Universite Paris Dauphine

Big Data

Single-source shortest path - Djikstra Algorithm

Students
Elie Abi Hanna Daher
Bilal El Chami
Badr Erraji

Professor
Mr Dario Colazzo

January 30, 2018

Contents

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.

A long side the implementation, a scalability experiments is needed to check the performance of the algorithm implemented.

2 Dijkstra Algorithm

The Dijkstra's algorithm finds the shortest path from source to all other nodes. The djikstra 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 :

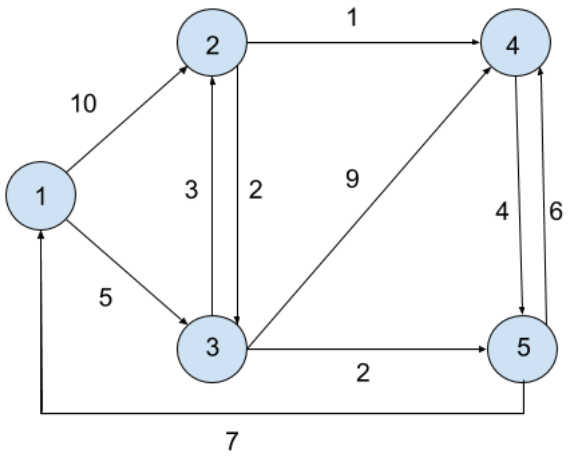

Figure 1: Example of a graph

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.

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 ∈ neighbors do
4:      dist ← distance + neighbor.distance
5:      p ← 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 ← ∞
3:     nodePath ← ∅
4:     graph ← ∅
5:     if isNode(v) then
6:         d_min ← v.distance
7:         nodePath ← v.path
8:         graph ← v.neighbors
9:     else
10:        if d_min ≤ v.distance then
11:            d_min ← v.distance
12:            nodePath ← v.path
13:     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 → Reduce1 → Map2 → Reduce2 → 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.

4 Results

4.1 Hadoop

4.2 Spark

5 Performance

References

[1] *Cloud Computing Lecture 4 - Graph Algorithms with MapReduce.* Jimmy Lin, The iSchool, University of Maryland, February 6, 2008.

A Appendix example

This an example of appendix