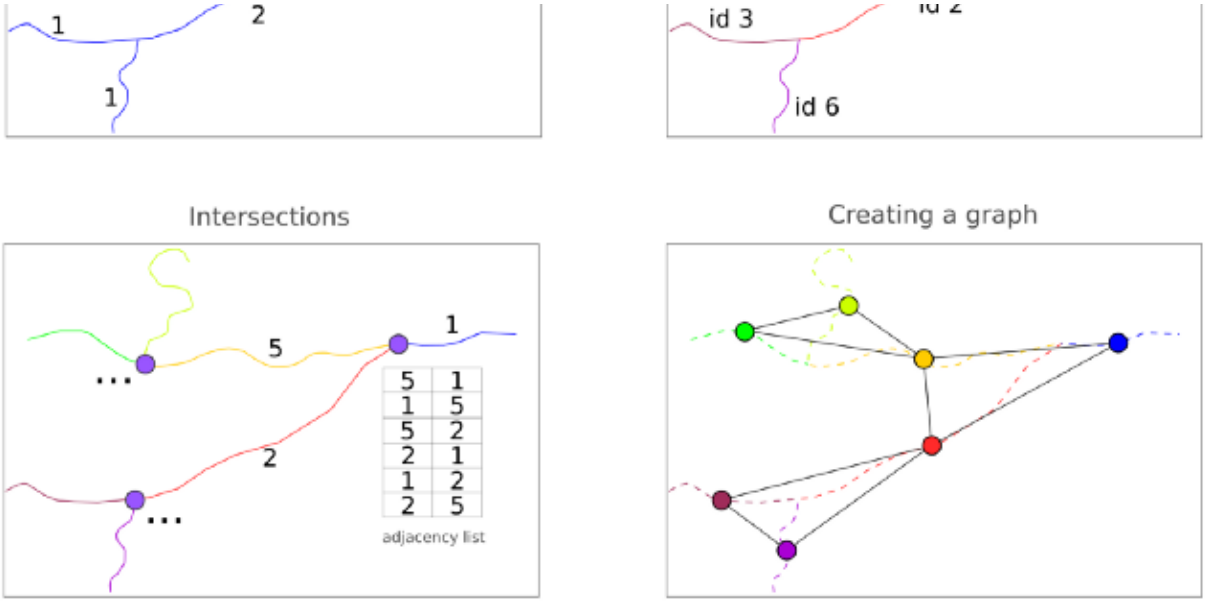


Figure 5. Steps for preprocessing a vector linear layer to represent it as a graph

As a result of preprocessing, the graph is formed as a mathematical object of the networkx Python library. Thus, the river segments are vertices in the graph. If the segments are connected to each other (they have intersections), then there are edges between the graph vertices.

Algorithm for ranking linear objects

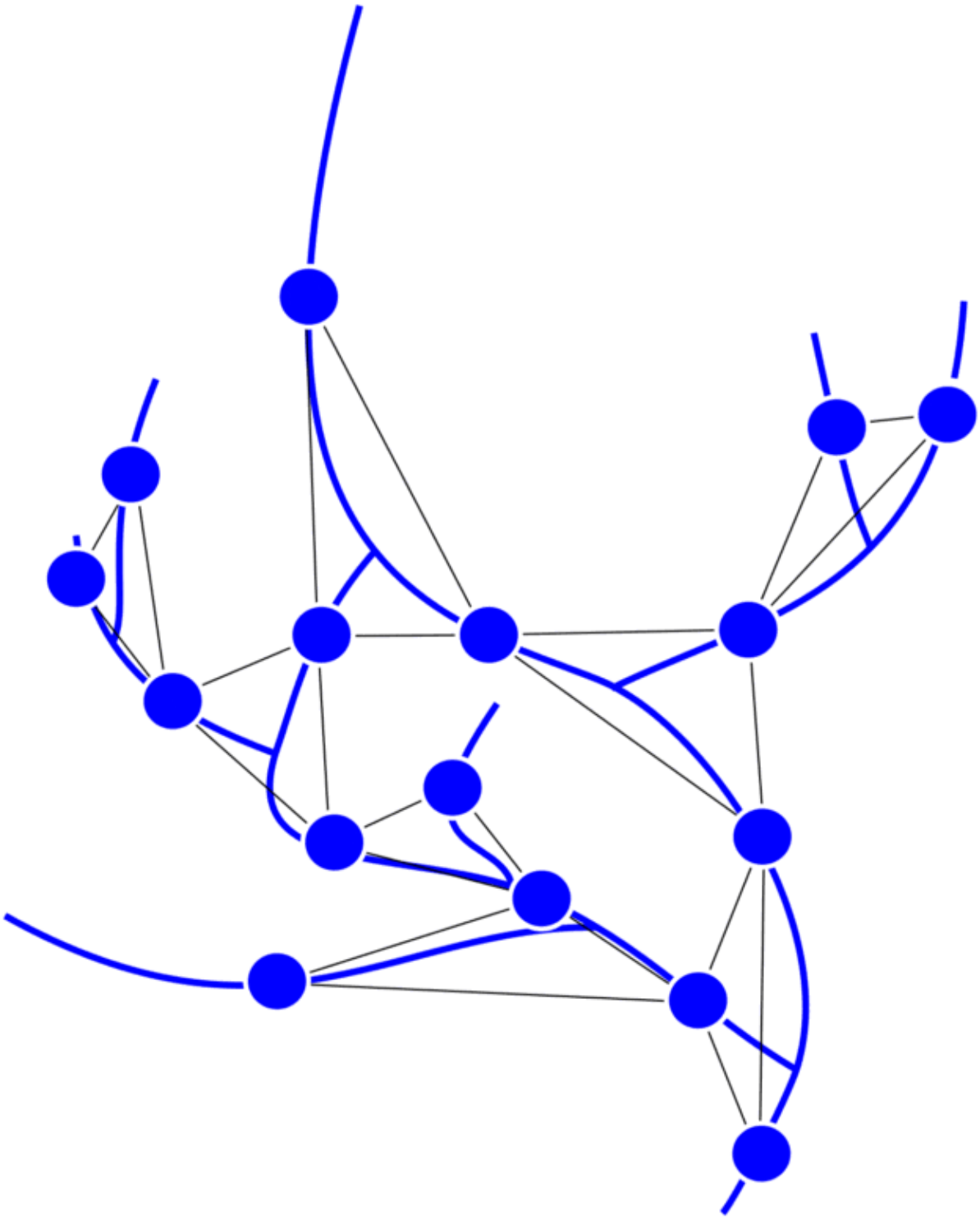
After the graph is formed, we know which vertex to start the search from (the point where it flows from the main river into the lake or sea). Let's call this vertex the closing one, since all other segments of the river network (vertexes) "flow" into it. We have divided the algorithm into several parts:

