Solving for the Single Source Shortest Path Problem using Dijkstra: MapReduce Chaining and Spark-Graphx

By
Sofiene Omri
Hamza Ferchichi
Taieb Jlassi

Under the supervision of
Prof. Dario Colazzo

MARCH 8, 2019
Paris-Dauphine University

List of Tables

List of Figures

Contents

1. Introduction

Recent decades have witnessed a sudden growth in the amount of data collected. For networks and graphs, the rise of social networks (Twitter, Wiki, and Facebook), gaming network (Friendster), Citation networks, web graph and Temporal networks (Stack overflow) resulted in exponential data increase especially with cheaper access to the internet. Indeed, this sudden increase resulted in questioning the existing programming paradigm and technology to extract knowledge for decision making. In response to this need, Google tow revolutionary paper in cloud computing that presented the MapReduce paradigm (Dean and Ghemawat, 2008) and the Google File System (GFS) (Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, 2003). Both papers were the base for the development of Hadoop, a MapReduce implementation for reliable and scalable distributed computing that makes efficient use of its Distributed File System (HDFS) for data storage. Hadoop today is one of the leading solutions that provides the possibility for massive parallel processing of data on multiple distributed computer clusters. This is indeed can be of great use for graph processing and knowledge extraction.

One of the most interesting issues in graph theory is the single-source shortest path problem, where the task is to find the shortest paths from a source node to all other nodes in a given graph such that the sum of the weights of its constituent edges is minimized. Various algorithms depending on the graph structure have been developed to solve this problem including: Dijkstra (Dijkstra,1959), binary heap (Johnson, 1977), Fibonacci heap (Fredman and Tarjan, 1984), Bellman–Ford algorithm (Bellman,1958), Gabow's algorithm (Gabow,1985) and Thorup (Thorup, 2004) etc. The classical implementation of these algorithms can be a time-consuming task on a large-scale network. Furthermore, those past years have witnessed an increase shift towards the usage of distributed computational rand storage resources, requiring a more parallelized implementation of similar algorithms.

In this project, we focus on the MapReduce implementation of Dijkstra to solve for the single source shortest path problem for large scale graphs. The algorithm has been implemented in Hadoop streaming (Python) and Spark (Graphx, Scala). We apply the algorithm and tested for scalability on three main datasets from the Stanford Large Network Dataset Collection. All of the three selected datasets are Temporal networks where edges have timestamps thus, we are solving for the fastest time to reach all node from a source node.

On the first hand, we have concluded that the size of cluster has a direct impact on the processing time of the developed MapReduce job. Furthermore, it was noted that the developed algorithm was suffering from the burden of inter-job tasks (Verification of convergence criteria, Deleting the content of the input directory, transferring output files to input directory, Deleting the output directory during job chaining). On the other hand, Spark-Graphx proved very performant beating the MapReduce jobs on all aspects.

2. Scope of the project

The goal of this research is to illustrate the MapReduce and spark implementation of the Dijkstra algorithm which is an iterative algorithm. In more details in this project we have:

- Developed a MapReduce Job for data preprocessing (transformation into an adjacency list representation).
- Developed a MapReduce job that advances the shortest path each time run.
- Developed a Linux script to iteratively run the Dijkstra MapReduce job until a convergence criterion is met.
- Created an EMR cluster on AWS for experimentation
- Made use of predeveloped libraries in Spark to efficiently implement the Dijkstra algorithm.

3. Single Source Shortest Path Problem

The single-source shortest path problem aims at finding the shortest paths from a source node to all other nodes in a given graph such that the sum of the weights (temporal stamps in our case) of its constituent edges is minimized.

Given:

- Directed Graph with a set of vertices connected through a set of edges: G= (V, E)
- Edge weights: w: E □ R
- V_s a selected source vertex

Objective: $\forall \omega \in R$ find the shortest path (a path of minimum weights) from V_s to all other vertices V_k

3.1. Graph Representations

Graph is a data structure composed of the following two main components:

- A finite set of vertices also called as nodes.
- A finite set of edges in the form of pairs (S, N); indicating that there is an edge from vertex S to a vertex N. The edges may also contain weights (timestamps, costs, distance etc.) and in that case the graph is called weighted while the absence of weights results in unweighted graphs

It should also be noted that there are 2 main types of Graph:

- Directed Graphs: The edge pairs (S, N) are ordered. In other word (S, N) is not (N, S)
- Undirected Graphs: The edge pairs (S, N) order doesn't matter

In this project we implement Dijkstra for weighted directed graphs and more specifically temporal networks where the weights are timestamps. In case of temporal networks, for example in Stack overflow answer to questions, answer and comments are vertices and answer time is the weight. In fact, these graphs can be represented mainly in three formats as depicted below:

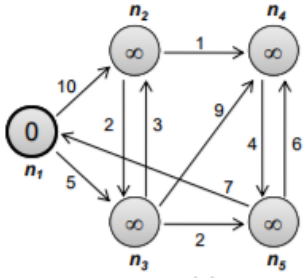
3.1.1. Edge List

An array of edges is used. The array contains N rows and three columns, where N is the total number of edges. The list contains three main components in case of weighted graphs:

- Source vertex (Node)

- Neighbor vertex (Node)
- Weight of the edge pair (S, N)

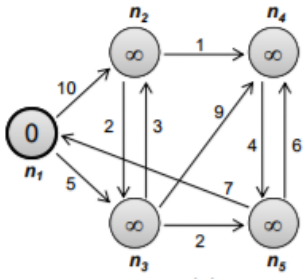
A sample representation is as follows:



Source Node	Neighbor	Weight
N1	N2	10
N1	N3	5
N2	N3	2
N2	N4	1
N3	N2	3
N3	N4	9
N3	N5	2
N4	N5	4
N5	N1	7
N5	N4	6

3.1.2. Adjacency Matrix

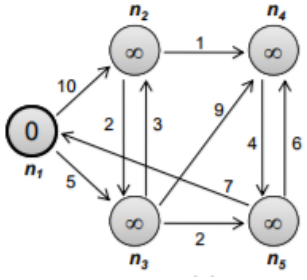
In this representation graphs are represented using a 2D matrix of size v*v where v is the total number of vertices in a given graph. Every slot strictly higher than zero indicates that there is an edge between the given vertices pair as show below:



	n1	n2	n3	n4	n5
n1	0	10	5	0	0
n2	0	0	2	1	0
n3	0	3	0	9	2
n4	0	0	0	0	4
n5	7	0	0	6	0

3.1.3. Adjacency List

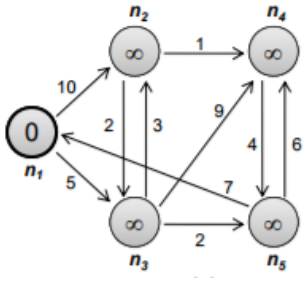
In this representation graphs are represented using a 2D array of size v*2 where v is the total number of vertices in a given graph. For every vertex a list of neighbor pairs (N: W) is given in the second column as shown below:



n1	n2:10, n3:5
n2	n3:2, n4:1
n3	n2:3, n4:9, n5:2
n4	n5:4
n5	n1:7, n4:6

3.2. Dijkstra algorithm

The algorithm was developed by Edsger W. Dijkstra in 1958 and used where the graph weights are non-negative ($\forall \omega, E \in R, \omega \geq 0$) and is a greedy one in other words the algorithm tends to always to select local minima under the assumption that this will lead to an optimal solution overall. The variant of the Dijkstra algorithm requires different graph structure with two leading representations; being an adjacency matrix and adjacency list.



