An implementation of Dijkstra's algorithm on MapReduce and Spark

MAP543 – Database Management II Project

Daniel Wientjens, Grégoire Virepinte

1. A quick description of the Dijkstra algorithm

1.1. Background

Dijkstra's algorithm is an algorithm which determines the shortest path between two nodes in a graph, and which was made in 1956 by Edsger Dijkstra, a Dutch computer scientist. Though the original algorithm determines the shortest path between two nodes, the variant we will be using is the **fixed single node variant in an unoriented graph**. In this case, we attempt to find the shortest distance between a single fixed node and all the other nodes, which produces a shortest path tree.

In our two implementations (MapReduce and Spark), we arbitrarily take node 1 to be our fixed node.

1.2. Real world applications

The graph over which the algorithm works can be seen as numerous practical real world applications:

- The most obvious one is to **find the shortest path between two points on a geographical map**, where crossroads are represented as nodes and roads are represented as lines, with weight equal to the time needed to go from one node to another.
- Another very useful application would be in **IP routing**, which allows two or more devices on different TCP/IP network to communicate with one another.
- A less common application would be that of **arbitrage trading**. For instance, suppose that one could convert $1,000$ USD to 940 EUR, and then 940 EUR to $1,050$ CAD, which you could convert back to $1,010$ USD. This problem could be represented as a shortest path problem: in the corresponding graph, the currencies would be the nodes, and the exchange rates would be represented the weights between each node.

1.3. Details

The graph is first of all converted into an **adjacency matrix**, where the first two columns represent the start and end nodes respectively and the third column represents the weights of the path betweent the two nodes in question. This matrix is used as the input for the Dijkstra algorithm.

The Dijkstra algorithm will take the fixed point and calculate the distances to a new point based on the adjacency matrix to every node where there is a connection and updates that distance if a shorter one is found i.e. with less weight. In the example we have been working with primarily, we have attributed a weight of 1 to every edge, but it could also be the case that the shortest path goes over several nodes with a lower weight compared to a direct connection with a higher weight.

To enable this, the algorithm searches exhaustively to find a shorter path if one exists. To do this a set is made to keep track of all the unvisited nodes where the distance by default is infinity, and 0 for the initial fixed point. The tentative distance from our initial node to its unvisited neighbours is then calculated and compared to the stored distance (infinity in the first case) the smaller distance is then kept. The visited

neighbours are taken out of the unvisited set and the procedure described above is repeated for the node in the unvisited set with the smallest tentative distance.

2. The implementation on MapReduce

As explained earlier, the input will take the form of an **adjacency matrix**, where the first column is the start node, the second column is the end node, and the third column is the weight (in our example, all weights are equal to 1).

Please note that we have worked on the **undirected variant** of the Dijkstra algorithm: that means that saying that there exists a connection from node a to node b with weight ω is equivalent to saying that there is a connection from node b to node a with weight ω. In other words, **all connections between nodes are symmetrical**.

Our MapReduce implementation will use three different programs:

- One first "basic" mapper to help us shape the initial adjacency matrix into a format we can work with,
- One mapper,
- One reducer.

We use the same reducer after the "basic" mapper and after the regular mapper.

2.1. Initialisation mapper

The initialisation mapper has two objectives:

- 1- Transform the initial adjacency matrix into an output that we can work with more easily with our main program,
- 2- Generate new lines with infinite distance between the source node and nodes that weren't initially connected to the source node.

2.1.1. Transformation of the adjacency matrix

The adjacency matrix holds all the information we need to reconstitute the graph we are working on, but it lacks a bit of information for the Dijkstra algorithm to work properly – namely the path taken to go from node a to node b, as well as a flag to determine whether or not a given node has been visited previously in the algorithm.

For this reason, we want to transform the matrix so that we can get the following type of output for every line:

```
(a,b  DISTANCE  a,b  (UN)VISITED)
```

Where all four columns are separated by a tab. In this output:

- a and b are two nodes that are connected together,
- DISTANCE corresponds to the **distance** between those two nodes,
- The second a, b corresponds to the **path** that is being taken to go from node a to node b,
- The last variable is a **flag** that takes the value 0 when either a or b is equal to 1, and 1 otherwise.