1. Ranking graph vertexes by distance from the closing segment, assigning the "rank" attribute (a measure of the segment's distance) and the "offspring" attribute (the number of sections of the river network that flow directly into this segment);
2. Assigning the "value" attribute for the total number of tributaries flowing into a given section and the "distance" attribute (the distance of the segment's extreme point from the mouth in meters).

Both stages are divided into several blocks, but the main idea is a two-stage scheme for assigning attributes.

The assignment of the attributes "rank" and "offspring"

The first stage of graph traversal is to rank vertexes by the degree of distance from the closing one. We planned to carry out the assignment of the attribute “rank” with iterative breadth-first search (BFS). Thus, starting from the closing vertex, we would move further and further away at each step, and at the same time, we would assign an attribute “rank”. But in this case, the following conflict may occur (animation below).



And what rank should we assign to this segment? There may be other problems with this attribute assignment algorithm, but we have listed one of the most vital.

Suggested solution: we can determine the ranks for some part of the river network (the main river), and rank segments based on this information. This approach can be seen on the following picture (Fig. 6).

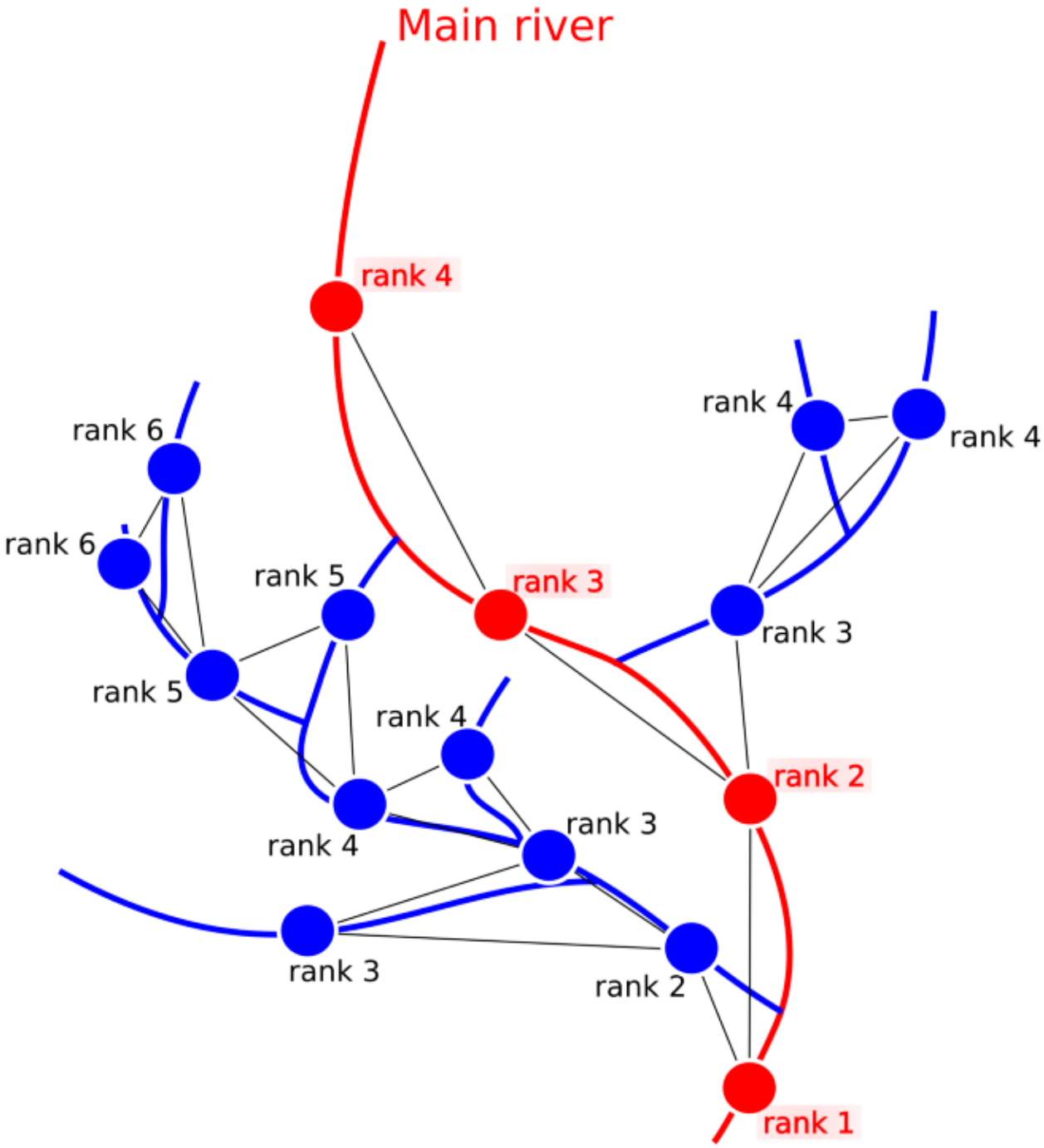


Figure 6. Conflict resolution by introducing a reference route in the graph

Thus, subgraphs can only join the reference route through 1 edge, and the other edges are excluded.