Adjacency Matrix

	n1	n2	n3	n4	n5
n1	0	10	5	0	0
n2	0	0	2	1	0
n3	0	3	0	9	2
n4	0	0	0	0	4
n5	7	0	0	6	0

Adjacency List

n1	n2:10, n3:5
n2	n3:2, n4:1
n3	n2:3, n4:9, n5:2
n4	n5:4
n5	n1:7, n4:6

We can note that the matrix representation can result in increased time and space complexity for large graph which is not the case for the algorithm version based on the adjacency list. The algorithm will run until a convergence criterion is met. The algorithm will run until all path updated distance stop improving or when all nodes have been visited at least once. The simple version of the algorithm is based on the so-called priority queue to keep track of the visited and no visited vertices as shown below:

$\boldsymbol{Dijkstra}(G, s)$
    $Q = PriorityQueue$
    $\boldsymbol{Foreach}\ v \in V \boldsymbol{Do}$
        $Dist[v] = \infty$
        $Pred[v] = -1$
    $Dist[s] = 0$
    $\boldsymbol{Foreach}\ v \in V \boldsymbol{Do}$
        $Q = (v, Dist[v])$
    $\boldsymbol{While}(Q \neq \emptyset) \boldsymbol{Do}$
        $U = min(Q)$
        $\boldsymbol{Foreach}\ neighbor\ v\ of\ u\ \boldsymbol{do}$
            $W = weight\ of\ edge\ (u, v)$
            $\text{Current\_Distance} = Dist[u] + w$
            $\boldsymbol{If}(\text{Current\_Distance} < Dist[v]) \boldsymbol{then}$
                $Decrease(Q, v, \text{Current\_Distance})$
                $Dist[v] = \text{Current\_Distance}$
                $Pred[v] = u$
$\boldsymbol{End}$

Running the above algorithm will advance the known distance frontier with each iteration as shown below:

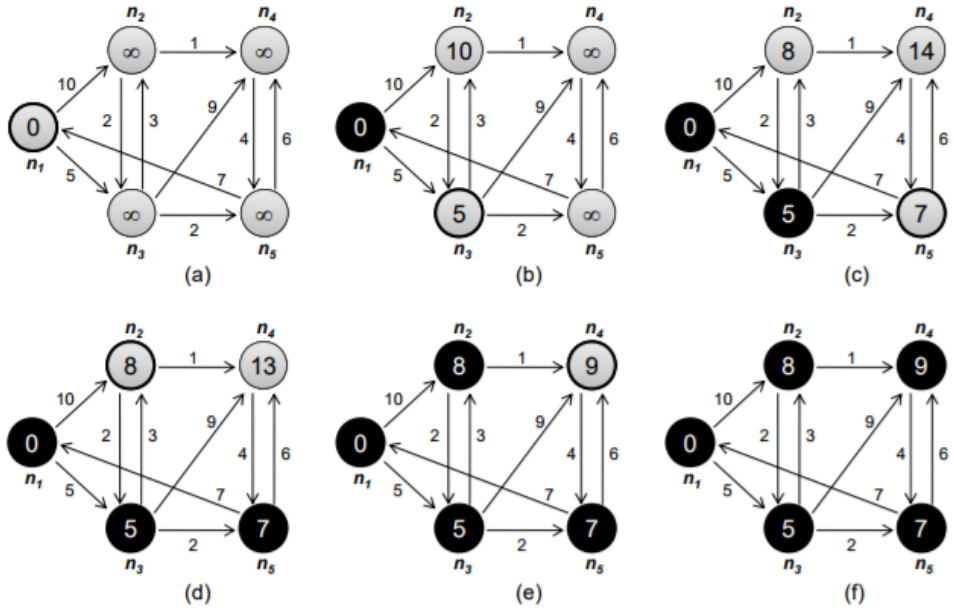


Figure 1: Sample Dijkstra iterations

3.3. From Edge list to Adjacency List: MapReduce Job

After a thorough search over the internet we have found out that all of the graphs are mainly provided in the edge list format. Of course, various shortest path algorithms such as Dijkstra require the data to be formatted in the form of Adjacency matrix or list.

On the one hand a matrix representation is the least efficient in this case as it increases space complexity. In fact, even if there is a low number of edges the space complexity will remain $O(V^2)$ where V is the number of vertices in a given graph. While on the other hand, Adjacency list actually saves more space with a complexity of $O(V + E)$, where E is the number of edges.

Given the above an efficient MapReduce representation would require an adjacency list representation instead of the edge one. In order to reformat the data and initialize the distances for our Dijkstra-MapReduce job we have developed a MapReduce job that format an edge list graph of the form (Source node, Target node, Edge weight) into an adjacency list and adds the initialized distances (0 for source node and infinity for all other nodes).

The preprocessing mapper is just read data mapper without any treatment. The mapper will receive as input an edge list and emit the following: (Key: Source Node, Value (Target Node, Weight))

$\boldsymbol{Mapper}(Node, (Neighbor, Weight))$

$\quad\boldsymbol{Emit}(Node, (Neighbor, Weight))$

```python
#!/usr/bin/python

import sys
from itertools import dropwhile

#function to identify header starting with predefined character to skip
def header(s):
    return s.startswith('#')

for line in dropwhile(header, sys.stdin):
    line = line.strip().split(' ')
    node, neighbor, weight = [str(x) for x in line]
    print('{}\t{}\t{}'.format(node, neighbor, weight))
```

Figure 2: Data Preprocessing Mapper, Python Code

The preprocessing reducer will collect neighbors for every source node into a list of the form neighbor weight pair list (N1:W1, N2:W2 etc..) and initialize distance to zero for the single source node and infinity for all other nodes.

$\boldsymbol{Reducer}(\text{Source\_node}, (Distance, neighbor\_list))$

$neighbor\_list = emptylist$

$\boldsymbol{Foreach}\ Source\_node \in V\ \boldsymbol{Do}$

$fill\ the\ neighbor\_list\ of\ \text{Source\_node}[i]$

$\boldsymbol{If}\ the\ current\ \text{Source\_node}\ is\ a\ starting\ node\ \boldsymbol{then}$

$Dist[\text{Source\_node}] = 0$

$\boldsymbol{Else}$

$Dist[\text{Source\_node}] = Inf$

$\boldsymbol{Emit}(\text{Source\_node}, (Distance, neighbor\_list))$

```python
#!/usr/bin/python

import sys

current_node = None
current_neighbors = []
starting_node = "0"
count=0


def Format_data(node, neighbors, Starting_node=False):
    # sort the neighbor list and create the path in the following format neigh:Dist,neigh2:Dist ....
    neighbors = sorted(neighbors, key=lambda x: x[0])
    path = ['{}:{}'.format(neighbor, distance) for (neighbor, distance) in neighbors]
    # set distance for strating node to zero and the rest to big number and print the expected line format
    distance = 0 if Starting_node else 9999999999999
    print('{}\t{}\t{}'.format(node, distance, ','.join(path)))


for line in sys.stdin:
    line = line.strip().split('\t')
    node, neighbor, weight = [str(x) for x in line]
    count=count+1
    if count==1:
        starting_node=node
    if node == current_node:
        current_neighbors.append((neighbor, weight))
    else:
        if current_node:
            Format_data(current_node, current_neighbors, Starting_node=(current_node==starting_node))
        current_node = node
        current_neighbors = [(neighbor, weight)]
# print the remaining data
Format_data(current_node, current_neighbors)
```

