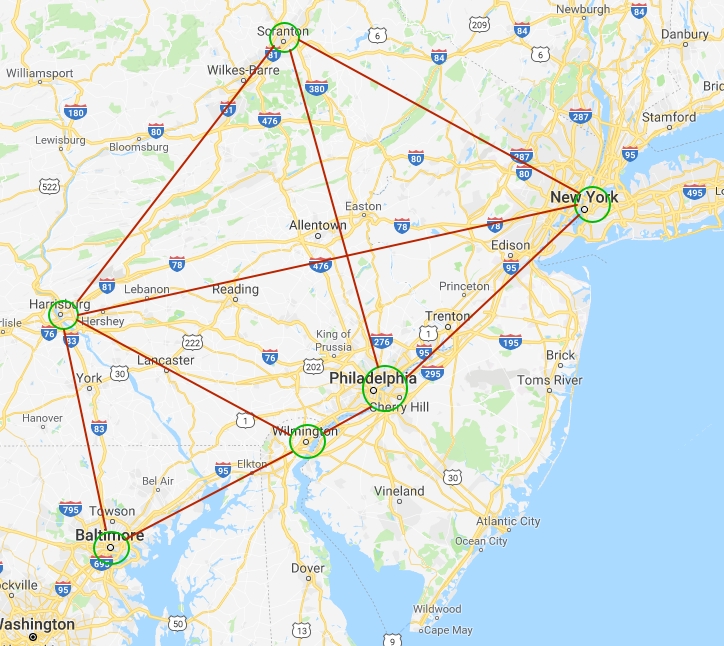
A graph is a structure which consists of vertices and edges between them. Graph theory finds its application in various fields such as computer science, linguistics, physics, chemistry, social sciences, biology, mathematics, and others. Problems connected with graph analysis are rather complicated, but there are many modern convenient instruments and libraries for these purposes.

In this post, we will consider the following example of the graph: the cities are the vertices and the distances between them are the edges. You can see the [Google Maps](https://maps.google.com/) illustration of this structure in the figure below.



To start the work with the graph mentioned above, let’s launch Spark shell. To do this go to the Spark Home Directory and type in the console:

1. ./bin/spark-shell

Now, we have to make some imports:

1. import org.apache.spark.graphx.Edge
2. import org.apache.spark.graphx.Graph
3. import org.apache.spark.graphx.lib.\_

Creating the property graph

To create property graph we should firstly create an array of vertices and an array of edges. For vertices array, type in your spark shell:

1. val verArray = **Array**(
2. (1L, ("Philadelphia", 1580863)),
3. (2L, ("Baltimore", 620961)),
4. (3L, ("Harrisburg", 49528)),
5. (4L, ("Wilmington", 70851)),
6. (5L, ("New York", 8175133)),
7. (6L, ("Scranton", 76089)))

The attributes of the vertices mean the city name and population, respectively.

As the output you will see the following:

1. verArray: Array[(Long, (String, Int))] = **Array**((1,(Philadelphia,1580863)), (2,(Baltimore,620961)), (3,(Harrisburg,49528)), (4,(Wilmington,70851)), (5,(New York,8175133)), (6,(Scranton,76089)))

To create edges array, type in the spark shell:

1. val edgeArray = **Array**(
2. **Edge**(2L, 3L, 113),
3. **Edge**(2L, 4L, 106),
4. **Edge**(3L, 4L, 128),
5. **Edge**(3L, 5L, 248),
6. **Edge**(3L, 6L, 162),
7. **Edge**(4L, 1L, 39),
8. **Edge**(1L, 6L, 168),
9. **Edge**(1L, 5L, 130),
10. **Edge**(5L, 6L, 159))

The first and the second arguments indicate the source and the destination vertices identifiers and the third argument means the edge property which, in our case, is the distance between corresponding cities in kilometers.

The above-mentioned input will give us the following output:

1. edgeArray: Array[org.apache.spark.graphx.Edge[Int]] = **Array**(**Edge**(2,3,113), **Edge**(2,4,106), **Edge**(3,4,128), **Edge**(3,5,248), **Edge**(3,6,162), **Edge**(4,1,39), **Edge**(1,6,168), **Edge**(1,5,130), **Edge**(5,6,159))

Next, we will create RDDs from the vertices and edges arrays by using the sc.parallelize() command:

1. val verRDD = sc.**parallelize**(verArray)
2. val edgeRDD = sc.**parallelize**(edgeArray)

We are ready to build a property graph. The basic property graph constructor takes an RDD of vertices and an RDD of edges and builds a graph.