This raises the following problem : how can we find such a route? — We will assume that the shortest route between the two most distant vertices in the graph, one of which is the closing one, is the reference route (we will soon add an update so that the user can set this route if they have knowledge of which river is the main one, but in the absence of such information, the reference route is determined by this way). This route can be obtained using the A* (A-star) algorithm, but this algorithm works with a weighted graph, and there are no weights on the edges of our graph yet. But we can set weights for the edges of the graph based on the segment lengths (we calculated them earlier).

- Assigning weights to the graph edges based on the lengths of segments. Simultaneously with this stage, one component is selected in the graph. The movement of the column is carried out using a breadth-first search. The assignment of weights can be demonstrated by the following figure (Fig. 7)

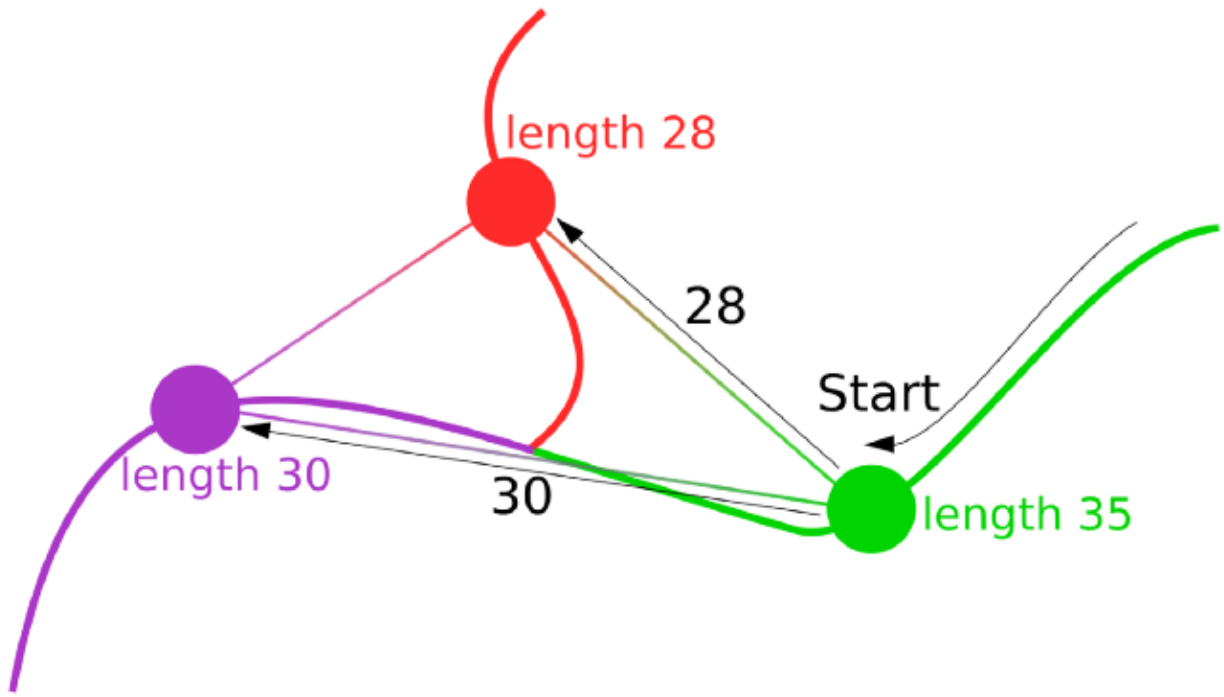


Figure 7. Assigning weights to graph edges

Thus, each vertex of the graph has the “length” attribute, which indicates the length of this segment of the river in meters. We also move attribute values from the graph vertices to the edges iteratively, starting to traverse the graph using BFS from the closing vertex.

