Work 2 Report

João Pacheco
56982
Faculdade de Ciências e Tecnologia

***Abstract*—This document will provide an analysis of a Big Data HCP problem utilizing Apache Spark.**
***Index Terms*—Spark, Big Data, HCP**

I. Introduction

The partnership between Big Data and High Computation Programming problems has been increasing throughout the years. In this work we transformed our data of Flights into a graph (airports would be represented by the nodes and the edges would be the flights while the weight of each edge would be the time of each flight) in order to apply Dijkstra's SPF algorithm and find the quickest route between 2 airports.

II. Goal

The Goal of this work, was to implement a version of Dijkstra's Shortest Path First algorithm and apply it in a big dataset to compare the performance of both the Spark version and the sequential version varrying the amount of Nodes in the Spark Cluster.

III. Program Functionality

When running the program, for both the Sequential and Spark version, to check the shortest path between NodeX and NodeY you need to run the command: `info NodeX and NodeY`.

The Spark Version allows the user to redefine an edge of the matrix by running `time NodeX NodeY Value`. This will be remembered throughout the execution of the program.

If the user inputs as a program argument "**1**", the program will use the Flights.csv dataset instead of generating one.

Note: To make sure the Spark Context is closed, the user must type **quit**.

IV. Implementation

My implementation follows along what was proposed by the teacher in order to make the algorithm function in Spark. The algorithm presented was Primm's so I had to modify it accordingly.

This implementation has 3 big steps: **Construction of the RDD l**, **Construction of the Adjacency Matrix** and **Recalculate l iteratively utilizing the algorithm**.

A. Construction of RDD l

To do this, I mapped every FlightRDD into a pair composed of $<$*Index*,$<$*Value*, *Visited*, *Predecessor*$>>$.

- **Index:** The Id of the Airport
- **Value:** The Value/Weight to go to that airport. Since this is the beginning of the process, utilizing the *temp* l created before this process which contains the initial values of l, it'll assign the values of it according to the existing of a direct edge from this airport and the source airport, otherwise will be ∞
- **Visited:** This value will be utilized later in the process. Right now all are set to $false$ except for the source.
- **Predecessor:** This field will be useful to finding the complete route at the end. Right now, the values with direct edges to the source will have it's ID in this field, the rest will have -1.

B. Building The Matrix

For this step, at the class initiation it receives the RDD of the Flights. With this, it creates a new RDD of Matrix Entries where the coordinates represent 2 airports and it's value represents the average time of the flights.

If the user did any edge modifications utilizing the $time$ command, it'll be created a new RDD of Matrix Entries containing the defined user entries and it'll do an **union** with both this RDD and the one created in the step before.

Using this RDD we can create a Coordinate Matrix and transpose it after. I did this way, it could have been built already transposed.

With this new Coordinate Matrix it was advised to transform it into an IndexedRowMatrix as each row are represented by its index (long-typed) and a local vector". This means in my project, the index represents the airport and the vector would contain 2 lists: first list represents the indexes and the second list the values.

Since I would later require to do a **join** between this Matrix and l I created an RDD pair which the first parameter was the airport index and the second parameter was the vector that contained the column. I made this RDD persist in memory as it's immutable (RDD's are immutable by nature, what I'm saying is that it won't be replaced) as it is required each time the **join** occurred.

C. Applying the Algorithm

The algorithm is in a cycle which will only stop when all the members of l were visited.

Applying Dijkstra's algorithm, first step was to discover the minimum. To do this, I call a method which takes l and the **source** ID. It starts by filtering l by all the nodes that haven't been visited yet. Then it maps it into a pair of $<$*Value*, *Index*$>$so it can run a **sortByKey()** on the value and sort the RDD putting the minimum first. Since the RDD is sorted with the minimum first, it returns the first one.

Then, there's a join between l and the Matrix RDD cached in the step before.

Now having the **join of** l **AND the Matrix** and the minimum's index and value, we calculate the new l. For this I call a method which takes the 3 mentioned and the toVisit list to update the booleans on the RDD.

