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](#_Toc92547251)

[Project description 2](#_Toc92547252)

[Context 2](#_Toc92547253)

[The problem 2](#_Toc92547254)

[Dijkstra’s algorithm 2](#_Toc92547255)

[Implementation 3](#_Toc92547256)

[Python-Hadoop 3](#_Toc92547257)

[The library 3](#_Toc92547258)

[Hadoop-streaming 3](#_Toc92547259)

[Scalability 3](#_Toc92547260)

[Spark 4](#_Toc92547261)

[Scalability 5](#_Toc92547262)

[References 6](#_Toc92547263)

[Annexes 7](#_Toc92547264)

# 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.

## 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. 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 in the appendix and on the github page:

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

## Python-Hadoop

In order to convert the Dijkstra’s algorithm to the MapReduce paradigm we used the [1] article as a base source. We split the work between two phases. The Mapper and the Reducer Between them a Shuffle&Sort lookalike function. will map for every node their possibility.

### 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

#### 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.

#### 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.

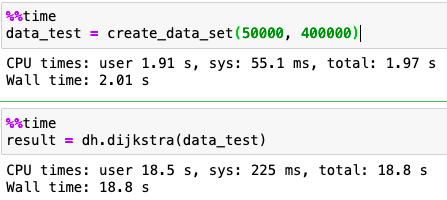
### Hadoop-streaming

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 of these 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 in order 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.



## Spark

# 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>

# Annexes