Figure 3: Data Preprocessing Reducer, Python Code

Below is the result of the execution of the above map reduce

Input to Mapper

#Source Node	#Neighbor	#Weight
N1	N2	10
N1	N3	5
N2	N3	2
N2	N4	1
N3	N2	3
N3	N4	9
N3	N5	2
N4	N5	4
N5	N1	7
N5	N4	6

Output of Reducer

n1	0	n2:10, n3:5
n2	999	n3:2, n4:1
n3	999	n2:3, n4:9, n5:2
n4	999	n5:4
n5	999	n1:7, n4:6

3.4. Dijkstra: MapReduce

As discussed earlier an important element in Dijkstra implementation is the priority queue list which maintain a list of the visited nodes. This serves as the stopping criteria given that Dijkstra is an iterative algorithm. However, it's not possible to implement such stopping criteria in MapReduce as it doesn't provide a mechanism for meta data exchange. Instead we have decided to adopt the variability of the overall distance of all nodes as a stopping criterion using a Linux script further explained later.

The developed MapReduce job works by mapping over all nodes and emitting the following:

- Complete node that contain all the data of a source node: Key: Source Node ID, Value: Distance, Neighbors-weights, Path
- Disaggregated neighbors of each complete node: key: node id of the neighbor, Value: Current distance + Weight of neighbor, Source Node

Note that in the first iteration the path will be empty and thus will be initialized to the source node by the mapper.

After shuffle and sort, reducers will receive keys corresponding to the Source node ids and distances corresponding to all paths leading to that node. The reducer will select the shortest of these distances and then update the distance in the node data structure and emits complete nodes with updated distances.

$\boldsymbol{Mapper}(Source\_node, (Distance, neighbors, Path))$

$\boldsymbol{Retreive}: Source\_node, Distance, neighbor\_list, Path(if\ not\ empty)$

$\boldsymbol{If}\ Path = empty\ \boldsymbol{then}$

$Path = Source\_Node\_Id$

$\boldsymbol{Emit}(Source\_Node, (Distance, neighbors, Path))$

$\boldsymbol{Foreach}\ Neighbor \in neighbor\_list\ \ \boldsymbol{Do}$

$\text{Neighbor\_Distance} = Source\_Distance + \text{Neighbor\_Weight}$

$\boldsymbol{Emit}(\text{Neighbor\_Node}, (Distance, Path))$

```python
#!/usr/bin/python

import sys
from itertools import dropwhile

#function to identify header starting with predefined character to skip
def header(s):
    return s.startswith('#')

for line in dropwhile(header, sys.stdin):
        line = line.strip().split('\t')
        nodeID = line[0]
        distance = int(line[1])
        # For each node check if they have neighbors or not
        if len(line) > 2:
            neighbors_list = line[2]
        else:
            neighbors_list = "empty"
        # Update previous path if it was not intialised else do so by setting all path equal to source node
        if len(line) == 4:
            path = line[3]
        else:
            path = nodeID
        # keeps the complete nodes data
        print('{}\t{}\t{}\t{}'.format(nodeID, distance, neighbors_list, path))
        # desagregate the neighbors and the relevant distance to their parent
        if neighbors_list!="empty":
            neighbors_list = neighbors_list.strip().split(',')
            for neighbors in neighbors_list:
                neighbor_nodeID, neighbor_distance = neighbors.strip().split(':', 1)
                neighbor_distance=int(neighbor_distance)
                neighbor_distance += distance
                neighbor_path = '{}->{}'.format(path, neighbor_nodeID)
                print('{}\t{}\t{}'.format(neighbor_nodeID, neighbor_distance, neighbor_path))
```

Figure 4: Dijkstra Mapper, Python Code

$$\textbf{\textit{Reducer}}\big(Source\_node, (Distance, neighbors, Path)\big)$$

$$\textbf{\textit{Retreive}}: Source\_node, Distance, Neighbor\_list, Path$$

$$\textbf{\textit{Foreach}}\ Source\_node \in V\ \textbf{\textit{Do}}$$

$$\textbf{\textit{Select}}\ Min\ Distance$$

$$Update\ Path$$

$$\textbf{\textit{Emit}}\big(Source\_node, (Distance, neighbors, Path)\big)$$

```python
#!/usr/bin/python

import sys

current_distance = None
current_nodeID = None
current_path = None
current_neighbours=None
for line in sys.stdin:
    line = line.strip().split('\t')
    nodeID = line[0]
    distance = int(line[1])
    #identify which is a neighbor noden(len()==3) and which is a source node (len()==4)
    if len(line) == 3:
        path = line[2]
    if len(line) == 4:
        path = line[3]
        current_neighbours = line[2]
    #Check for the min distance and update the source node distance and path
    if current_nodeID == nodeID:
        if distance < current_distance:
            current_distance = distance
            current_path = path
    else:
        if current_nodeID:
            print('%s\t%s\t%s\t%s' % (current_nodeID, current_distance, current_neighbours, current_path))
        current_nodeID = nodeID
        current_distance = int(distance)
        current_path = path

# the below is needed to output the last value in the reducer
print('%s\t%s\t%s\t%s' % (current_nodeID, current_distance, current_neighbours, current_path))
```

Figure 5:Dijkstra Reducer, Python Code

3.5. Chaining Dijkstra MapReduce jobs in Hadoop

Like many real-world problems, Dijkstra algorithm requires multiple jobs chained in order to reach the final frontier with the minimum distances. In fact, each single MapReduce iteration will only advance the "Known Frontier" by one hop thus subsequent iterations will include more and more reachable nodes as the frontier expands exploring the whole graph.

Map Reduce Map Reduce Map Reducer Map Reducer ……

The job chaining in this case will run until a convergence criterion is met. In this case the algorithm will run until the overall distance stop improving. In order, to implement the Dijkstra map reduce we have used a Linux Bash script the run the MapReduce job, computes the overall distance and breaks the loops when the overall distance stops improving.

