

UNIVERSITE PARIS DAUPHINE

BIG DATA

Single-source shortest path - Dijkstra Algorithm

Students

Elie ABI HANNA DAHER
Bilal EL CHAMI
Badr ERRAJI

Professor

Mr Dario COLAZZO

January 30, 2018



Contents

1	Project goal	2
2	Dijkstra Algorithm	2
3	Implementation	3
3.1	Input	3
3.1.1	Data	3
3.1.2	Prepare	4
3.2	Mapper	5
3.3	Reducer	6
3.4	Job Chaining	7
4	Code description	8
4.1	Hadoop	8
4.2	Spark	8
5	Results	9
5.1	Hadoop	9
5.2	Spark	10
6	Scalability experiments	10
6.1	Data	10
6.2	Environment	10
6.3	Hadoop	10
6.4	Spark	11
A	Hadoop - Mapper python code	13
B	Hadoop - Reducer python code	14
C	Spark code	15
D	Prepare python code	17
E	Add distance to graph - python code	18

1 Project goal

The goal of the project is to find the shortest paths from a source node to all other nodes in the graph using the Dijkstra's algorithm. The algorithm should be implemented in both Python-Hadoop and Spark.

Alongside the implementation, a scalability experiment is needed to check the performance of the algorithm implemented.

You can find source code, data samples, documentation, etc in our repository : github.com/bilal-elchami/bigdata .

2 Dijkstra Algorithm

The Dijkstra's algorithm finds the shortest path from source to all other nodes. The Dijkstra algorithm is very similar to the BFS algorithm, the only difference is that the distance between neighbors isn't 1, distance can differ from a neighbor to another.

One of the most common and well-studied problems in graph theory is the single-source shortest path problem or Dijkstra's algorithm, where the task is to find shortest paths from a source node to all other nodes in the graph (or alternatively, edges can be associated with costs or weights, in which case the task is to compute lowest-cost or lowest-weight paths). This algorithm is very similar to the Breadth-first search algorithm, the only difference is that the distance between neighbors isn't equal to 1 but it can differ from a neighbor to another.

However, this algorithm assumes sequential processing. So, the challenge was to solve this problem in parallel, and more specifically, with a MapReduce job. This section will be discussed later in the Mapper/Reducer sections of this document.

3 Implementation

3.1 Input

3.1.1 Data

The map task should receive the following information

- node
- distance
- neighbors data that contains the list of neighbors with their respective distance to the node
- path

So let's take the following example with 1 being the start node :

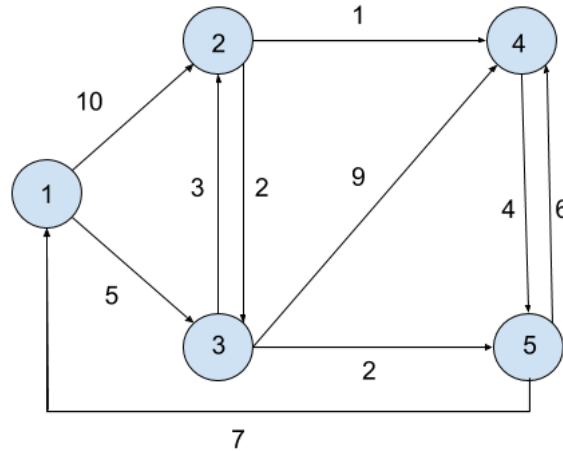


Figure 1: Example of a graph

For the first iteration, the path will be empty. So input data will look like this :

- 1 0 2,10:3,5:
- 2 999 3,2:4,1:
- 3 999 2,3:4,9:5,2:
- 4 999 5,4:
- 5 999 1,7:4,6:

So as you can see, the start node has a distance of 0, and all other node have a distance of 999 which represent an infinite number. The neighbor list contains each neighbor node with their respective distance, the neighbors and the distance are separated by a "," and neighbors are separated by ":".