1. val graph = **Graph**(verRDD, edgeRDD)

Now we have our property graph, and it is time to consider basic operations which can be performed with graphs such as filtration by vertices, filtration by edges, operations with triplets and aggregation.

Virtual Machines for data science

[Download](https://vm.datascience-school.com/)

Filtration by vertices

To illustrate the filtration by vertices let’s find the cities with population more than 50000. To implement this, we will use the filter operator:

1. graph.vertices.filter {
2. **case** (id, (city, population)) => population > 50000
3. }.collect.foreach {
4. **case** (id, (city, population)) =>
5. **println**(s"The population of $city is $population")
6. }

And this is the result we get:

The population of Scranton is 76089

The population of Wilmington is 70851

The population of Philadelphia is 1580863

The population of New York is 8175133

The population of Baltimore is 620961

Triplets

One of the core functionalities of GraphX is exposed through the triplets RDD. There is one triplet for each edge which contains information about both the vertices and the edge information. Let’s take a look through graph.triplets.collect.

As an example of working with triplets, we will find the distances between the connected cities:

1. **for** (triplet <- graph.triplets.collect) {
2. **println**(s"""The distance between ${triplet.srcAttr.\_1} and
3. ${triplet.dstAttr.\_1} is ${triplet.attr} kilometers""")
4. }

As a result, you should see:

The distance between Baltimore and Harrisburg is 113 kilometers

The distance between Baltimore and Wilmington is 106 kilometers

The distance between Harrisburg and Wilmington is 128 kilometers

The distance between Harrisburg and New York is 248 kilometers

The distance between Harrisburg and Scranton is 162 kilometers

The distance between Wilmington and Philadelphia is 39 kilometers

The distance between Philadelphia and New York is 130 kilometers

The distance between Philadelphia and Scranton is 168 kilometers

The distance between New York and Scranton is 159 kilometers

Filtration by edges

Now, let’s consider another type of filtration, namely filtration by edges. For this purpose, we want to find the cities, the distance between which is less than 150 kilometers. If we type in the spark shell,

1. graph.edges.filter {
2. case **Edge**(city1, city2, distance) => distance < 150
3. }.collect.foreach {
4. case **Edge**(city1, city2, distance) =>
5. **println**(s"The distance between $city1 and $city2 is $distance")
6. }

we will see the next result:

The distance between 2 and 3 is 113

The distance between 2 and 4 is 106

The distance between 3 and 4 is 128

The distance between 4 and 1 is 39

The distance between 1 and 5 is 130

Aggregation

Another interesting task which can be considered here is aggregation. We will find total population of the neighboring cities. But before we start, we should change our graph a little. The reason for this is that GraphX deals only with directed graphs. But to take into account edges in both directions, we should add the reverse directions to the graph. Let's take a union of reversed edges and original ones.

1. val undirectedEdgeRDD = graph.reverse.edges.**union**(graph.edges)
2. val graph = **Graph**(verRDD, undirectedEdgeRDD)

Now we have an undirected graph with all the edges and directions taken into account, so we can perform the aggregation using [aggregateMessages](https://spark.apache.org/docs/latest/graphx-programming-guide.html#aggregate-messages-aggregatemessages) operator:

1. val neighbors = graph.aggregateMessages[Int](ectx => ectx.**sendToSrc**(ectx.dstAttr.\_2), \_ + \_)

To see the result, type:

1. neighbors.**foreach**(**println**(\_))

You should get the following output,

(4,2251352)

(1,8322073)

(5,1706480)

(2,120379)

(6,9805524)

(3,8943034)

where the first argument is the vertex id and the second argument is the total population of the neighboring cities.

Conclusions

GraphX is very useful Spark component which has many applications in different fields, from computer science to biology and social sciences. In this post, we have considered the simple graph example where the vertices are the cities and the edges are the distances between them. Some basic operations such as filtration by vertices, filtration by edges, operations with triplets and aggregation have been applied to this graph. All in all, we showed that Apache Spark GraphX component is very convenient and applicable for graph computations.