Job Chaining Algorithm

$$\textbf{Retreive}: Input\_folder\_Path, Mapper, Reducer, Output\_folder\_path$$
$$\textbf{While} Convergence\_criteria\ not\ met\ \textbf{Do}$$
$$\textbf{Run}\ Dijkstra\ MapReduce\ Job$$
$$Current\_distance = \sum of all\ Distances$$
$$\textbf{If} Current\_distance\ stop\ decreasing\ \textbf{then}$$
$$Convergence\_criteria\ is\ met$$
$$\textbf{BreakLoop}$$
$$\textbf{Print}(Output)$$

```
Input_Dir="$1"
Mapper="$2"
Reducer="$3"
Output_Dir="$4"

Iterative_Dijikstra() {
    All_Distance=0
    Iter=0
    Conv_Criteria=true

    #Run the mapreduce job untill conergence criteria is met
    while $Conv_Criteria; do
        Iter=$((Iter+1))
        Current_Distance=0
        echo "#Iteration N°$Iter"
         #run the map reduce job
        hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
        -input "$Input_Dir" \
        -output "$Output_Dir" \
        -file "$Mapper" \
        -mapper "$Mapper" \
        -file "$Reducer" \
        -reducer "$Reducer"

        # the below block including the if statement checks whether the global distance stopped improving thus convergence criteria is met
        while read Dist; do
            Current_Distance=$((Current_Distance + Dist))
            done <<< "$(hdfs dfs -cat "$Output_Dir"/* | cut -d$'\t' -f2)"

        echo "#Distance Sum: $Current_Distance"

        if  [ "$Current_Distance" -eq "$All_Distance" ] &&  [ ! "$All_Distance" -eq 0 ]
            then
            Conv_Criteria= false
            break
            fi
        All_Distance=$Current_Distance
        # Move the current job output to the input dir after deleting its content
        hdfs dfs -rm -r "$Input_Dir"/*
        hdfs dfs -mv "$Output_Dir"/* "$Input_Dir"
        #he output dir also should be deleted as the hadoop jar create one each time
        hdfs dfs -rm -r -f "$Output_Dir"
        done
    # print results without neighbors
    hdfs dfs -cat "$Output_Dir"/* | sort -n |  cut -d$'\t' -f1,2,4
}

Iterative_Dijikstra
```

Figure 6: Job Chaining of Dijkstra MapReduce, Linux Code

3.6. Spark: Graphx

In order to implement Dijkstra algorithm, we were faced with tow choices. Either to develop an algorithm similar to the MapReduce or use the pre-developpe graph processing platform Graphx. After careful analysis we concluded that the best Dijkstra implementation is best optimal using Graphx. In fact, Graphx is the graph processing layer on top of Spark which is optimized for processing large graph in a timely manner it also provides a library of useful algorithms. However, Graphx is only available in Scala thus we used it to implement existing Dijkstra solution for our Graph analysis.

Actually, Graphx uses a 2 RDDs, one for vertices and the other for edges to represent a given graph. Such representation solves the issue of partitioning large graphs resulting in a huge advantage for Spark. In order, to implement the algorithm it suffices to map all vertices and edges of the input graph and initialize the source node distance to zero and the rest to positive infinity. Once done using the GraphLoader.edgeListFile and the mapVertices objects the Dijkstra algorithm is then implemented. the full code is available in Appendix B and below

```scala
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
val source_node: VertexId = 0
val graph = GraphLoader.edgeListFile(sc, "hdfs:/user/hadoop/Working_Dir/RawData_Input_Dir/email-Eu-core-temporal.txt")
val Graph_Vertices = graph.mapVertices((id, _) =>
      if (id == source_node) Array(0.0, id)
      else Array(Double.PositiveInfinity, id)
    )
val Dijkstra = Graph_Vertices.pregel(Array(Double.PositiveInfinity, -1))(
      (id, dist, newDist) => {
        if (dist(0) < newDist(0)) dist
        else newDist
      },
      triplet => {
        if (triplet.srcAttr(0) + triplet.attr < triplet.dstAttr(0)) {
          Iterator((triplet.dstId, Array(triplet.srcAttr(0) + triplet.attr, triplet.srcId)))
        }
        else {
          Iterator.empty
        }
      },
      (a, b) => {
        if (a(0) < b(0)) a
        else b
      }
    )

  val Dijkstra_output: RDD[String] = Dijkstra.vertices.map(vertex => vertex._1 + "    " + vertex._2(0) + "    " + vertex._2(1).toInt)
  Dijkstra_output.collect.foreach(println(_))
  Dijkstra_output.saveAsTextFile("hdfs:/user/hadoop/Working_Dir/output")
```

Figure 7: Spark-Graphx Scala Code

4. Scalability and Experimental Analysis

4.1. Data Description

In order to conduct experimentation, we have used three main datasets from the Stanford Large Network Dataset Collection. All of our dataset are temporal networks where edges have timestamps, in other words the weights are the timestamps.

4.1.1. Email-Eu-core temporal network (986 nodes, 332,334 temporal edges)

This graph was generated from EU research institution where a directed edge (u, v, t) means that person u sent an e-mail to person v at time t. the network contains 986 nodes and 332,334 temporal edges. Further details regarding the data can be found here.

4.1.2. Stack Overflow temporal network (2,601,977 nodes, 63,497,050 temporal edges)

This graph was generated from the interaction on the stack exchange web site Stack Overflow. The dataset provided on the SNAP website contains three different datasets including answer to questions, comment to questions and comments to answers. For example, for the answer to questions we have a directed edge (u, v, t) means that person u answered person v question at time t. In this case we have

selected the union of all the three-graph resulting in a network that contains 2,601,977 nodes and 63,497,050 temporal edges.

4.2. AWS Cluster setup

In order to run our experiments, we have created 2 different EMR clusters as follows:

- 3 nodes cluster (1 master, 2 slaves)
- 5 nodes cluster (1 master, 2 slaves)

All of the clusters are composed of general purpose m3.large instances with the following specificities:

- 64-Bit processor
- VCPU: 2
- RAM:7.5 GB
- Storage: 32 GB
- Network performance: Moderate

Cluster configuration have also included:

- S3 Bucket a storage service provided by amazon
- Generated the private key to connect to the created cluster
- Modified the security group inflow rules to allow our laptops to bypass security measures and connect to the cluster

4.3. Spark vs MapReduce on a 3-node cluster

4.3.1. MapReduce Jobs

The MapReduce job is composed of a data preprocessing job and a Hadoop streaming job. We have run the algorithm using the script in Appendix A on the Email-Eu-core temporal network dataset of 986 node.

The data preprocessing lasted for a total time of 53 second as shown below:

```
19/03/09 17:14:17 INFO streaming.StreamJob: Output directory: /user/hadoop/Working_Dir/Input_Dir

real    0m52.473s
user    0m9.430s
sys     0m0.577s
[hadoop@ip-172-31-35-27 ~]$
```

Figure 8: Data preprocessing job: overall time

This included a total estimated run time of 3 minute 12 second for a total of 16 map tasks and 1 minutes 8 second for a total of 7 reduce tasks. In other word each mapper spent **12 second** and each reducer spent **10 second**. Further detail is shown below:

```
        Job Counters
                Killed map tasks=1
                Killed reduce tasks=1
                Launched map tasks=16
                Launched reduce tasks=7
                Data-local map tasks=8
                Rack-local map tasks=8
                Total time spent by all maps in occupied slots (ms)=8623620
                Total time spent by all reduces in occupied slots (ms)=6117750
                Total time spent by all map tasks (ms)=191636
                Total time spent by all reduce tasks (ms)=67975
                Total vcore-milliseconds taken by all map tasks=191636
                Total vcore-milliseconds taken by all reduce tasks=67975
                Total megabyte-milliseconds taken by all map tasks=275955840
                Total megabyte-milliseconds taken by all reduce tasks=195768000
```

Figure 9: Data preprocessing: Job Counter details

On the other hand, the Dijkstra MapReduce job was run for 34 iterations for a total time of 34 minutes 20 seconds.

```
985     294624  1001->738->532->335->985
986     6408363 1001->738->783->66->986
987     188515  0->189->410->663->540->987
988     264019  0->189->236->90->978->988
989     66730798        1001->644->424->989
990     415696  1001->738->945->142->990
991     98686   1001->685->541->991
992     25177049        1001->506->90->746->992
993     535103  0->189->410->663->540->987->849->993
994     188515  0->189->410->663->540->994
995     1254201 1001->738->951->692->334->995
996     183689  1001->738->996
997     130021  1001->685->541->333->997
998     33529768        1001->738->783->2->998
999     12784031        1001->685->541->415->127->999
1000    0       1000
1001    0       1001
1002    0       1002
1003    498347  1001->738->923->772->454->291->1003
1004    72829   0->977->1004

real    34m19.672s
user    14m16.351s
sys     0m51.740s
[hadoop@ip-172-31-35-27 ~]$
```