This task is performed by the following function, where are

- G — graph
- start — the vertex from which to start traversal
- dataframe — pandas dataframe with 2 columns: id_field (segment/vertex ID) and ‘length’ (the length of this segment)
- id_field — the field in the dataframe to use for mapping IDs to graph vertexes
- main_id — index for the main river in the river network (default = None)

```
# Function for assigning weights to graph edges
def distance_attr(G, start, dataframe, id_field, main_id = None):

    # List of all vertexes that can be reached from the start vertex using BFS
    vert_list = list(nx.bfs_successors(G, source=start))
    # One of the most remote vertices in the graph (this will be necessary for A*)
    last_vertex = vert_list[-1][-1][0]

    for component in vert_list:
        vertex = component[0]  # The vertex where we are at this iteration
        neighbors = component[1]  # Vertices that are neighboring (which we haven't visited ye

        dist_vertex = int(dataframe['length'][dataframe[id_field] == vertex])
        # Assign the segment length value as a vertex attribute
        attrs = {vertex: {'component' : 1, 'size' : dist_vertex}}
        nx.set_node_attributes(G, attrs)

        for n in neighbors:

            # If the main index value is not set
            if main_id == None:
                # Assign weights to the edges of the graph
                # The length of the section in meters (int)
                dist_n = int(dataframe['length'][dataframe[id_field] == n])
            # Otherwise we are dealing with a complex composite index
            else:
                # If the vertex we are at is part of the main river
                if vertex.split(':')[0] == main_id:
                    # And at the same time, the vertex that we "look" at from the vertex "vert
                    if n.split(':')[0] == main_id:
                        # The weight value must be zero
                        dist_n = 0
                    else:
                        dist_n = int(dataframe['length'][dataframe[id_field] == n])
                else:
                    dist_n = int(dataframe['length'][dataframe[id_field] == n])
```

```
                dist_n = int(dataframe['length'][dataframe[id_field] == n])
            attrs = {(vertex, n): {'weight': dist_n},
                     (n, vertex): {'weight': dist_n}}
            nx.set_edge_attributes(G, attrs)

            # Assign attributes to the nodes of the graph
            attrs = {n: {'component' : 1, 'size' : dist_n}}
            nx.set_node_attributes(G, attrs)


        # Look at the surroundings of the vertex where we are located
        offspring = list(nx.bfs_successors(G, source = vertex, depth_limit = 1))
        offspring = offspring[0][1]
        # If the weight value was not assigned, we assign it
        for n in offspring:

            if len(G.get_edge_data(vertex, n)) == 0:

                ################################
                # Assigning weights to edges #
                ################################
                if main_id == None:
                    dist_n = int(dataframe['length'][dataframe[id_field] == n])
                else:
                    if vertex.split(':')[0] == main_id:
                        if n.split(':')[0] == main_id:
                            dist_n = 0
                        else:
                            dist_n = int(dataframe['length'][dataframe[id_field] == n])
                    else:
                        dist_n = int(dataframe['length'][dataframe[id_field] == n])
                attrs = {(vertex, n): {'weight': dist_n},
                         (n, vertex): {'weight': dist_n}}
                nx.set_edge_attributes(G, attrs)
                ################################
                # Assigning weights to edges #
                ################################

            elif len(G.get_edge_data(n, vertex)) == 0:

                ################################
                # Assigning weights to edges #
                ################################
                if main_id == None:
                    dist_n = int(dataframe['length'][dataframe[id_field] == n])
                else:
                    if vertex.split(':')[0] == main_id:
                        if n.split(':')[0] == main_id:
```

```
                            dist_n = 0
                        else:
                            dist_n = int(dataframe['length'][dataframe[id_field] == n])
                    else:
                        dist_n = int(dataframe['length'][dataframe[id_field] == n])
                attrs = {(vertex, n): {'weight': dist_n},
                         (n, vertex): {'weight': dist_n}}
                nx.set_edge_attributes(G, attrs)
                ################################
                # Assigning weights to edges #
                ################################

    for vertex in list(G.nodes()):
        # If the graph is incompletely connected, then we delete the elements that we can't ge
        if G.nodes[vertex].get('component') == None:
            G.remove_node(vertex)
        else:
            pass
    return(last_vertex)

# The application of the algorithm
last_vertex = distance_attr(G, '7126:23', dataframe, id_field = 'id', main_id = '7126')
```

- A* (A-star) search for the shortest path on a weighted graph between the closing vertex and one of the most distant vertices (a segment in the river network). This shortest route between the two most distant vertices in the graph is called “reference route”;
- Ranking by distance of all vertexes in the reference route. Vertex, from where we start the traversal, the value is assigned rank 1, the next vertex is 2, then — 3, etc.
- Iterative traversal of a graph with the beginning at the vertices of the reference route with isolation of the considered subgraphs. If one of the graph branches already has a connection to the vertices of the reference route, the edges that link the subgraph to other reference vertices are removed.

You can see the source code here

- G — graph
- start — the vertex from which to start traversal
- last_vertex — one of the farthest vertices from the closing one in the graph