When producing the output, we use a little tweak: whenever b is actually our source node, **we reverse its position with** a, so as to get 1 as our first character – which will be much more convenient a bit later, when we sort the lines by nodes.

2.1.2. Generation of new lines with infinite distance

Another challenge will be to be able to print new lines between the source node and other nodes that were not initially connected with the source node, with infinite distance (which, in our code, was capped to 9999, an arbitrary large number). Indeed, there is no way to know *a priori* which are the nodes that are not connected to the source node. It also seems quite difficult to get beforehand a full list of the nodes that belong to the graph, given only the adjacency matrix.

Given this challenge, we found a few possible solutions to solve our problem:

- The first one would be to **store any node we find in our graph inside of a Python set** (to avoid duplicates), and then generate new lines with infinite distance with all nodes that were found in our set. The problem with that solution is that it is hardly scalable, since we may find a graph with too many nodes to store in our memory – and this would return an error.
- A second solution would be to **tamper with the adjacency matrix beforehand**, so as to generate the new lines artificially. We decided against this solution because we wanted to keep the input as clean as possible, so that the user does not have to worry about adding in connections that do not exist in the graph they are trying to convert into an adjacency matrix.
- A third solution, which is the one we have adopted in our mapper, is to **print two new lines for every line** that we find that does not involve the source node.

For instance, assume that we are working on the line "4;274;1" in our algorithm – that is, there is a line from node 4 to node 274 with weight 1. Since the node 1 is not involved in this line, our mapper will print the following three lines:

```
## 4,274           1    4,274   1

## 1,274        9999    1,274   0

## 1,4      9999    1,4 0
```

This ensures that we connect every node to the source node. Of course, there is a small price to pay: namely, we will print an unnecessary high number of lines. However, this is the compromise that we chose because contrary to the first solution, we do not need to store anything inside the memory.

2.2. Initialisation reducer (and Dijkstra reducer)

After this initialisation mapper, we run a reducer so as to remove any unncessary lines. Note that we use the exact same reducer after the initialisation phase and after the main mapper.

The goal of our reducer is to remove any duplicate connexions between any two given nodes. For example, let us assume that during the previous mapping phase, we ended up with the following lines:

```
## 1,3      9999    1,3 0

## 1,4         1    1,4 0

## 1,4      9999    1,4 0

## 1,4      9999    1,4 0

## 1,4      9999    1,4 0

## 1,4      9999    1,4 0

## 1,5      9999    1,5 0

## 4,5         1    4,5 1
```

We notice that there are quite a few duplicate lines: indeed, the $1, 4$ connection was repeated five times. The aim of our reducer will be to keep the first line (since the distance is equal to 1, and is the lowest distance among the five), and get rid of all four other lines.

To achieve this, we take advantage of the **shuffle-and-sort** mechanism that takes place between the mapping phase and the reducing phase. Indeed, thanks to the arrangement of our columns, we have managed to sort our mapping output by node connection: as such, all $1, 4$ connections are going to be next to one another inside of the input of our reducer.

Our reducer is therefore going to work in a very straightforward manner: for every line, we keep track of the connection we are working with.

- If the connection is the same as the one from the line before, we compare the distance of the previous line with that of the current line, and keep the shortest distance and the according path.
- However, if the connection is different, we print the previous connection, along with the according distance and flag, and go on to the next connection.

Once we run the reducer, we end up with this type of output:

```
## 0        9999    1,3 1,3
## 0           1    1,4 1,4
## 0        9999    1,5 1,5
## 1           1    4,5 4,5
```

The first column is our (un)visited flag, the second column is the distance, the third column keeps track of the starting and ending node, and the fourth column is the path taken by the algorithm.

2.3. Main mapper

One may note that the order of the columns from the initialisation mapper is different from the order of the columns from the reducer. This is solely because we want to take advantage of the shuffle-and-sort mechanism in a different way in our two cases:

- For the reducer, we wanted to sort our output by connections, so we put that as our first column,
- However, for our main mapper, we want to sort our output firstly by the (un)visited flag, and secondly by the distance. This ensures that **we always find at the top of our input the unvisited node with the lowest distance to the source node**.

Thanks to the shuffle-and-sort, we end up with the following input:

```
## 0           1    1,4 1,4
## 0        9999    1,3 1,3
## 0        9999    1,5 1,5
## 1           1    4,5 4,5
```

This means that the node we will be visiting for this iteration will be node 4. At this point, our aim will be to find all neighbours to node 4, and find their distance to the source node.