3.1.2 Prepare

Technically, the format we set for the data is very hard to implement to a large graph. Usually the graph is represented by the distance between nodes. So for the graph provided in the previous page, the input data will look like this

- 1 2 10
- 1 3 5
- 2 3 2
- 2 4 1
- 3 2 3
- 3 4 9
- 3 5 2
- 4 5 4
- 5 1 7
- 5 4 6

We created a job MapReduce - called prepare that will format the usual format of a graph to the format we are asking for.

The prepare algorithm will take in consideration the nodes that dont connect to another node but is a neighbor for another node.

3.2 Mapper

A map task receive (K,V)

- Key : node
- Value : distance, neighbors data, path

In the first iteration, the path will be empty.

The map task will :

1. emit the node with his information (distance, neighbors data and path)
2. \forall neighbor \in neighbors, it will emit
 - Key : neighbor
 - Value : (node distance + distance of node to the neighbor , node path + neighbor).

The pseudo code of the mapper is as follow :

Algorithm 1 Mapper

```
1: procedure MAP (node, (distance, neighbors, path))
2:   Emit (node, (distance, neighbors, path))
3:   for all neighbor  $\in$  neighbors do
4:     dist  $\leftarrow$  distance + neighbor.distance
5:     p  $\leftarrow$  path + neighbor.id
6:     Emit (neighbor.id, (dist, p))
```

3.3 Reducer

The reducer will gather the possible distance for each node and selects the minimum one and set the path of the minimum one selected.

The reducer task will iterate through the nodes generated from the mapper task and for each node it will :

1. identify if the node contain the data structure or just a distance
2. identify the minimum distance of the node
3. Emit the minimum distance and the graph structure with the path

The pseudo code of the reducer is as follow :

Algorithm 2 Reducer

```
1: procedure REDUCER (node, [(distance, neighbors, path), (distance,  
   path),...])  
2:    $d_{min} \leftarrow \infty$   
3:   nodePath  $\leftarrow \emptyset$   
4:   graph  $\leftarrow \emptyset$   
5:  
6:   for  $v \in values$   
7:  
8:   if isNode(v) then  
9:      $d_{min} \leftarrow v.distance$   
10:    nodePath  $\leftarrow v.path$   
11:    graph  $\leftarrow v.neighbors$   
12:  else  
13:    if  $d_{min} \leq v.distance$  then  
14:       $d_{min} \leftarrow v.distance$   
15:      nodePath  $\leftarrow v.path$   
16:  Emit (node, ( $d_{min}$ , graph, nodePath))
```

3.4 Job Chaining

Since Dijkstra algorithm is iterative and we were using python streaming in for the project, we needed to execute several MapReduce steps with overall job scenarios, which means the last reduce output will be used as input for the next map job.

Map1 \rightarrow Reduce1 \rightarrow Map2 \rightarrow Reduce2 \rightarrow Map3...

So the simplest way to achieve it was by creating a shell script, that automates what we were doing manually.

The script launches the Hadoop streaming job, merges the output parts into one file and then moves it into the data directory so it can be used as input for the next job.

It's not the optimal solution but it can do the job in an "acceptable" amount of time.

In spark, we didn't face any problem with the iteration, since we can do it with a simple while or for loop.

The Job Chain script is available on another github repository :
[https : //github.com/bilal – elchami/hadoop_streaming_job_chaining](https://github.com/bilal-elchami/hadoop_streaming_job_chaining).

4 Code description

Source code can be found in the appendices or in the github repository.

4.1 Hadoop

The MapReduce python code was a straight forward implementation of the pseudo code elaborated above. We didn't find any difficulties implementing the algorithm.

4.2 Spark

The conception of the script has been adapted according to the Hadoop streaming job.