Figure 10: Dijkstra job chaining output and overall time

```
        Job Counters
                Killed map tasks=1
                Launched map tasks=17
                Launched reduce tasks=7
                Data-local map tasks=16
                Rack-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=12555135
                Total time spent by all reduces in occupied slots (ms)=5616450
                Total time spent by all map tasks (ms)=279003
                Total time spent by all reduce tasks (ms)=62405
                Total vcore-milliseconds taken by all map tasks=279003
                Total vcore-milliseconds taken by all reduce tasks=62405
                Total megabyte-milliseconds taken by all map tasks=401764320
                Total megabyte-milliseconds taken by all reduce tasks=179726400
```

Figure 11: Dijkstra job chaining a sample iteration job counter

Going in details into the component of the job chaining we have found the following:

- Each iteration took on average 60 seconds

- Inter-jobs time took on average 12 seconds (including copying output to input and deleting the output directory
- Each map job took approximately 17 seconds
- Each reduce job took approximately 9 seconds

This seems fine given that the dataset is a medium one however we can note that the inter-jobs time caused a 7 minutes delay for the overall treatment.

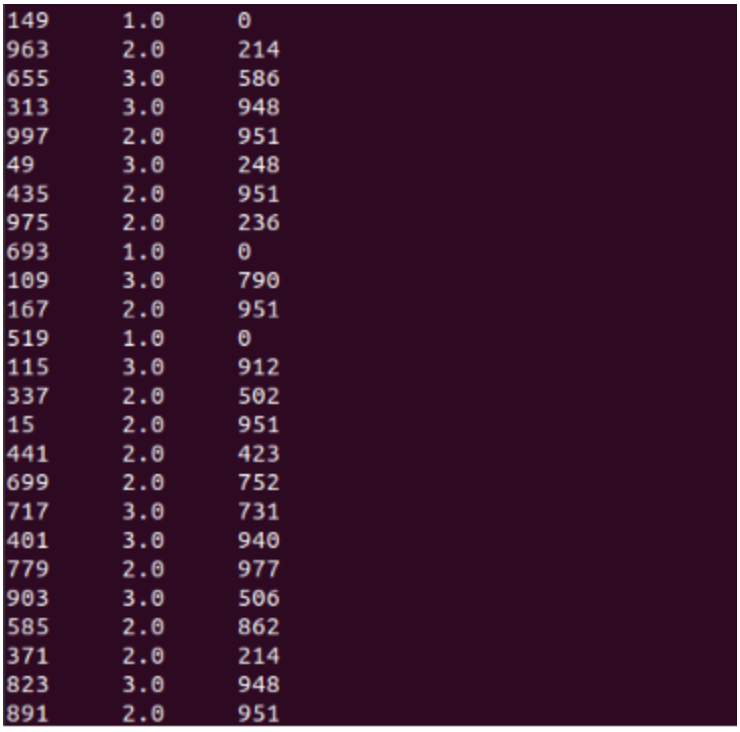
4.3.2. Spark-Graphx

Even though the MapReduce may have seemed efficient with minor delay due writing data on HDFS, the job on spark optimized graph network processor only lasted for 2 seconds (total time spent on loading edges, setting initial distances, and performing the Dijkstra algorithm) as shown below:

```
scala> var graph =time{ GraphLoader.edgeListFile(sc, "hdfs:/user/hadoop/Working_Dir/RawData_Input_Dir/email-Eu-core-temporal.txt")}
Elapsed time: 451189276ns
graph: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@1d20aae4

scala>

scala> var Graph_Vertices = time{ graph.mapVertices((id, _) =>
     |       if (id == source_node) Array(0.0, id)
     |       else Array(Double.PositiveInfinity, id)
     |     )}
Elapsed time: 73540856ns
Graph_Vertices: org.apache.spark.graphx.Graph[Array[Double],Int] = org.apache.spark.graphx.impl.GraphImpl@29863c20

scala>

scala> var Dijkstra =time{ Graph_Vertices.pregel(Array(Double.PositiveInfinity, -1))(
     |       (id, dist, newDist) => {
     |         if (dist(0) < newDist(0)) dist
     |         else newDist
     |       },
     |       triplet => {
     |         if (triplet.srcAttr(0) + triplet.attr < triplet.dstAttr(0)) {
     |           Iterator((triplet.dstId, Array(triplet.srcAttr(0) + triplet.attr, triplet.srcId)))
     |         }
     |         else {
     |           Iterator.empty
     |         }
     |       },
     |       (a, b) => {
     |         if (a(0) < b(0)) a
     |         else b
     |       }
     |     )
     | }
Elapsed time: 1167431161ns
Dijkstra: org.apache.spark.graphx.Graph[Array[Double],Int] = org.apache.spark.graphx.impl.GraphImpl@35efb878
```

Figure 12: Spark-Graphx time performance

The spark output includes the source node, distance and the previous node as shown below



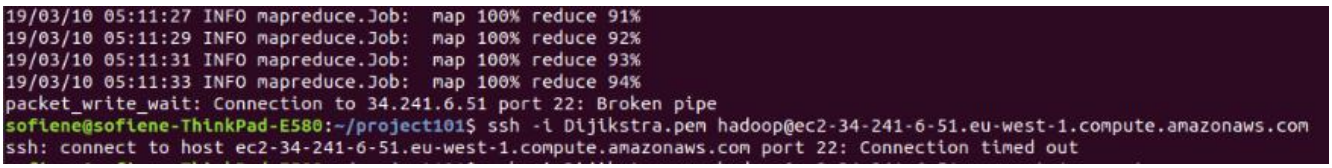
```
149     1.0     0
963     2.0     214
655     3.0     586
313     3.0     948
997     2.0     951
49      3.0     248
435     2.0     951
975     2.0     236
693     1.0     0
109     3.0     790
167     2.0     951
519     1.0     0
115     3.0     912
337     2.0     502
15      2.0     951
441     2.0     423
699     2.0     752
717     3.0     731
401     3.0     940
779     2.0     977
903     3.0     506
585     2.0     862
371     2.0     214
823     3.0     948
891     2.0     951
```

Figure 13: Spark-Graphx output

4.4. Scalability test: 3 node cluster Vs 5 node cluster

4.4.1. MapReduce Jobs

In order to test for scalability and evaluate cluster performance we have used the Stack Overflow temporal network (2,601,977 nodes, 63,497,050 temporal edges) on two different cluster with different size and same instances configurations. We have run both the preprocessing and the Dijkstra jobs on both clusters before a connection issue caused the termination of the Dijkstra job on both clusters. This was also coupled with exhausted resources due to the long run time of both clusters.



```
19/03/10 05:11:27 INFO mapreduce.Job:  map 100% reduce 91%
19/03/10 05:11:29 INFO mapreduce.Job:  map 100% reduce 92%
19/03/10 05:11:31 INFO mapreduce.Job:  map 100% reduce 93%
19/03/10 05:11:33 INFO mapreduce.Job:  map 100% reduce 94%
packet_write_wait: Connection to 34.241.6.51 port 22: Broken pipe
sofiene@sofiene-ThinkPad-E580:~/project101$ ssh -i Dijikstra.pem hadoop@ec2-34-241-6-51.eu-west-1.compute.amazonaws.com
ssh: connect to host ec2-34-241-6-51.eu-west-1.compute.amazonaws.com port 22: Connection timed out
```