To do that, we loop over all lines, and for every line, we check whether the source node or the destination node is equal to node 4. In our example, we find that there is an obvious candidate: the fourth line, with nodes $4, 5$. We then add the distance to go from node 1 to node 4 and the destination to go from node 4 to node 5 in order to get a tentative distance from node 1 to node 5, and we update the corresponding path. This would go as such:

```
## 0           1    1,4 1,4
```

```
## 0        9999    1,3 1,3
## 0        9999    1,5 1,5
## 1           2    1,5 1,4,5
```

As you may note, in the fourth line, we have updated the distance, the path, and also the start node and destination nodes.

The final output of the mapper will be as such:

```
## 1,4         1    1,4 1
## 1,3      9999    1,3 0
## 1,5      9999    1,5 0
## 1,5         2    1,4,5   1
```

2.4. Main reducer

As we mentionned in part 2.2., the reducer we use after the initialisation mapper is the same as the reducer we use in this part, so the basic functioning of the program is still valid.

After the reducer, we end up with the following output:

```
## 0        9999    1,3 1,3
## 1           1    1,4 1,4
## 0           2    1,5 1,4,5
```

The graph has thus been updated with a better distance for node 5.

We then iterate the algorithm again and again until all nodes have been visited by the algorithm.

2.5. A few drawbacks of the algorithm

We have found two main drawbacks in this algorithm.

- The first issue is that our particular implementation of the algorithm **does not work very well with parallelisation**, specifically with the mapper. In fact, for every iteration, our code retrieves the working node because it is the first line of the input – this implies that this input cannot get divided into several chunks. This also suggests that our specific implementation would probably not scale very well.
- The second and most important issue to our mind is that **MapReduce and Hadoop do not support iterative tasks very well**, which is problematic since the Dijkstra algorithm is by nature an iterative algorithm. This suggests that using MapReduce for that might be a mismatch. In fact, it took us more than a hundred pipes on a local setting to manage to get the final result for the graph example that was provided – doing the same thing in MapReduce, and running the same tasks hundreds of time may not be very efficient, especially since the results of every iteration will be written into memory each time.

3. The implementation on Spark

The main issue of MapReduce is that it does not bode well with iterative tasks. On the other hand, Spark's RDDs were made to solve this type of issue, which makes it a perfect candidate for the shortest path problem.

3.1. Preparation

We firstly load the input inside of Spark. Please note that the input is still the exact same adjacency matrix as the one we already used in MapReduce.

In this case, our goal will be to transform this matrix into that type of input for every line of the matrix:

```
print("['1,4', (1, '1,4', 0)]")
```

```
## ['1,4', (1, '1,4', 0)]
```

The key is $'1,4'$, which corresponds to the connection between a starting node and a destination node. The value is comprised of three elements:

- The first element, 1, is the distance between nodes 1 and 4,
- The second element, $1,4$, is the path taken to go from 1 to 4,
- The third element, 0, is the flag to determine whether node 4 has been visited or not.

We then run a function, `initial_transformer`, to help us **generate new connections between the source node and nodes that weren't initially connected to it**, very similarly to what we have already done in the MapReduce algorithm. Again, we did not want to try to store the nodes into a dictionary, because the number of nodes could potentially be too high to keep in memory. We then run a `reduceByKey` because similarly to our MapReduce initialisation, we end with a very high number of lines.

3.2. The iteration

At the initialisation, we define by default the vertex as 1, the path taken as 1, and the distance to the source as 0.

We then run the following transformations:

- The `transformer` function's output makes sure that we can easily specify the three possible outcomes for a row instead of a complex lambda function using a complex if-else structure. With the transformer function we aim to search for a common key between the vertex searched and its antecedent. We loop over each arc and if the source vertex (aka fixed source point) and the destination is defined in the search we remove the destination vertex from the key. If the source vertex is the vertex defined in the key we remove the second node in the key since we don't have any use for it. In all other cases we do not change anything.
- Then we keep track of the last distance in the algorithm, along with the current vertex and corresponding path by means of looping over a while function using the previously mentioned two functions.

Further comments can be found in the code located in the Appendix.