The loop breaking condition is identified with the help of an accumulator that breaks the loop when the distances at every node no longer change at the next frontier.

```
def minDistance(nodeValue1, nodeValue2):
    neighbors = None
    distance = 0
    path = ''
    if nodeValue1[1] != 'None':
        neighbors = nodeValue1[1]
    else:
        neighbors = nodeValue2[1]
    dist1 = nodeValue1[0]
    dist2 = nodeValue2[0]
    if dist1 <= dist2:
        distance = dist1
        path = nodeValue1[2]
    else:
        count.add(1);
        distance = dist2
        path = nodeValue2[2]
    return (distance, neighbors, path)
```

Figure 2: Incrementation of the accumulator in min distance function

The accumulator was used as a flag representing if the distance of a node was changed in the “minDistance(nodeValue1, nodeValue2)” function as you can see in the above code

```

oldCount = 0
iterations = 0
while True:
    iterations += 1
    nodesValues = nodes.map(lambda x: x[1])
    neighbors =
        nodesValues.filter(lambda nodeDataFilter: nodeDataFilter[1]!=None).map(
            lambda nodeData: map(
                lambda neighbor: customSplitNeighbor(
                    nodeData[2], nodeData[0], neighbor
                ), nodeData[1]
            )
        ).flatMap(lambda x: x)
    mapper = nodes.union(neighbors)
    reducer = mapper.reduceByKey(lambda x, y: minDistance(x, y))
    nodes = reducer.map(lambda node: customSplitNodesIterative(node))
    nodes.count() # We call the count to execute all the RDD transformations
    if oldCount == count.value:
        break
    oldCount=count.value

```

Figure 3: Breaking condition to terminate the algorithm

5 Results

You can check the execution commands in the README file in the repository.

5.1 Hadoop

When we pass the above graph as input to the Hadoop streaming MapReduce, the output file will contain the following data

```

1 0 2,10:3,5: 1
2 8 3,2:4,1: 1 → 3 → 2
3 5 2,3:4,9:5,2: 1 → 3
4 9 5,4: 1 → 3 → 2 → 4
5 7 1,7:4,6: 1 → 3 → 5

```

And each element in each line represents:

- the current's node id
- the calculated distance from the “Start Node” to the “Current Node”
- the neighbors of the “Current Node” with its distance
- the shortest path found from the “Start Node” to the “Current Node”

5.2 Spark

The following output represents the spark calculation results. It is the same as the Hadoop output but without the unnecessary neighbors data.

```
>>> result.collect()
[(u'1', 0, u'1'), (u'3', 5, u'1->3'), (u'2', 8, u'1->3->2'), (u'5', 7, u'1->3->5'),
 (u'4', 9, u'1->3->2->4'), (u'6', 12, u'1->3->5->6')]
>>>
```

Figure 4: Example of a graph

6 Scalability experiments

6.1 Data

We used the Facebook data, that consists of friends list from Facebook. Facebook data has been anonymized by replacing the id of the facebook of each user with a new value.

The graph contains 4039 nodes and 88234 edges.

The facebook data was downloaded from the Stanford's website :
<https://snap.stanford.edu/data/egonets-Facebook.html>.

We created a Map job that adds a random distance (between 1 and 10) for each intersection. The created job is a python script, and can be found under the following folder `python/add-distance-graph.py` in the repository.

After adding random distances, another map job was executed was created to format the data and make it compatible with our MapReduce input (check section 3.1.2).

6.2 Environment

We did the experiment on the Dauphine Cluster by using 10 cores for Spark, and 10 reducers for Hadoop Python Streaming.

6.3 Hadoop

The facebook data were executed 100 times on Hadoop Streaming with help of the Job Chain Script. Each iteration took about 7.281 seconds. The time is acceptable for the following data, but will eventually increase for bigger graph.