Figure 14:Scalability test: Broken pipe error before reaching final results

The above error is mainly due to the ADSL in the house which resulted in the job termination before reaching the final results. Details of the Dijkstra jobs are as follows:

	3 Node Cluster	**5 Node Cluster**
Start time	2019-03-10 19:05	2019-03-10 19:45
End time (broken pipe)	2019-03-11 05:11	2019-03-11 05:11
Elapsed time	10 hr. 6 min	9 hr. 7 min
Number of iterations completed before broken pipe	135	169
Preprocessing time	3 min 46 sec	2 min 12 sec
Average preprocessing per map task	1 min 33 sec (16 map tasks)	1 min 1 sec (32 map tasks)
Average preprocessing per reduce task	52 sec (7 reduce tasks)	25 sec (29 reduce tasks)
Dijkstra: Average time per iteration	4 min 15 sec	2 min 35 sec
Dijkstra: Average time per inter-job	1 min	1 min 2 sec
Dijkstra: Average time per map task	1 min 22 sec (21 map tasks)	1 min 15 sec (32 map tasks)
Dijkstra: Average time per reduce task	1 min 23 sec (7 reduce tasks)	1 min (15 reduce tasks)

Table 1: 3 Node vs 5 Node clusters-comparative performance on MapReduce job chaining

We can note simply based on the above and even though we did not reach the final result that a 5-node cluster is far more performant than a 3 node one on every aspect of the above.

4.4.2. Spark-Graphx

Even though the MapReduce was not followed through due to unexpected issue, it still completely inefficient compared to Spark-Graphx which only lasted for few seconds on both clusters as shown below:

	3 Node Cluster	**5 Node Cluster**
Total time spent on loading edges	66 sec	48 sec
Setting initial distances	0.08 sec	0.07 sec
Performing the Dijkstra algorithm	45 sec	38 sec

Table 2: 3 node vs 5 node clusters-Spark-Graphx comparative performance

We can also note that spark-Graphx is a bit more performant on 5 node cluster compared to a 3-node cluster.

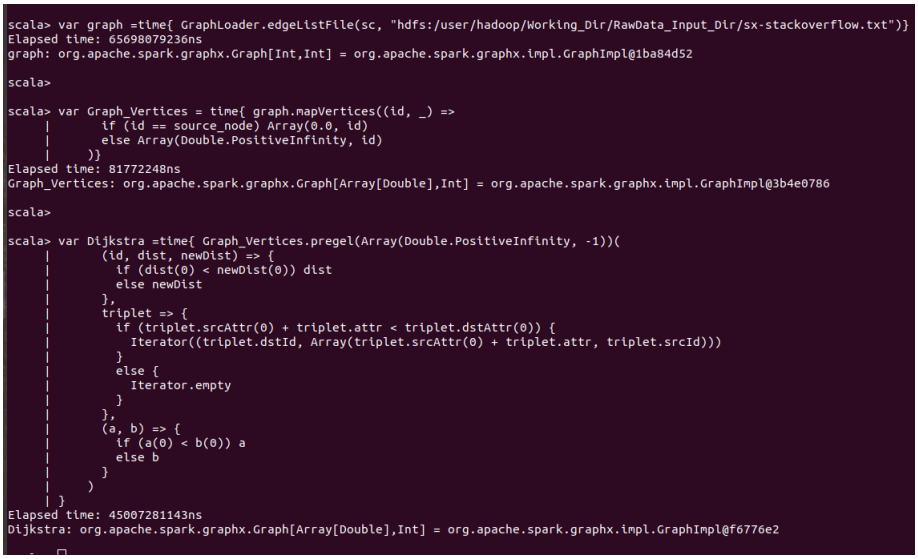


Figure 15: Scalability-Spark on 3 Nodes cluster

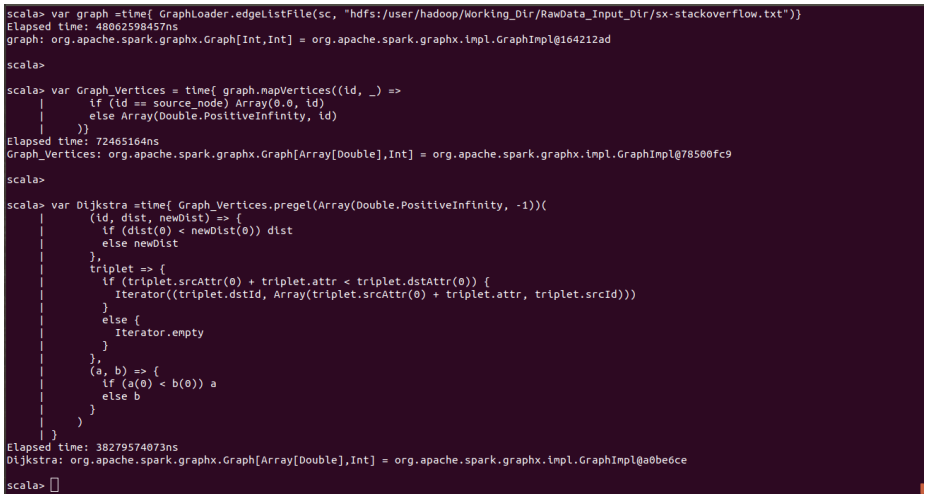


Figure 16: Scalability-Spark on 5 Nodes cluster

5. Conclusion

After running a big graph on tow clusters with different size we have noted that MapReduce streaming is burdened especially during job chaining. This is mainly due in the case of our algorithm to the inter-job tasks including

- Verification of convergence criteria through the overall distance summation (a complexity of o(n) as all nodes have to be checked)
- Deleting the content of the input directory
- Transferring output files to input directory
- Deleting the output directory

However, it’s clear that the size of the used cluster has a huge impact on the processing time which was noted in our scalability test. Furthermore, we have noted spark-Graphx power which outperformed the developed MapReduce job This of course is due to the amount of optimization provided by such package

6. Problems and challenges during the project

Challenge 1: All graph datasets found are in the form of edge list and only network graph contains weights in the form of time stamp.

Solution: We developed a MapReduce job to convert the edge list into required format for our mapper

Challenge 2: Trying to connect to the pre-available clusters on Rosetta we faced an error regarding the availability of the Shuffle_Sort service

Solution: After various google search the only way to solve it was to modify the config file of Yarn which we had not access to. In order to bypass this, we created clusters from scratch using the AWS platform

Challenge 3: Trying to connect to the created cluster, we have faced the “Timed out connection” Error

Solution: In order to solve this, we had to modify the inbound rule in the master node security group to allow SSH connection from port 22 and from any IP address or our specific IP

Challenge 4: Trying to use spark shell, we have found the “SC not found error”.