3.3. Strong and weak points

The main strong point of the Spark implementation as opposed to the MapReduce algorithm is that it actually supports iteration very well, which actually makes it viable to use this solution to run the Dijkstra algorithm.

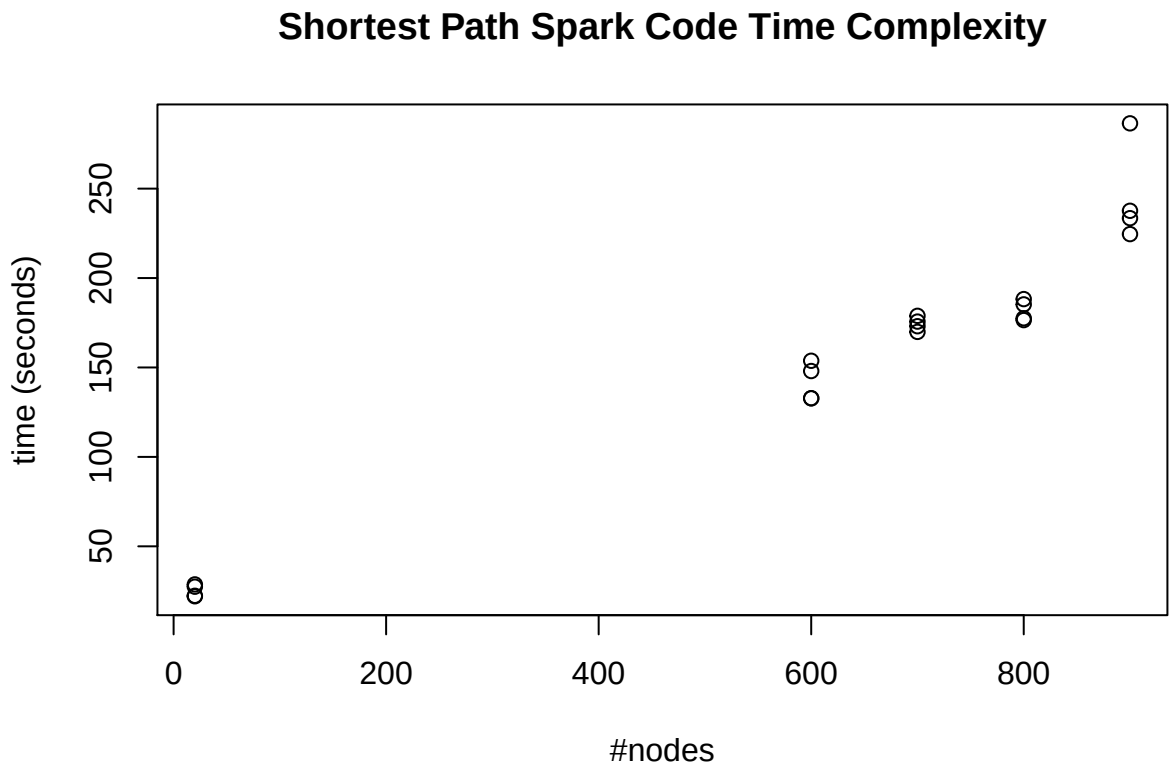
On the other hand, as we will see in the next section, our implementation of the algorithm does not seem to scale *that* well on Spark. We ran our code on the sample graph (with around a thousand nodes), and while the result appears to be correct, it took us around 260 to get it with the Spark Hadoop streaming, which seems to be quite a long time.