6.4 Spark

In Spark, the iteration took much more longer than hadoop. We faced some scalability issues with bigger graph structures. Those issues can be spotted in the formatting data section which was implemented from a more readable structure. We identified the code that was causing the latency in the execution :

```
oldCount = 0
iterations = 0
while True:
    iterations += 1
    nodesValues = nodes.map(lambda x: x[1])
    neighbors =
        nodesValues.filter(lambda nodeDataFilter: nodeDataFilter[1] != None).map(
            lambda nodeData: map(
                lambda neighbor: customSplitNeighbor(
                    nodeData[2], nodeData[0], neighbor
                ), nodeData[1]
            )
        ).flatMap(lambda x: x)
    mapper = nodes.union(neighbors)
    reducer = mapper.reduceByKey(lambda x, y: minDistance(x, y))
    nodes = reducer.map(lambda node: customSplitNodesIterative(node))
    nodes.count() # We call the count to execute all the RDD transformations
    if oldCount == count.value:
        break
    oldCount = count.value
```

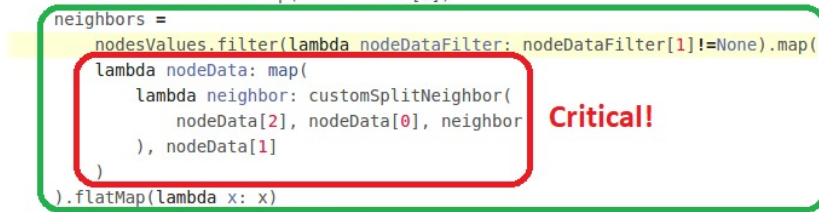


Figure 5: Critical code that can be optimized

Unfortunately, it's not the most optimized way to implement the Dijkstra's algorithm.

References

- [1] *Cloud Computing Lecture 4 - Graph Algorithms with MapReduce*. Jimmy Lin, The iSchool, University of Maryland, February 6, 2008.
- [2] *Hadoop code from github*. <https://github.com/Troll-Mcloving/SistemasDistribuidosAvroHadoop>
- [3] *Dijkstra Algorithm - wikipedia page*

A Hadoop - Mapper python code

```
#!/usr/bin/env python3
import sys

for line in sys.stdin:
    lineValues = line.strip().split(' ')
    nid = lineValues[0]
    distance = int(lineValues[1])
    neighbors = 'no-neighbors'
    if len(lineValues) > 2:
        neighbors = lineValues[2]
    path = nid
    if len(lineValues) > 3:
        path = lineValues[3]
        elements = path.split('->')
        if elements[len(elements) - 1] != nid:
            path = lineValues[3] + '->' + nid
    print(nid + ' NODE ' + str(distance) + ' ' + neighbors + ' ' + str(path));

    if neighbors != 'no neighbors':
        adjacencyList = neighbors.split(':')
        for i in range(len(adjacencyList) - 1):
            neighborData = adjacencyList[i].split(',')
            neighbor = neighborData[0]
            currentPath = path + '->' + neighbor
            neighborDistance = distance + int(neighborData[1])
            print(neighbor + ' VALUE ' + str(neighborDistance) + ' ' + currentPath)
```

B Hadoop - Reducer python code

```
#!/usr/bin/env python3
import sys

minDistance = 9999;
currentMinNode = None;
neighbors = None;
currentNode = None;

def emit(path):
    print(str(currentNode) + ' ' + str(minDistance) + ' ' + neighbors + ' ' + path);

for line in sys.stdin:
    lineValues = line.strip().split(' ');
    nid = int(lineValues[0]);
    type = lineValues[1];
    distance = int(lineValues[2]);

    if type == 'NODE':
        if currentNode != None:
            if currentMinNode == None:
                currentMinNode = nid;
            emit(path);
            path = lineValues[4];
            neighbors = lineValues[3];
            currentNode = nid;
            minDistance = distance;
            currentMinNode = nid;
        else:
            if distance < minDistance:
                minDistance = distance;
                currentMinNode = nid;
                path = lineValues[3];

emit(path);
```

