Une image contenant texte

Description générée automatiquement

|  |
| --- |
| Systems, languages, and paradigms for Big Data |
| Single-source shortest path problem |
|  |

|  |
| --- |
| HU Estelle & CHEKROUN Alexandre  2021 – 2022 |

Table des matières

[Introduction 2](#_Toc92653820)

[Project description 2](#_Toc92653821)

[Working method 2](#_Toc92653822)

[Context 2](#_Toc92653823)

[The problem 2](#_Toc92653824)

[Dijkstra’s algorithm 2](#_Toc92653825)

[Implementation 4](#_Toc92653826)

[Python-Hadoop 4](#_Toc92653827)

[The library 4](#_Toc92653828)

[Hadoop-streaming 5](#_Toc92653829)

[Scalability 5](#_Toc92653830)

[Spark 5](#_Toc92653831)

[Roadmap 6](#_Toc92653832)

[References 7](#_Toc92653833)

# Introduction

## Project description

The task is to find shortest paths from a source node to all other nodes in the graph. This problem is solved by the Dijkstra’s algorithm, which is sequential. The project has a double purpose. First get familiar with Dijkstra’s algorithm, then devise a MapReduce version of the algorithm. As the algorithm is iterative so the MapReduce job must be iterated a certain number of times.

We must provide both a Python-Hadoop streaming and Spark implementation of the algorithm and test it on simple graph. Moreover, we need to test its scalability with big data.

## Working method

Before anything, we searched on the internet some resources to have a better understanding of the Dijkstra algorithm and how to implement it. We found a git repository [6] which implemented Dijkstra algorithm using Hadoop and Spark. It helped us grasp the problem. We also used the book recommended on MapReduce [1] which helped us a lot. Our work is mainly based on this book.

We decided to work on pair programming because we found it more efficient. While one was writing the code, the other could check the code written and figure out what to do next.

## Context

### The problem

In graph theory, the single-source shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph and minimizing the sum of the weights of its constituent edges. It is one of the most common and well-studied problems in graph theory. [1]

### Dijkstra’s algorithm

The Dijkstra’s algorithm finds the shortest path from one node, the source, to all other nodes. [2] This algorithm is very similar to the Breadth-first search algorithm. The only difference is that the distance between neighbors is not equal to 1 but it can vary from a neighbor to another. [3]

Dijkstra's Algorithm works on the basis that any sub-path of the shortest path is also the shortest path between vertices of the sub-path. Dijkstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest sub-path to those neighbors. [4]

The algorithm uses a greedy approach because we find the next best solution hoping that the result is the best solution for the whole problem. [5]

It can be described as follows: [3]

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. The tentative distance of a node v is the length of the shortest path discovered so far between the node v and the starting node.
3. For the current node, consider all its unvisited neighbors and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
4. When we are done considering all the unvisited neighbors of the current node, mark the current node as visited and remove it from the unvisited set.
5. If the destination node has been marked or if the smallest tentative distance among the nodes in the unvisited set is infinity, then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new current node, and go back to step 3.

The principal challenge of the problem is that we face a sequential processing algorithm, and we need to work it out to solve this problem in parallel and with the MapReduce paradigm.

# Implementation

All the code can be found on the Github page:

https://github.com/alchekroun/dijkstra\_hadoop\_spark.

## Python-Hadoop

To convert the Dijkstra’s algorithm to the MapReduce paradigm we used the [1] article as a base source. We split the work in two phases. The Mapper and the Reducer, between them a Shuffle&Sort lookalike function.

### The library

#### Mapper

The Mapper, will, for every node of the graph, calculate the distance from the selected node to its neighbors. The input will be a python dictionary and will return a python list

The pseudo-code of the mapper is:

1: **method**

2:

3: **Emit**

4:

5: **for** **all** **do**

6:

7:

8: **Emit**

#### Shuffle & Sort like

This function will map for every node their possibility. The input will be a python list and will return a python dictionary.

#### Reducer

The Reducer, will, for every possible path, select the minimum weighted path and keep it. The input will be a python dictionary and will return a python dictionary.

The pseudo of the reducer is:

1: **method**

2:

3:

4: **for** **all** **do**

5: **If** **then**

6:

7: **Else if**  **then**

8:

9:

10: **Emit**

#### Main function

The main function named Dijkstra can be directly called and will prepare the data, process it, and return it in an easy workable format. The number of Map Reduce jobs to do can be determined by the diameter of the input graph. To find the diameter we should have to use other library and modify the structure of our code, which can be greedy on time for the algorithm. Hence, we use the proposition of [1] and make 6 iterations for our loop.

### Hadoop-streaming

For the Hadoop-streaming, two classes will be called. For the mapper it is mapper.py and for the reducer it is reducer.py. Both classes call library function. By this way, we could test our code independently from Hadoop-streaming and test scalability.

### Scalability

As the subject asks, we must test the scalability of our implementation. For this, we defined a function to create large data set on demand.

In this example we create a data set with 50 000 nodes and will create 400 000 edges. Our algorithm finds the solution in less than 20seconds. (Figure 1)

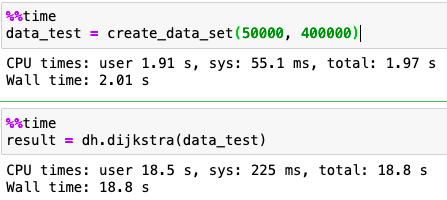


Figure 1 : Scalability experiment

## Spark

The spark solution is based on the previous implementation. As the Hadoop-streaming solution, we use the proposition of [1] and make 6 iterations for our loop. (Figure 2)

Une image contenant texte

Description générée automatiquement

Figure 2 : Spark part of the code

# Roadmap

We delivered a stable version, workable and performant, yet perfectible. We could add multiple “nice features” such as a Panel application user friendly and easily launchable on BinderHub.

* Panel Application
* Spark code optimization
* Hadoop code optimization
* Test coverage score

# References

[1] “Data-Intensive Text Processing with MapReduce” - Jimmy Lin and Chris Dyer

[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.).

[3] Wikipedia page <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

[4] Programiz page <https://www.programiz.com/dsa/dijkstra-algorithm>

[5] Hector Ortega-Arranz; Diego R. Llanos; Arturo Gonzalez-Escribano; Morgan & Claypool Publishers, Morgan and Claypool Life Sciences (2014). “The Shortest-Path Problem: Analysis and Comparison of Methods”

[6] Bilal Elchami’s git repository <https://github.com/bilal-elchami/dijkstra-hadoop-spark>