4. Experimental Analysis

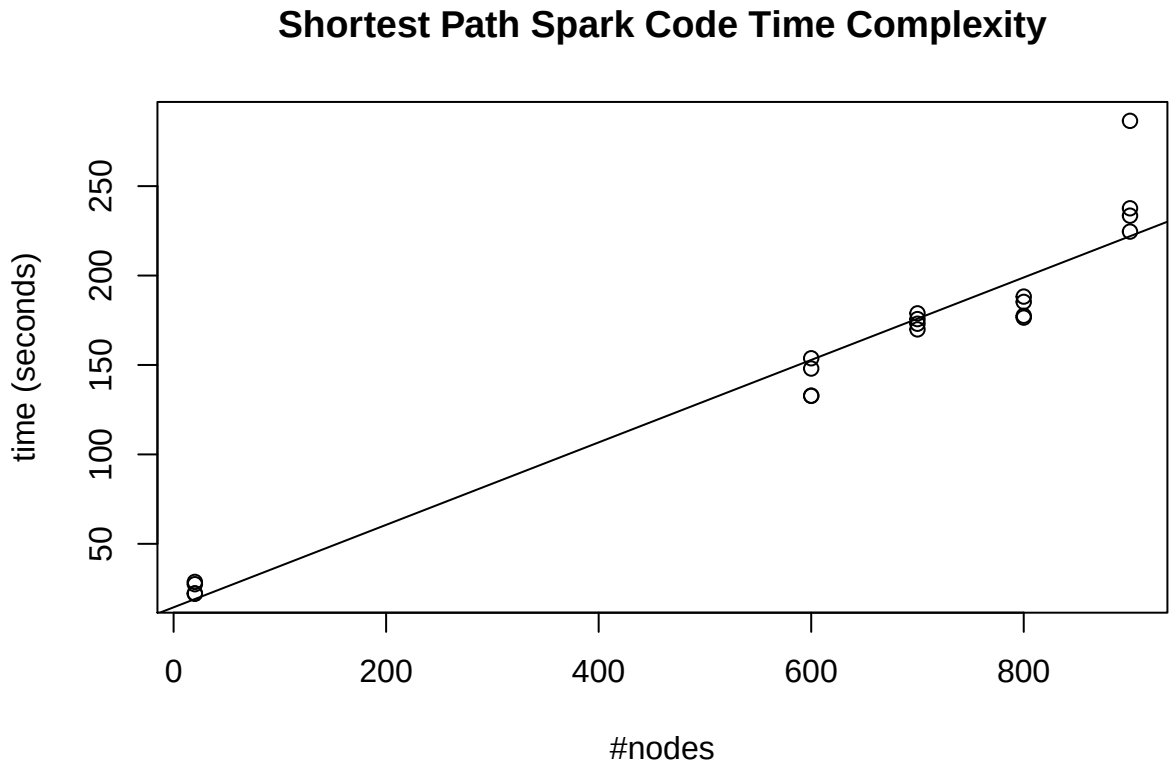
4.1. Time complexity

We performed some further analysis on how our Spark code scales with time by running the code with a different adjacency matrices where the only difference is the amount of nodes. The amount of rows was taken as a constant to ensure that result does not get influenced by the number of iterations possible. So basically we are keeping the number of edges constant but are increasing the number of vertices. The result can be seen below.

```
x<- matrix(data=1,nrow=20,ncol=2)
colnames(x)<-c("#nodes","time (seconds)")
x[1:4,1]<-20
x[1,2]<-22.08
x[2,2]<-28.71
x[3,2]<-22.34
x[4,2]<-27.43

x[5:8,1]<-600
x[5,2]<-153.75
x[6,2]<-132.74
x[7,2]<-132.83
x[8,2]<-148.01

x[9:12,1]<-700
x[9,2]<-178.87
x[10,2]<-169.89
x[11,2]<-173.06
x[12,2]<-175.65

x[13:16,1]<-800
x[13,2]<-188.23
x[14,2]<-185.27
x[15,2]<-176.49
x[16,2]<-177.51

x[17:20,1]<-900
x[17,2]<-286.49
x[18,2]<-224.57
x[19,2]<-233.47
x[20,2]<-237.61

plot(x, main="Shortest Path Spark Code Time Complexity")
```




```
x<- as.data.frame(x)
lm1<-lm(x$`time (seconds)`~x$`#nodes`)
lm2<-lm(x$`time (seconds)`~x$`#nodes`+I(x$`#nodes`)^2)
plot(x, main="Shortest Path Spark Code Time Complexity")+abline(lm1)
```




```
## integer(0)
```

As we can see from the graphs above the process seems roughly linear but if more nodes are added and the number of edges stay constant the higher the number of unconnected nodes gets. This will also have an impact on the running time of the algorithm and should be further experimented with. Unfortunately this falls a bit outside of the scope of our project and will be left to future researchers.

4.2. Stability

Stability-wise, we managed to run 5 input matrices that differ in amount of nodes but not in the amount of edges without problem. Furthermore we built a break into the while loop in case of a ValueError.