This method returns a new updated l by mapping the joined previous l and the Matrix to a new pair which has the structure of l, calculating the new values of each position by checking with the associated Vector if the **sum** of the **minimum's value** and the corresponding the value in it's column with the minimum's index if it's less than the **current value**. If so, that becomes the new value and on the **predecessor field** it switches with the minimum's index, if not it stays the same.

This step also takes advantage of the toVisit aux structure to update the booleans of each row in the **RDD**.

V. Results

To test I used both the Cluster and my PC (specs in the annex Fig.4. The cluster was used to test situations where the number of nodes in Spark $>$1.

A. 1 Node

	Spark	**Sequential**
Route	Node1 $\rightarrow$ $Node26$ $\rightarrow$ $Node45$	Node1 $\rightarrow$ $Node11$ $\rightarrow$ $Node26 \rightarrow Node89$
Route Time(ms)	13.0	5.0
Computation Time(ms)	6313	6313

TABLE I
100 Nodes, 10% edge chance, node1, node45

	Spark	**Sequential**
Route	Node5 $\rightarrow$ $Node235$ $\rightarrow$ $Node490$	Node5 $\rightarrow$ $Node344$ $\rightarrow$ $Node455 \rightarrow Node490$
Route Time(ms)	131.0	144.0
Computation Time(ms)	275563	9449

TABLE II
500 Nodes, 25% edge chance, Node5, Node490

B. 3 Nodes

For these tests, I used higher node counts (more vertices in the graph) to take advantage of the Cluster. I also only tested with the custom Dataset given by the teacher so it was better to test around with different values/dataset sizes.

	Spark	**Sequential**
Route	Node1 $\rightarrow$ $Node26$ $\rightarrow$ $Node45$	Node1 $\rightarrow$ $Node11$ $\rightarrow$ $Node26 \rightarrow Node89$
Route Time(ms)	13.0	5.0
Computation Time(ms)	6313	6313

TABLE III
100 Nodes, 10% edge chance, node1, node45

	Spark	**Sequential**
Route	TPA $\rightarrow$ MCI $\rightarrow$ SEA $\rightarrow$ KTN $\rightarrow$ $WRG \rightarrow PSG$	TPA $\rightarrow$ MCI $\rightarrow$ SEA $\rightarrow$ KTN $\rightarrow$ $WRG \rightarrow PSG$
Route Time(m)	672.6317768535087	672.6317768535087
Computation Time(ms)	13170	2254

TABLE IV
TPA $\rightarrow PSG$

	Spark	**Sequential**
Route	LGA $\rightarrow SYR \rightarrow JFK$	LGA $\rightarrow SYR \rightarrow JFK$
Route Time(m)	250.16829268292685	250.16829268292685
Computation Time(ms)	13117	2186

TABLE V
LGA $\rightarrow JFK$

	Spark	**Sequential**
Route	Node30 $\rightarrow$ $Node20$ $\rightarrow$ $Node50$	Node30 $\rightarrow$ $Node20$ $\rightarrow$ $Node50$
Route Time(ms)	55.0	55.0
Computation Time(ms)	30389	2209

TABLE VI
100 Nodes, 40% edge chance, node30, node50

	Spark	**Sequential**
Route	Node125 $\rightarrow$ $Node374 \rightarrow Node333$	Node125 $\rightarrow$ $Node374 \rightarrow Node333$
Route Time(m)	109.0	109.0
Computation Time(ms)	113154	64045

TABLE VII
500 Nodes, 25% edge chance, node125, node333

C. *Time Operation*

```
Type quit to exit spark
info Node3 Node8
Route: Node3 -> Node6 -> Node7 -> Node8 with time: 5.0m
Computed in 1402 ms.
time Node3 Node8 4
Added Node3(5) -> Node8(7) with weight: 4
info Node3 Node8
Route: Node3 -> Node8 with time: 4.0m
Computed in 751 ms.
time Node3 Node6 0
Added Node3(5) -> Node6(8) with weight: 0
info Node3 Node8
Route: Node3 -> Node6 -> Node7 -> Node8 with time: 3.0m
Computed in 793 ms.
```