C Spark code

```
textFile = sc.textFile("data/input.dat")

count = sc.accumulator(0)

def customSplitNodesTextFile(node):
    if len(node.split(' ')) < 3:
        nid, distance = node.split(' ')
        neighbors = None
    else:
        nid, distance, neighbors = node.split(' ')
        neighbors = neighbors.split(':')
        neighbors = neighbors[:len(neighbors) - 1]
    path = nid
    return (nid, (int(distance), neighbors, path))

def customSplitNodesIterative(node):
    nid = node[0]
    distance = node[1][0]
    neighbors = node[1][1]
    path = node[1][2]
    elements = path.split('->')
    if elements[len(elements) - 1] != nid:
        path = path + '->' + nid;
    return (nid, (int(distance), neighbors, path))

def customSplitNeighbor(parentPath, parentDistance, neighbor):
    if neighbor != None:
        nid, distance = neighbor.split(',')
        distance = parentDistance + int(distance)
        path = parentPath + '->' + nid
        return (nid, (int(distance), 'None', path))

def minDistance(nodeValue1, nodeValue2):
    neighbors = None
    distance = 0
    path = ''
    if nodeValue1[1] != 'None':
        neighbors = nodeValue1[1]
    else:
        neighbors = nodeValue2[1]
    dist1 = nodeValue1[0]
    dist2 = nodeValue2[0]
    if dist1 <= dist2:
        distance = dist1
        path = nodeValue1[2]
    else:
        count.add(1);
        distance = dist2
        path = nodeValue2[2]
    return (distance, neighbors, path)

def formatResult(node):
    nid = node[0]
    minDistance = node[1][0]
    path = node[1][2]
    return nid, minDistance, path
```



```

nodes = textFile.map(lambda node: customSplitNodesTextFile(node))

oldCount = 0
iterations = 0
while True:
    iterations += 1
    nodesValues = nodes.map(lambda x: x[1])
    neighbors = nodesValues.filter(lambda nodeDataFilter: nodeDataFilter[1] != None).map(
        lambda nodeData: map(
            lambda neighbor: customSplitNeighbor(
                nodeData[2], nodeData[0], neighbor
            ), nodeData[1]
        )
    ).flatMap(lambda x: x)
    mapper = nodes.union(neighbors)
    reducer = mapper.reduceByKey(lambda x, y: minDistance(x, y))
    nodes = reducer.map(lambda node: customSplitNodesIterative(node))
    nodes.count() # We call the count to execute all the RDD transformations
    if oldCount == count.value:
        break
    oldCount = count.value

print('Finished after: ' + str(iterations) + ' iterations')
result = reducer.map(lambda node: formatResult(node))

```

D Prepare python code

```
#!/usr/bin/env python3

import sys

infini = 999
pointsTo = ''
currentNode = None
startNode = sys.argv[1]
nodes = []
neighbors = []

def emit():
    dist = infini
    if currentNode == startNode:
        dist = 0
    nodes.append(currentNode)
    print(currentNode + ' ' + str(dist) + ' ' + pointsTo)

def addNeighbor(neighbor):
    added = False
    for i in range(len(neighbors)):
        if neighbor == neighbors[i]:
            added = True
    if added == False:
        neighbors.append(neighbor)

for line in sys.stdin:
    lineValues = line.strip().split(' ')
    nid = lineValues[0]
    neighbor = lineValues[1]
    distance = int(lineValues[2])
    if currentNode != None:
        if currentNode != nid:
            emit()
            pointsTo = ''
            currentNode = nid
        else:
            currentNode = nid
            addNeighbor(neighbor)
            pointsTo = pointsTo + neighbor + ',' + str(distance) + ':'

emit()

for i in range(len(neighbors)):
    found = False
    for x in range(len(nodes)):
        if nodes[x] == neighbors[i]:
            found = True
    if found == False:
        nodes.append(neighbors[i])
        dist = infini
        if neighbors[i] == startNode:
            dist = 0
        print(neighbors[i] + ' ' + str(dist) + ' ' + pointsTo)
```

E Add distance to graph - python code

```
#!/usr/bin/env python3

from random import randint
import sys

minDistance = 1
maxDistance = 10

for line in sys.stdin:
    lineValues = line.strip().split(' ')
    nid1 = lineValues[0]
    nid2 = lineValues[1]
    randomDistance = randint(minDistance, maxDistance)
    print(nid1 + ' ' + nid2 + ' ' + str(randomDistance))
```