5. Appendix

5.1. MapReduce Code

5.1.1. Basic Mapper

```
#!/usr/bin/python
#  This is the first mapper, that we use to transform the initial matrix into something
# we can work with.
#  The end result consists in four columns, as follows:
# 1- A flag (0 if the departure node is the source and the line not treated yet,
# 1 otherwise),
# 2-, The distance between the two nodes,
# 3- The departure and arrival node, and 4- The optimal path to get to the last node
import sys
for line in sys.stdin: # Split the sparse matrix into four columns:
    line = line.strip().split(';') # Departure node, arrival node, distance, and departure node again

    if line[0] == '1': # If the starting node is 1, print it as normal
        print("%s,%s\t%s\t%s,%s\t0" % (line[0], line[1], str.rjust(line[2], 8), line[0], line[1]))

    elif line[1] == '1': # Similar
        print("%s,%s\t%s\t%s,%s\t0" % (line[1], line[0], str.rjust(line[2], 8), line[1], line[0]))

    else:
        print("%s,%s\t%s\t%s,%s\t1" % (line[0], line[1], str.rjust(line[2], 8), line[0], line[1]))
        # We firstly print the line as normal.
        # We then add two lines that are needed for the initialisation of the Dijkstra algorithm.
        # Indeed, at the start, we need to set up the distance between the source and the nodes
        # that aren't connected to it to + infinity (9999 in our program).
        # We have no way from the start to know which nodes they are, so for every arc that do not
        # involve the source, we print the following lines:
        print("1,%s\t%s\t1,%s\t0" % (line[1], str.rjust('9999', 8), line[1]))
        print("1,%s\t%s\t1,%s\t0" % (line[0], str.rjust('9999', 8), line[0]))

else:
  print("1,1\t%s\t1,1\t0" % str.rjust("0", 8)) # Print the first line (distance of 0 between the
  # node 1 and 1, may be necessary for the stability of the algorithm)
```

5.1.2. Main mapper

```
#!/usr/bin/python
import sys
# With the previous mapper, we sort our input over the columns 'flag' and 'distance'.
# This means that the first line is going to be the node with the shortest distance to the source,
# that has not yet been treated in the algorithm.
current_node = None
for line in sys.stdin:
    line = line.strip().split('\t')
    line[1] = int(line[1]) # This line is necessary to add two distances together (a bit later)
    if current_node == None and line[0] == '0': # If the line is the first line, and the node was
    # not treated by the algorithm yet
        current_node = line[2].split(',')[-1] # Set the current node to the first node
        # (once for the whole mapper)
        distance = line[1]
        path = line[3]
        print("%s\t%s\t%s\t1" % (line[2], line[1], line[3]))
    elif line[2].split(',')[0] == current_node and line[0] == '1': # If the first node is equal
    # to the current node:
        print("1,%s\t%s\t%s\t1" % (line[2].split(',')[1], line[1] + distance, path + ',' + line[3].spli
        # Add distance and modify the path
    elif line[2].split(',')[1] == current_node and line[0] == '1': # If the second node is equal
    # to the current node (since the graph is non-directed):
        print("1,%s\t%s\t%s\t1" % (line[2].split(',')[0], line[1] + distance, path + ',' + line[3].spli
    else:
        print("%s\t%s\t%s\t%s" % (line[2], line[1], line[3], line[0])) # If the line is 'normal',
        # print it without any modification
```

5.1.3 General purpose reducer

```
#!/usr/bin/python
import sys
# For the last output, we put the third column (starting node, arrival node) first because
# we wanted to sort over it.
# For every line, we check if the (starting node, arrival node) is equal to that of the previous line
# If it is the case, we keep the minimum distance between both arcs, along with the corresponding path.
# Otherwise, we print the line as normal, and put the columns in their original order (to sort over
# our flag and over the distance).
last_line = None
for line in sys.stdin:
    line = line.strip().split('\t')
    if last_line == None: # If this is the first line in the algorithm
        last_line = line
    # For the next two 'elif' instructions: if the starting and the ending nodes are the same for
    # two consecutive lines:
    elif line[0] == last_line[0] and int(line[1]) < int(last_line[1]): # If the distance for the
    # first line is higher:
        last_line = line # Do not print the first line
    elif line[0] == last_line[0] and int(line[1]) >= int(last_line[1]): # If the distance for the
    # second line is higher:
        last_line[3] = 0
```

```
        continue
    else: # Otherwise, if the two consecutive lines have different starting or ending nodes:
        print("%s\t%s\t%s\t%s" % (last_line[3], str.rjust(str(last_line[1]), 4), last_line[0], last_lin
        # Print line as normal
        last_line = line
else: # Do not forget to print the last line!
    print("%s\t%s\t%s\t%s" % (line[3], str.rjust(str(line[1]), 4), line[0], line[2]))
```