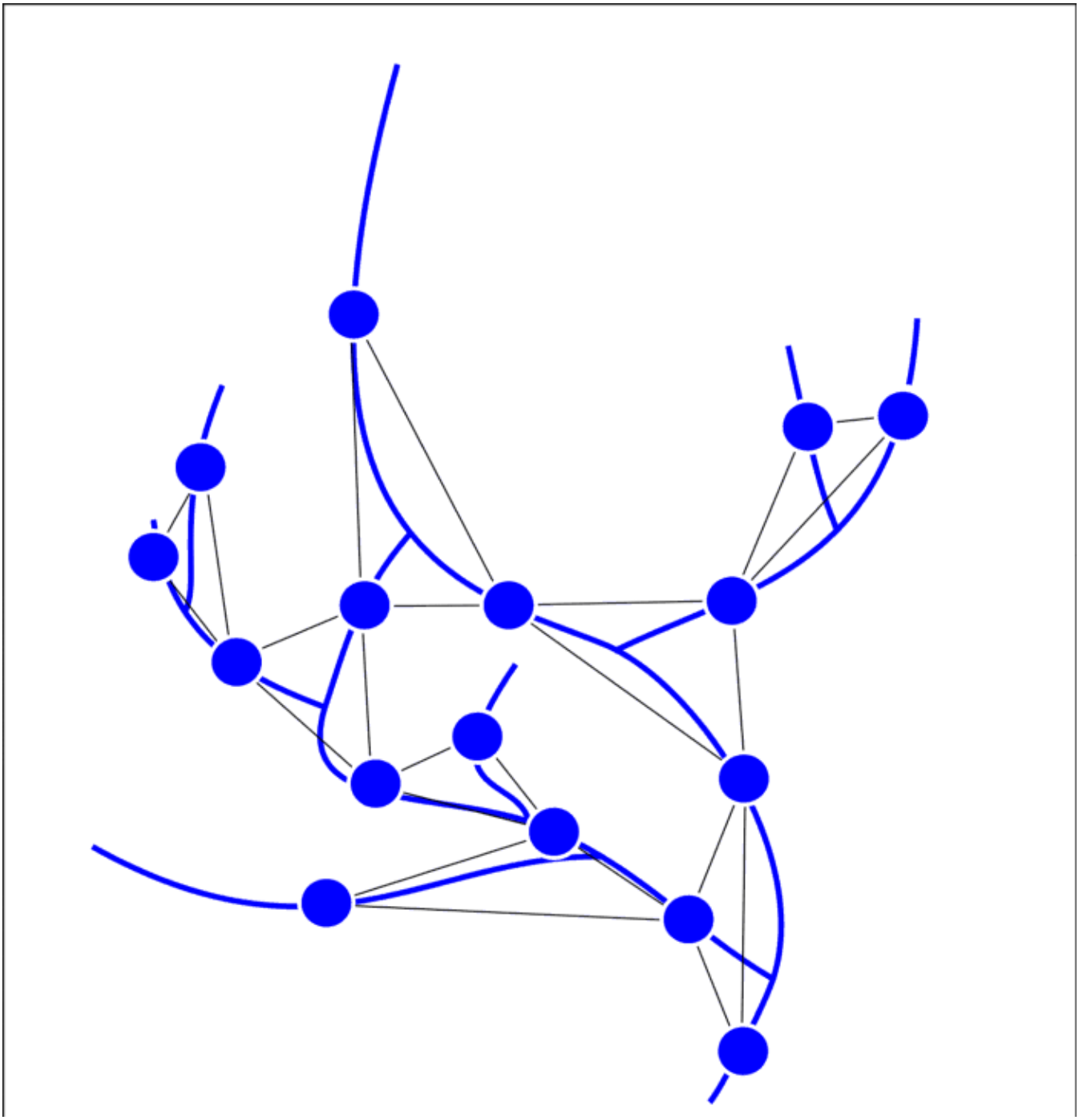
```
# Function for assigning 'rank' and 'offspring' attributes to graph vertices
def rank_set(G, start, last_vertex):

    # Traversing a graph with attribute assignment
    # G           --- graph as a networkx object
    # vertex      --- vertex from which the graph search begins
    # kernel_path --- list of vertexes that are part of the main path that the search is being
    def bfs_attributes(G, vertex, kernel_path):

        # Creating a copy of the graph
        G_copy = G.copy()

        # Deleting all edges that are associated with the reference vertexes
        for kernel_vertex in kernel_path:
            # Leaving the reference vertex from which we start the crawl
            if kernel_vertex == vertex:
                pass
            else:
                # For all other vertexes, we delete edges
                kernel_n = list(nx.bfs_successors(G_copy, source = kernel_vertex, depth_limit
                kernel_n = kernel_n[0][1]
                for i in kernel_n:
                    try:
                        G_copy.remove_edge(i, kernel_vertex)
                    except Exception:
                        pass

        # The obtained subgraph is isolated from all reference vertices, except the one
        # from which the search begins at this iteration
        # Breadth-first search
        all_neighbors = list(nx.bfs_successors(G_copy, source = vertex))

        ###############################################################################
        #                                Attention!                                   #
        # Labels are not assigned on an isolated subgraph, but on the source graph    #
        ###############################################################################
        for component in all_neighbors:
            v = component[0] # The vertex where we are at this iteration
            neighbors = component[1] # Vertices that are neighboring (which we haven't visited

            # Value of the 'rank' attribute in the considering vertex
            att = G.nodes[v].get('rank')
            if att != None:
                # The value of the attribute to be assigned to neighboring vertices
                att_number = att + 1

            # We look at all the closest first offspring
            first_n = list(nx.bfs_successors(G, source = v, depth_limit = 1))
```

```
            first_n = list(nx.bfs_successors(G, source = v, depth_limit = 1))
            first_n = first_n[0][1]

            # Assigning ranks to vertices
            for i in first_n:
                # If the neighboring vertex is the main node in this iteration, then skip it
                # vertex - the reference point from which we started the search
                if i == vertex:
                    pass
                else:
                    current_i_rank = G.nodes[i].get('rank')
                    # If the rank value has not yet been assigned, then assign it
                    if current_i_rank == None:
                        attrs = {i: {'rank': att_number}}
                        nx.set_node_attributes(G, attrs)
                    # If the rank in this node is already assigned
                    else:
                        # The algorithm either "looks back" at vertices that it has already vi
                        # In this case we don't do anything
                        # Either the algorithm "came up" to the main path (kernel path) in the
                        if any(i == bearing_v for bearing_v in kernel_path):
                            G.remove_edge(v, i)
                        else:
                            pass

            # Additional "search"
            for neighbor in neighbors:
                # We look at all the closest first offspring
                first_n = list(nx.bfs_successors(G, source = neighbor, depth_limit = 1))
                first_n = first_n[0][1]

                for i in first_n:
                    # If the neighboring vertex is the main node in this iteration, then skip
                    # vertex - the reference point from which we started the search
                    if i == vertex:
                        pass
                    else:
                        # The algorithm either "looks back" at vertices that it has already vi
                        # In this case we don't do anything
                        # Either the algorithm "came up" to the main path (kernel path) in the
                        if any(i == bearing_v for bearing_v in kernel_path):
                            G.remove_edge(neighbor, i)
                        else:
                            pass

    # Finding the shortest path A* - building a route around which we will build the next sear
    a_path = list(nx.astar_path(G, source = start, target = last_vertex, weight = 'weight'))