Fig. 1. Time Operation Example 1

```
info Node1 Node8
Route: Node1 -> Node7 -> Node8 with time: 6.0m
Computed in 1268 ms.
time Node1 Node7 20
Added Node1(1) -> Node7(6) with weight: 20
info Node1 Node8
Route: Node1 -> Node5 -> Node2 -> Node7 -> Node8 with time: 8.0m
Computed in 745 ms.
```

Fig. 2. Time Operation Example 2

VI. Analysis and Shortcomings

Visualizing the results, the sequential always beat the Spark implementation. It's to note that once the number of Nodes in the Dataset rose, the difference between them got shorter.

Observing Tables VI and VII, we can see that the difference in performance went down. In the 100 node version, the sequential was $\approx$ 10x faster, while in the 500 node version, the sequential was only $\approx$ 2x faster.

I can induce from this that at a certain high level of Nodes in the graph, would make it so at some point, the Spark implementation would level out with the sequential and at some point overtake it in terms of performance.

A. Bottleneck

Inspecting Spark's visual guide in the browser it was seen the place the process would end up taking more time was in the operation **first()** to get the minimum in each iteration. I couldn't find a way around this as it is required in every iteration to get the new minimum.

The other step that took more time was due to the creation of the CoordinateMatrix. In a single view, this step is the one with higher cost, but it doesn't compare to the amount of time the sum of all the **first()** operations took.

Fig.3 shows what Spark would show as the regular Top 4 heavier operations. To note that **first()** would show per time the minimum needed to be calculated, and in so it would show times the number of nodes in the graph -1 (the source).

B. Time Function

As seen in Fig1 and Fig2, the time function allows the user to modify the graph edges.

In the first example by creating a direct route to the end, the algorithm picks it as the best option. Then by updating the previous best path to have a shorter time overall, it picks it again as the new best one.

In the second example, by making an edge of the found quickest path cost a large number, it returns a new path which coincidentally as all the original Nodes.

Description	Submitted	Duration
reduce at CoordinateMatrix.scala:158 reduce at CoordinateMatrix.scala:158	2020/12/18 16:57:11	0.5 s
first at SSSPSpark.java:139 first at SSSPSpark.java:139	2020/12/18 16:57:12	0.1 s
first at SSSPSpark.java:139 first at SSSPSpark.java:139	2020/12/18 16:57:12	85 ms
first at SSSPSpark.java:139 first at SSSPSpark.java:139	2020/12/18 16:57:12	58 ms

Fig. 3. The regular top 4

VII. Annex

OS Name	Microsoft Windows 10 Pro
Version	10.0.19042 Build 19042
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-7F83NOO
System Manufacturer	Micro-Star International Co., Ltd.
System Model	MS-7C90
System Type	x64-based PC
System SKU	To be filled by O.E.M.
Processor	AMD Ryzen 5 5600X 6-Core Processor, 3701 Mhz, 6 Core(s), 12 Logical Proce...
BIOS Version/Date	American Megatrends Inc. 1.40, 10/29/2020
SMBIOS Version	2.8
Embedded Controller Version	255.255
BIOS Mode	UEFI
BaseBoard Manufacturer	Micro-Star International Co., Ltd.
BaseBoard Product	MPG B550 GAMING CARBON WIFI (MS-7C90)
BaseBoard Version	1.0
Platform Role	Desktop
Secure Boot State	Off
PCR7 Configuration	Binding Not Possible
Windows Directory	C:\Windows
System Directory	C:\Windows\system32
Boot Device	\Device\HarddiskVolume1
Locale	United States
Hardware Abstraction Layer	Version = "10.0.19041.488"
User Name	DESKTOP-7F83NOO\joaor
Time Zone	GMT Standard Time
Installed Physical Memory (RAM)	16.0 GB
Total Physical Memory	15.9 GB
Available Physical Memory	6.75 GB
Total Virtual Memory	23.9 GB
Available Virtual Memory	7.81 GB
Page File Space	8.00 GB
Page File	C:\pagefile.sys

Fig. 4. PC Specs