Solution: In order to resolve this, we used the Sudo spark-shell instead to get the relevant privileges. Its also worth mentioning that it would be better to avoid the Sudo as you want be able to write output data to HDFS due to issue of root privileges

Challenge 5: We have tried to download the relevant output files from HDFS

Solution: The best solution is to transfer all files on the HDFS to the S3 bucket however we had not the privellages to generate the Access key and the Secret access key

7. Reference

Jeffrey Dean and Sanjay Ghemawat (2004). *MapReduce: Simplified Data Processing on Large Clusters, Google*

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung (2003), The Google File System, Google

Bellman, Richard (1958). "On a routing problem". Quarterly of Applied Mathematics. 16: 87–90.

Dijkstra, E. W. (1959). *A note on two problems in connexion with graphs*. Numerische Mathematik. 1: 269–271.

Fredman, Michael Lawrence; Tarjan, Robert E.(1984). *Fibonacci heaps and their uses in improved network optimization algorithms*. 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338–346.

Thorup, Mikkel (1999). *Undirected single-source shortest paths with positive integer weights in linear time*". Journal of the ACM. 46 (3): 362–394.

8. Appendix A: Code to replicate the MapReduce chaining on both datasets

```
#Step1: download relevant codes
        wget "https://www.dropbox.com/s/q3xlngls3ysl3ad/Prep_Mapper.py"
        wget "https://www.dropbox.com/s/1cksusjtq9qx8jx/Prep_Reducer.py"
        wget "https://www.dropbox.com/s/8fcnyutsbd1h6np/mapper.py"
        wget "https://www.dropbox.com/s/u19tud5jpetyzkn/reducer.py"
        wget "https://www.dropbox.com/s/egogsrgt44tjkmz/Iterative_Dijikstra_Hadoop.sh"
#Step 2: provide relevant access permission
        chmod +x *.py
        chmod +x *.sh
#Step 3: Create the relevant directory structure
        hdfs dfs -mkdir /user/hadoop/Working_Dir
        hdfs dfs -mkdir /user/hadoop/Working_Dir/RawData_Input_Dir
#Step 4: Download Raw data form ( NodeID, NeighborID, Weight) from SNAP.
        #Dataset 1: email-Eu-core temporal network (Nodes        986 , Temporal Edges 332,334)
        wget "https://snap.stanford.edu/data/email-Eu-core-temporal.txt.gz"
        gunzip email-Eu-core-temporal.txt.gz
        hdfs dfs -put  email-Eu-core-temporal.txt /user/hadoop/Working_Dir/RawData_Input_Dir
        #Dataset 2: Stack Overflow temporal network (Nodes 2,601,977, Temporal Edges 63,497,050)
        wget "https://snap.stanford.edu/data/sx-stackoverflow.txt.gz"
        gunzip sx-stackoverflow.txt.gz
        hdfs dfs -put  sx-stackoverflow.txt /user/hadoop/Working_Dir/RawData_Input_Dir
#Note that for each data you will need to delete the Working_Dir then follow step 3 throughout 7
        hdfs dfs -rm -r -f  /user/hadoop/Working_Dir
#Step 5: Run the data cleaning and reformatting job: Note this will create the input directory for the next Dijkstra stream job
        time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
        -input /user/hadoop/Working_Dir/RawData_Input_Dir \
        -output /user/hadoop/Working_Dir/Input_Dir \
        -file /home/hadoop/Prep_Mapper.py \
        -mapper /home/hadoop/Prep_Mapper.py \
        -file /home/hadoop/Prep_Reducer.py \
        -reducer /home/hadoop/Prep_Reducer.py
#Step 7: Run the iterative Dijkstra
        Input_Dir=/user/hadoop/Working_Dir/Input_Dir
        Output_Dir=/user/hadoop/Working_Dir/Output_Dir
        Mapper=/home/hadoop/mapper.py
        Reducer=/home/hadoop/reducer.py
        time ./Iterative_Dijikstra_Hadoop.sh $Input_Dir $Mapper $Reducer $Output_Dir
```

9. Appendix B: Code to replicate the Spark-Graphx experiment on both datasets

```
#Step 1: Create the relevant directory structure
        hdfs dfs -mkdir /user/hadoop/Working_Dir
        hdfs dfs -mkdir /user/hadoop/Working_Dir/RawData_Input_Dir
#Step 2: Download Raw data form ( NodeID, NeighborID, Weight) from SNAP.
        #Dataset 1: email-Eu-core temporal network (Nodes       986 , Temporal Edges 332,334)
        wget "https://snap.stanford.edu/data/email-Eu-core-temporal.txt.gz"
        gunzip email-Eu-core-temporal.txt.gz
        hdfs dfs -put  email-Eu-core-temporal.txt /user/hadoop/Working_Dir/RawData_Input_Dir
        #Dataset 2: Stack Overflow temporal network (Nodes 2,601,977, Temporal Edges 63,497,050)
        wget "https://snap.stanford.edu/data/sx-stackoverflow.txt.gz"
        gunzip sx-stackoverflow.txt.gz
        hdfs dfs -put  sx-stackoverflow.txt /user/hadoop/Working_Dir/RawData_Input_Dir
#Step 3: Run spark
        spark-shell
#Copy and paste the below code, make sure to change the name of the input file to the relevant one and
change the output directory after first run, note for the stack overflow the starting node ID is 1 not zero
        import org.apache.spark._
        import org.apache.spark.graphx._
        import org.apache.spark.rdd.RDD
        val source_node: VertexId = 0
        val graph = GraphLoader.edgeListFile(sc, "hdfs:/user/hadoop/Working_Dir/RawData_Input_Dir/email-
Eu-core-temporal.txt")
        val Graph_Vertices = graph.mapVertices((id, _) =>
                if (id == source_node) Array(0.0, id)
                else Array(Double.PositiveInfinity, id)
               )
        val Dijkstra = Graph_Vertices.pregel(Array(Double.PositiveInfinity, -1))(
                (id, dist, newDist) => {
                        if (dist(0) < newDist(0)) dist
                        else newDist
                },
                triplet => {
                        if (triplet.srcAttr(0) + triplet.attr < triplet.dstAttr(0)) {
                          Iterator((triplet.dstId, Array(triplet.srcAttr(0) + triplet.attr, triplet.srcId)))
```

```
            }
            else {
              Iterator.empty
            }
        },
        (a, b) => {
            if (a(0) < b(0)) a
            else b
        }
    )

val Dijkstra_output: RDD[String] = Dijkstra.vertices.map(vertex => vertex._1 + "    " + vertex._2(0) + "
" + vertex._2(1).toInt)

Dijkstra_output.collect.foreach(println(_))

Dijkstra_output.saveAsTextFile("hdfs:/user/hadoop/Working_Dir/output")
```