```

```
    ##############################
    #   Route validation block   #
    ##############################
    true_a_path = []
    for index, V in enumerate(a_path):
        if index == 0:
            true_a_path.append(V)
        elif index == (len(a_path) - 1):
            true_a_path.append(V)
        else:
            # Previous and next vertices for the reference path (a_path)
            V_prev = a_path[index - 1]
            V_next = a_path[index + 1]

            # Which vertexes are adjacent to this one
            V_prev_neighborhood = list(nx.bfs_successors(G, source = V_prev, depth_limit = 1))
            V_prev_neighborhood = V_prev_neighborhood[0][1]
            V_next_neighborhood = list(nx.bfs_successors(G, source = V_next, depth_limit = 1))
            V_next_neighborhood = V_next_neighborhood[0][1]

            # If the next and previous vertices are connected to each other without an interme
            # in the form of vertex V, then vertex V is excluded from the reference path
            if any(V_next == VPREV for VPREV in V_prev_neighborhood):
                if any(V_prev == VNEXT for VNEXT in V_next_neighborhood):
                    pass
                else:
                    true_a_path.append(V)
            else:
                true_a_path.append(V)
    ##############################
    #   Route validation block   #
    ##############################

    # Verification completed
    a_path = true_a_path
    RANK = 1
    for v in a_path:
        # Assign the attribute rank value - 1 to the starting vertex. The further away, the gr
        attrs = {v: {'rank' : RANK}}
        nx.set_node_attributes(G, attrs)
        RANK += 1

    # The main route is ready, then we will iteratively move from each node
    for index, vertex in enumerate(a_path):
        # Starting vertex
        if index == 0:
            next_vertex = a_path[index + 1]
```

```
            # Disconnect vertices
            G.remove_edge(vertex, next_vertex)

            # Subgraph BFS block
            bfs_attributes(G, vertex = vertex, kernel_path = a_path)

            # Connect vertices back
            G.add_edge(vertex, next_vertex)


        # Finishing vertex
        elif index == (len(a_path) - 1):
            prev_vertex = a_path[index - 1]

            # Disconnect vertices
            G.remove_edge(prev_vertex, vertex)

            # Subgraph BFS block
            bfs_attributes(G, vertex = vertex, kernel_path = a_path)

            # Connect vertices back
            G.add_edge(prev_vertex, vertex)

        # Vertices that are not the first or last in the reference path
        else:
            prev_vertex = a_path[index - 1]
            next_vertex = a_path[index + 1]

            # Disconnect vertices
            # Previous with current vertex
            try:
                G.remove_edge(prev_vertex, vertex)
            except Exception:
                pass
            # Current with next vertex
            try:
                G.remove_edge(vertex, next_vertex)
            except Exception:
                pass

            # Subgraph BFS block
            bfs_attributes(G, vertex = vertex, kernel_path = a_path)

            # Connect vertices back
            try:
                G.add_edge(prev_vertex, vertex)
                G.add_edge(vertex, next_vertex)
```

```
            except Exception:
                pass

    # Attribute assignment block - number of descendants
    vert_list = list(nx.bfs_successors(G, source = start))
    for component in vert_list:
        vertex = component[0] # The vertex where we are at this iteration
        neighbors = component[1] # Vertices that are neighboring (which we haven't visited yet
        # Adding an attribute - the number of descendants of this vertex
        n_offspring = len(neighbors)
        attrs = {vertex: {'offspring' : n_offspring}}
```



Thus, the reference route can be considered as the main river in the river network, when all other segments are tributaries to the main one.

Moreover, this approach allows you to achieve good results when ranking rivers in “difficult” places, such as the mouth, where some branches first depart from the main river, and then, gaining new tributaries, flow into the main river again.

Assigning the “value” and “distance” attributes

So, on the graph, all vertices are assigned the values of the “rank” and “offspring” attributes.

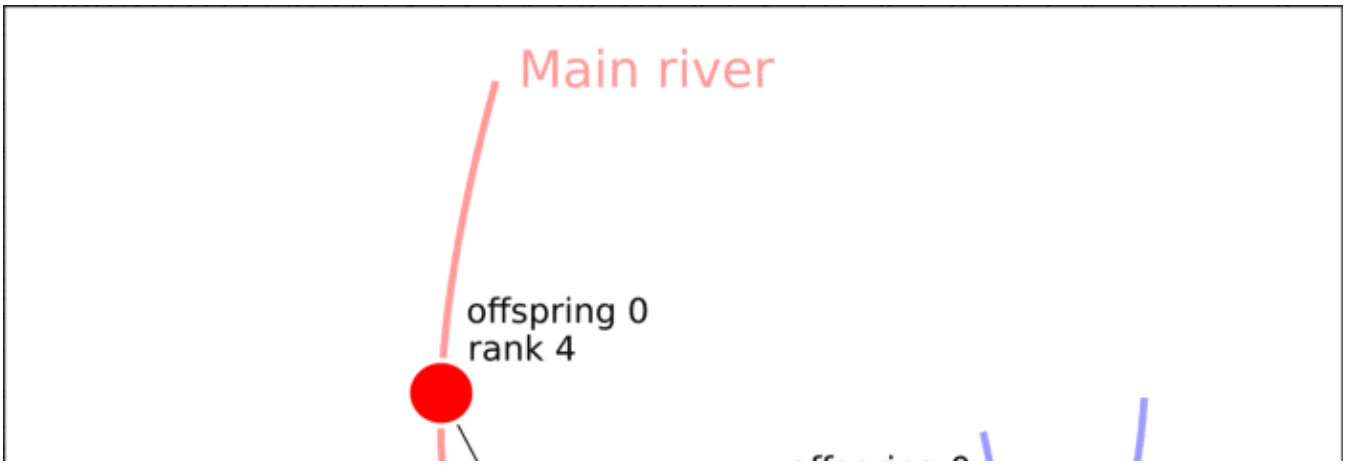
If the vertex has no offspring, it means that no tributaries flow into this segment of the river network. Therefore, this vertex must be assigned the value “value” — 1. Then, for each node that has descendants (the rank of the descendants is always 1 less than the rank of the considering vertex) with the value “value” equal to 1, we need to count the number of descendants. The sum of “value” of all descendants of the considering vertex — the “ value” for considering vertex. Then, this procedure is repeated for other ranks.