5.2. Spark Code

```
import time
start_time = time.time()
def transformer(x):
    """
    This function will be used inside of one of our maps. We use this instead of a lambda
    because the output is relatively complex (three possible outcomes, instead of the usual
    two with an if ... else structure).
    We loop over each arc:
        - If the source vertex is 1 (starting vertex) and the destination is the vertex defined
          in our search: we remove the first node in the key.
        - If the source vertex is the vertex defined in our search: we remove the second node in the ke
        - Otherwise, we make no change.
    The aim is to get a common key between the vertex searched and its antecedent.
    """
    if x[0].split(',')[0] == '1' and x[0].split(',')[1] == vertex:
    # If the source vertex is 1 and the destination is the vertex searched
        return(x[0].split(',')[1], x[1])
    elif x[0].split(',')[0] == vertex: # If the source vertex is the vertex we search
        return((x[0].split(',')[0], (x[1][0] + distance, path + ',' + x[1][1].split(',')[-1], x[1][2]))
    elif x[0].split(',')[-1] == vertex: # If the second vertex is the vertex se search
        return((x[0].split(',')[1], (x[1][0] + distance, path + ',' + x[1][1].split(',')[0], x[1][2])))
    else:
        return(x[0], x[1]) # No transformation
def initial_transformer(x, ind):
    if x[0].split(',')[ind] == '1':
        return('1,'+x[0].split(',')[ind], (0, '1', 0))
    else:
        return('1,'+x[0].split(',')[ind], (999, '1,'+x[0].split(',')[ind], 0))
# Get the initial document
document = sc.textFile("hdfs:///user/hadoop/wc/input/input_matrix.txt")
document = document.map(lambda x: x.split(";"))
# Transform the document to get the following output:
# (node1, node2, distance, node1, node2, (un)visited)
# (the last two elements are here to keep track of the path)
matrix = document.map(lambda x: (x[0] + ',' + x[1], (int(x[2]), x[0] + ',' + x[1], 0)) if x[1] != '1' e
matrix = matrix.map(lambda x: (x[0], (x[1][0], x[1][1], 1)) if x[0].split(',')[0] != '1' else x)
matrix = matrix.union(matrix.map(lambda x: initial_transformer(x, 0)))
matrix = matrix.union(matrix.map(lambda x: initial_transformer(x, -1)))
matrix = matrix.reduceByKey(lambda x, y: (x[0], x[1], x[2]) if x[0] < y[0] else (y[0], y[1], y[2]))
# Keep track of the last distance in the algorithm, along with the current vertex and corresponding pat
vertex = '1'
distance = 0
```

```
path = '1'
while True:
    # First transformation (see description of the function to learn more)
    trans_matrix = matrix.map(lambda x: transformer(x))
    # Retrieve the proper key
    map_matrix = trans_matrix.map(lambda x: (x[0], x[1]) if len(x[0].split(',')) > 1 else ('1,' + x[1][
    # Merge everything
    reduce_matrix = map_matrix.reduceByKey(lambda x, y: (x[0], x[1], 0) if x[0] < y[0] else (y[0], y[1]
    # Update the current node
    try:
        current_node_RDD = reduce_matrix.filter(lambda x: x[1][2] == 0).sortBy(lambda x: x[1][0])
        current_node = current_node_RDD.first()
        vertex = current_node[0].split(',')[-1]
        distance = current_node[1][0]
        path = current_node[1][1]
        if matrix == reduce_matrix:
            break
        else:
            reduce_matrix = reduce_matrix.map(lambda x: (x[0], (x[1][0], x[1][1], 1)) if x == current_n
            matrix = reduce_matrix
    except ValueError:
        break
print('Time:', time.time() - start_time) # Keep track of the time needed
```