10. Appendix C: Dijkstra chaining time stats on 3 Node Cluster

3 Node Cluster stats						
Iteration number	**Iteration start time**	**Iteration end time**	**Elapsed time per iteration**	**Inter-job elapsed time**	**Average time per map task**	**Average time per reduce task**
87	12:55:03 AM	12:59:01 AM	03:58	-	01:24	01:21
88	1:00:01 AM	1:03:49 AM	03:48	01:00	01:19	01:17
89	1:04:47 AM	1:08:36 AM	03:49	00:58	01:23	01:15
90	1:09:35 AM	1:13:31 AM	03:56	00:59	01:23	01:22
91	1:14:30 AM	1:18:46 AM	04:16	00:59	01:25	01:34
92	1:19:45 AM	1:23:37 AM	03:52	00:59	01:20	01:20
93	1:24:36 AM	1:28:36 AM	04:00	00:59	01:21	01:25
94	1:29:36 AM	1:33:39 AM	04:03	01:00	01:24	01:27
95	1:34:39 AM	1:38:39 AM	04:00	01:00	01:25	01:23
96	1:39:38 AM	1:43:32 AM	03:54	00:59	01:23	01:17
97	1:44:31 AM	1:48:38 AM	04:07	00:59	01:25	01:27
98	1:49:37 AM	1:53:39 AM	04:02	00:59	01:21	01:27
99	1:54:38 AM	1:58:40 AM	04:02	00:59	01:21	01:26
100	1:59:41 AM	2:03:37 AM	03:56	01:01	01:22	01:22
101	2:04:36 AM	2:08:42 AM	04:06	00:59	01:25	01:27
102	2:09:43 AM	2:13:44 AM	04:01	01:01	01:22	01:26
103	2:14:44 AM	2:19:06 AM	04:22	01:00	01:28	01:39
104	2:20:09 AM	2:24:46 AM	04:37	01:03	01:32	01:48
105	2:25:46 AM	2:29:54 AM	04:08	01:00	01:28	01:25
106	2:30:54 AM	2:35:17 AM	04:23	01:00	01:27	01:40
107	2:36:17 AM	2:40:20 AM	04:03	01:00	01:23	01:24
108	2:41:20 AM	2:45:33 AM	04:13	01:00	01:28	01:28
109	2:46:34 AM	2:50:47 AM	04:13	01:01	01:29	01:26
110	2:51:47 AM	2:55:52 AM	04:05	01:00	01:26	01:25
111	2:56:53 AM	3:01:07 AM	04:14	01:01	01:27	01:33
112	3:02:09 AM	3:06:29 AM	04:20	01:02	01:32	01:29
113	3:07:30 AM	3:11:44 AM	04:14	01:01	01:30	01:27
114	3:12:45 AM	3:17:05 AM	04:20	01:01	01:26	01:36
115	3:18:06 AM	3:22:29 AM	04:23	01:01	01:30	01:34
116	3:23:29 AM	3:28:00 AM	04:31	01:00	01:31	01:43
117	3:29:00 AM	3:33:07 AM	04:07	01:00	01:25	01:36
118	3:34:08 AM	3:38:33 AM	04:25	01:01	01:36	01:34
119	3:39:34 AM	3:43:50 AM	04:16	01:01	01:31	01:30
120	3:44:50 AM	3:49:08 AM	04:18	01:00	01:28	01:33
121	3:50:09 AM	3:54:38 AM	04:29	01:01	01:36	01:34
122	3:55:39 AM	4:00:18 AM	04:39	01:01	01:31	01:53
123	4:01:18 AM	4:05:47 AM	04:29	01:00	01:34	01:32
124	4:06:48 AM	4:11:15 AM	04:27	01:01	01:35	01:35
125	4:12:16 AM	4:16:39 AM	04:23	01:01	01:32	01:36
126	4:17:41 AM	4:21:59 AM	04:18	01:02	01:30	01:29
127	4:23:01 AM	4:27:34 AM	04:33	01:02	01:31	01:42
128	4:28:35 AM	4:32:59 AM	04:24	01:01	01:37	01:42

129	4:34:19 AM	4:38:55 AM	04:36	01:20	01:32	01:46
130	4:39:56 AM	4:44:22 AM	04:26	01:01	01:30	01:37
131	4:45:23 AM	4:49:53 AM	04:30	01:01	01:37	01:35
132	4:50:55 AM	4:55:26 AM	04:31	01:02	01:37	01:36
133	4:56:28 AM	5:00:55 AM	04:27	01:02	01:30	01:36
134	5:01:55 AM	5:06:24 AM	04:29	01:00	01:37	01:33

11. Appendix D: Dijkstra chaining time stats on 5 Node Cluster

5 Node Cluster stats						
Iteration number	**Iteration start time**	**Iteration end time**	**Elapsed time per iteration**	**Interjob elapsed time**	**Average time per map task**	**Average time per reduce task**
125	02:24:54	02:27:36	02:42	-	01:12	01:01
126	02:28:37	02:31:12	02:35	01:01	01:14	00:56
127	02:32:17	02:34:46	02:29	01:05	01:15	00:45
128	02:35:46	02:38:21	02:35	01:00	01:16	00:51
129	02:39:23	02:41:56	02:33	01:02	01:16	00:49
130	02:42:57	02:45:29	02:32	01:01	01:15	00:48
131	02:46:30	02:49:06	02:36	01:01	01:18	00:51
132	02:49:06	02:52:58	03:52	00:00	01:17	01:06
133	02:54:00	02:56:38	02:38	01:02	01:13	00:56
134	02:57:39	03:00:22	02:43	01:01	01:13	00:59
135	03:01:23	03:03:58	02:35	01:01	01:14	00:53
136	03:05:00	03:07:34	02:34	01:02	01:16	00:49
137	03:08:36	03:11:07	02:31	01:02	01:15	00:50
138	03:12:08	03:14:42	02:34	01:01	01:17	00:50
139	03:15:44	03:18:20	02:36	01:02	01:19	00:50
140	03:19:21	03:21:54	02:33	01:01	01:18	00:48
141	03:22:57	03:25:36	02:39	01:03	01:19	00:53
142	03:26:38	03:29:19	02:41	01:02	01:16	00:56
143	03:30:20	03:33:04	02:44	01:01	01:19	00:55
144	03:34:06	03:36:58	02:52	01:02	01:18	01:03
145	03:38:00	03:40:39	02:39	01:02	01:20	00:51
146	03:41:42	03:44:18	02:36	01:03	01:18	00:49
147	03:45:19	03:47:57	02:38	01:01	01:20	00:48
148	03:48:59	03:51:35	02:36	01:02	01:21	00:50
149	03:52:41	03:55:16	02:35	01:06	01:19	00:49
150	03:56:17	03:58:53	02:36	01:01	01:16	00:54
151	03:59:55	04:02:33	02:38	01:02	01:20	00:51
152	04:03:35	04:06:18	02:43	01:02	01:21	00:55
153	04:07:23	04:10:26	03:03	01:05	01:19	01:15
154	04:11:28	04:14:20	02:52	01:02	01:17	01:05
155	04:15:22	04:18:16	02:54	01:02	01:23	01:03
156	04:19:19	04:22:07	02:48	01:03	01:23	00:55
157	04:23:09	04:25:53	02:44	01:02	01:21	00:55
158	04:26:55	04:29:32	02:37	01:02	01:20	00:48
159	04:30:33	04:33:17	02:44	01:01	01:19	00:56
160	04:34:20	04:37:12	02:52	01:03	01:21	01:05
161	04:38:15	04:40:58	02:43	01:03	01:21	00:53
162	04:42:00	04:44:48	02:48	01:02	01:22	00:59

163	04:45:50	04:48:46	02:56	01:02	01:20	01:03
164	04:49:49	04:52:32	02:43	01:03	01:24	00:52
165	04:53:34	04:56:19	02:45	01:02	01:19	00:58
166	04:57:21	05:00:16	02:55	01:02	01:21	01:05
167	05:01:18	05:04:02	02:44	01:02	01:22	00:53
168	05:05:05	05:08:01	02:56	01:03	01:18	01:09