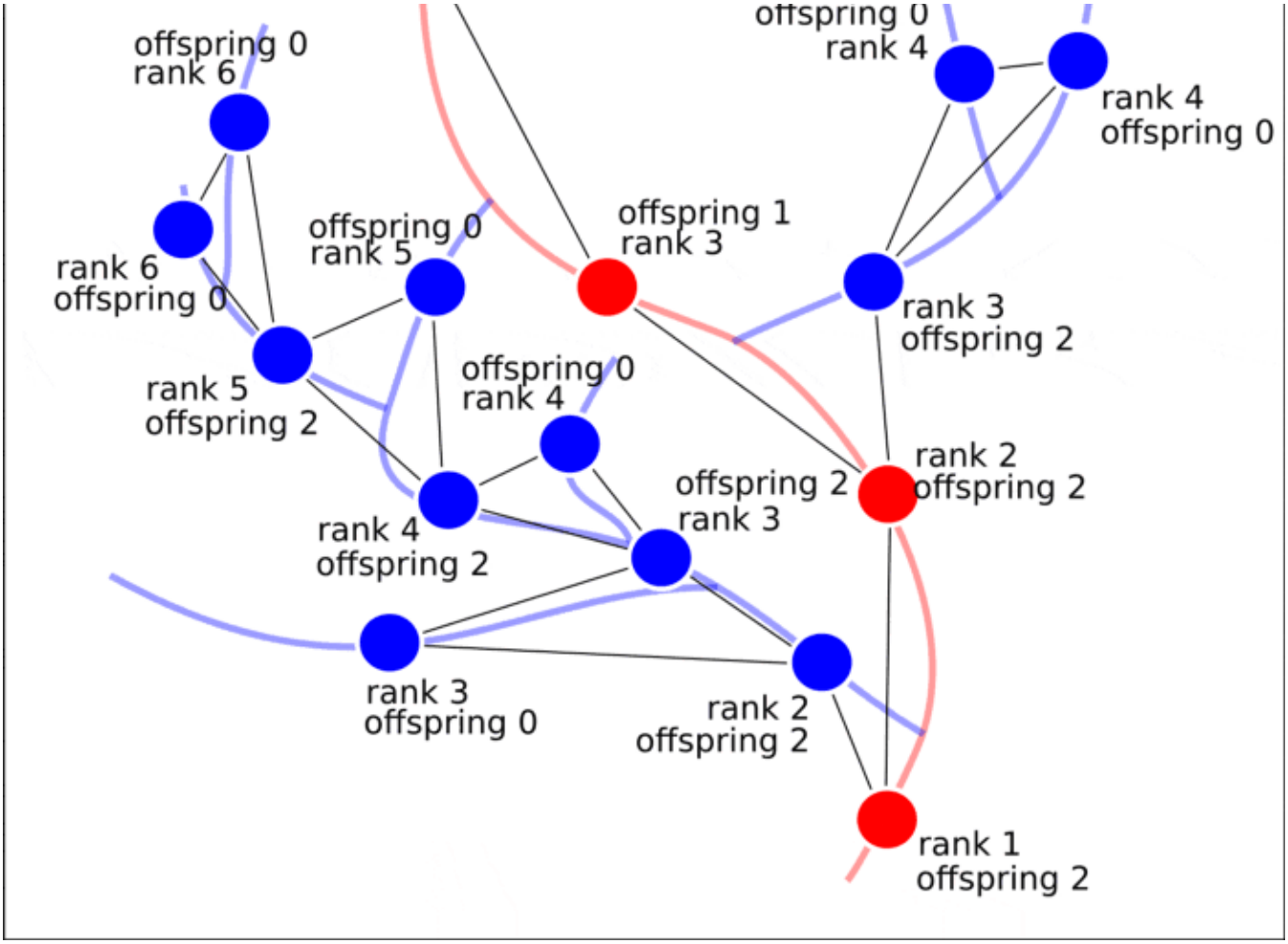
```
# Function for determining the order of river segments similar to the Shreve method
# In addition, the "distance" attribute is assigned
def set_values(G, start, considering_rank, vert_list):

    # For each vertex in the list
    for vertex in vert_list:

        # If value has already been assigned, then skip it
        if G.nodes[vertex].get('value') == 1:
            pass
        else:
            # Defining descendants
            offspring = list(nx.bfs_successors(G, source = vertex, depth_limit = 1))
            # We use only the nearest neighbors to this vertex (first descendants)
            offspring = offspring[0][1]

            # The cycle of determining the values at the vertices of a descendant
            last_values = []
            for child in offspring:
                # We only need descendants whose rank value is greater than that of the vertex
                if G.nodes[child].get('rank') > considering_rank:
```

```
                    if G.nodes[child].get('value') != None:
                        last_values.append(G.nodes[child].get('value'))
                    else:
                        pass
                else:
                    pass

            last_values = np.array(last_values)
            sum_values = np.sum(last_values)

            # If the amount is not equal to 0, the attribute is assigned
            if sum_values != 0:
                attrs = {vertex: {'value' : sum_values}}
                nx.set_node_attributes(G, attrs)
            else:
                pass

# Function for iteratively assigning the value attribute
def iter_set_values(G, start):

    # Vertices and corresponding attribute values
    ranks_list = []
    vertices_list = []
    offspring_list = []
    for vertex in list(G.nodes()):
        ranks_list.append(G.nodes[vertex].get('rank'))
        vertices_list.append(vertex)
        att_offspring = G.nodes[vertex].get('offspring')

        if att_offspring == None:
            offspring_list.append(0)
        else:
            offspring_list.append(att_offspring)

    # Largest rank value in a graph
    max_rank = max(ranks_list)

    # Creating pandas dataframe
    df = pd.DataFrame({'ranks': ranks_list,
                       'vertices': vertices_list,
                       'offspring': offspring_list})

    # We assign value = 1 to all vertices of the graph that have no offspring
    value_1_list = list(df['vertices'][df['offspring'] == 0])
    for vertex in value_1_list:
        attrs = {vertex: {'value' : 1}}
        nx.set_node_attributes(G, attrs)

```

```
    # For each rank, we begin to assign attributes
    for considering_rank in range(max_rank, 0, -1):

        # List of vertices of suitable rank
        vert_list = list(df['vertices'][df['ranks'] == considering_rank])
        set_values(G, start, considering_rank, vert_list)


    # Assigning the "distance" attribute to the graph vertices
    # List of all vertexes that can be reached from the start vertex using BFS
    vert_list = list(nx.bfs_successors(G, source = start))
    for component in vert_list:
        vertex = component[0] # The vertex where we are at this iteration
        neighbors = component[1] # Vertices that are neighboring (which we haven't visited yet
        # If we are at the closing vertex
        if vertex == start:
            # Length of this segment
            att_vertex_size = G.nodes[vertex].get('size')
            # Adding the value of the distance attribute
            attrs = {vertex: {'distance' : att_vertex_size}}
            nx.set_node_attributes(G, attrs)
        else:
            pass

        vertex_distance = G.nodes[vertex].get('distance')
        # For each neighbor, we assign an attribute
        for i in neighbors:
            # Adding the value of the distance attribute
            i_size = G.nodes[i].get('size')
            attrs = {i: {'distance' : (vertex_distance + i_size)}}
            nx.set_node_attributes(G, attrs)
```

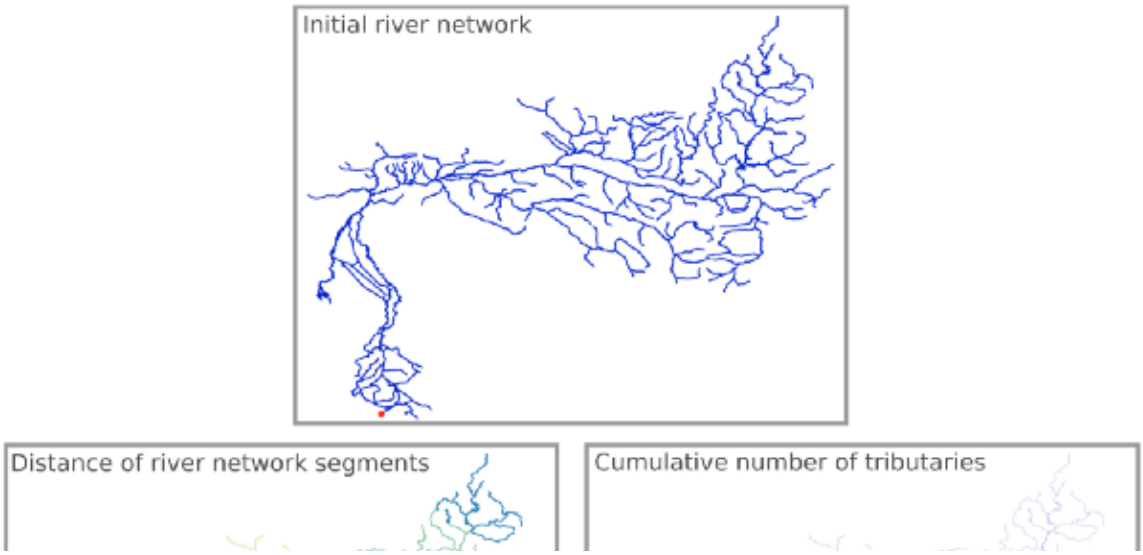
Thus, we iteratively move to the closing vertex.





At the same time as assigning the “value” attribute to the graph vertices, the “distance” attribute is assigned, which characterizes the distance of segments from the mouth not by the number of segments that must be overcome to reach the closing one, but by the distance in meters that will need to be overcome to reach the river mouth.

The result of using the algorithm can be seen in figure 8. the Distance of river network segments is shown based on the “rank” attribute and the total number of tributaries is shown based on the “value” attribute.



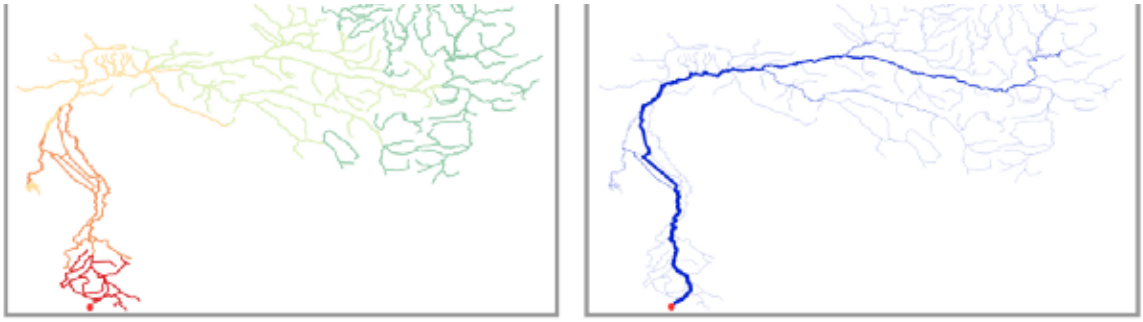


Figure 8. Results of using the algorithm (OpenStreetMap data, the Distance of river network segments is shown based on the “rank” attribute, the Total number of flowing tributaries is shown based on the “value” attribute”)

Conclusion

As you can see, the presented algorithm allows you to rank rivers without using additional information, such as digital elevation model. The obligatory input parameter, except for the main layer of the river network, is the position of the closing segment (where the river flows into the sea or lake), which can be specified as a point.

Thus, with the minimum amount of required input data, it is possible to obtain structured derived information that characterizes the elements of the river basin using the implemented algorithm.

An open implementation of the algorithm in Python, as well as a plugin for QGIS, can be used by anyone. All processing is carried out by one thread, the user does not need to run all the functions separately.

We are glad to answer your questions and see your comments.

Repository with the code:

- https://github.com/ChrisLisbon/QGIS_LinesRankingPlugin

Feel free to contact us by e-mail:

- mik_sar@mail.ru

Hydrology Ranking Algorithm Qgis Graph