Project report
Connected Component Finder in MapReduce

Master IASD

Pierre-François Massiani

final project for the class
Systems and Paradigms for Big Data

June 7th, 2020

1 Introduction

Distributed computing In many areas of research and industry, graphs are a very convenient way of representing data. Social networks, proteins, point clouds for 3D imagery, ... can be modeled as graphs, and one of the many ways to extract data from such graphs consists in finding its *connected components*. A connected component is a subgraph of the original graph from which any two vertices are connected by a path. As graphs keep getting bigger and bigger, new challenges arise. For example, a single computer is generally not able to store a graph of reasonable size by today's standards, let alone find its connected components. Such graphs are typically stored in *distributed storage systems*, that is groups of computers in which each machine stores and performs operations on the part of the graph it has. A distributed storage system generally takes the form of a *cluster* running a *distributed filesystem software*, such as HDFS. This architecture has shown to be a real game changer for *big data*, since it can be very robust (data can be easily duplicated across several machines) and gets over the limitations of handling the data on a single computer. However, in order to take advantage of the physical architecture of the cluster, computer scientists must write specific algorithms that can naturally be parallelized. MapReduce is such a programming paradigm, and most distributed filesystems (such as Hadoop) also come with an implementation of it.

MapReduce and Spark The MapReduce programming paradigm was first proposed in 2004, and 5 years later, Hadoop published an implementation of it seamlessly integrated with its very efficient distributed filesystem, HDFS. In order to guarantee robustness, Hadoop's implementation of MapReduce frequently writes and reads on hard drives, which is a serious bottleneck for speed. Shortly after Hadoop's MapReduce implementation was published, a UC Berkeley PhD. student developed Spark in order to speed up computations. The paradigm Spark is based upon is slightly different from MapReduce[1], but its main advantage is speed: Spark keeps the data in the RAM of each node of the cluster, with only a limited interaction with the hard drive. This enables Spark to be 10 to 100 times faster than MapReduce. This comes at the price of robustness: in case a worker fails, all the computations it has done so far are lost. In order to address this issue, Spark has come up with the idea of *resilient distributed datasets*: instead of storing the results of the computations, Spark stores the computation graph.

[1]But all operations that can be done in MapReduce can be done in Spark, and conversely.

This has major advantages, such as enabling *lazy evaluation* of RDDs, or optimizating the global computations before actually performing them.

The CCF algorithm The rise of distributed computation paradigms such as MapReduce and Spark has enabled engineers and researchers to come up with highly efficient, parallelizable algorithms. In [2], the authors describe the "Connected Component Finder"(CCF), an iterative algorithm based on MapReduce that finds the connected components in a graph. The authors achieve state-of-the-art performance when finding connected components in huge graphs, such as the Web-Google dataset, which can be found in [3].

In this project, we propose a PySpark implementation of the CCF algorithm [2]. We implemented the two variants described in the original article, and the code can be found in Appendix A. We start by describing and commenting our implementation in Section 2, and present our results in Section 3. Appendix B provides a working code to reproduce the results.

In order to easily reproduce the results on your machine, you will also find the code on GitHub at the following address:

`https://github.com/PFMassiani/ccf-pyspark`

2 Implementation

2.1 Data description

Formally, an undirected graph is a tuple $G = (V, E)$ where V is the set of vertices and $E \subset V \times V$ is the set of edges. The input data that the algorithm expects is E, that is, an unordered list of pairs of vertices. The goal of the algorithm is, for each node of the graph, to map it to the identifier of its connected component. For simplicity, we define the identifier of a component to be the smallest identifier of the vertices it contains. With this convention, connected components can be identified to vertices, and the output of the algorithm is again a list of $(v_1, v_2) \in V \times V$, where v_1 is the node of the graph, and v_2 is the node with the smallest identifier in its connected component.

2.2 Algorithms

We do not provide here an explanation of the original algorithms or why they work: we recommend that you read the original article [2] if you want more information on that topic. The goal of this part is to comment the implementation choices that we made.

2.2.1 CCF-Iterate

The authors present two versions of CCF-Iterate: with, and without secondary sorting. We have implemented both versions, and also present a variant of the without secondary sorting version.

Without secondary sorting: offline minimum This function can be found in Appendix A.2.1. It is the vanilla implementation of Algorithm 1 from [2]. The main bottlenecks are the call to `groupByKey` - which triggers a shuffle - and the fact that the minimum is computed by calling the `min` function. The first bottleneck cannot be avoided, since a shuffle is needed to gather all the vertices of an estimated component in the same partition. However, the call to `min` forces the whole component to be loaded in the RAM, and can lead to `OutOfMemory` errors since the space complexity of this approach is linear in the size of the largest connected component. The other two approaches try to address this issue by computing the minimum in a more efficient way.

Without secondary sorting: online minimum The code for this version can be found in Appendix A.2.2. The main difference with the previous approach is that, instead of simply calling the `groupByKey` method and end up with all the nodes of a connected component in big array of which we need to find the minimum, we compute the minimum online by calling `reduceByKey`. Starting from a list of `(vertex1, vertex2)` pairs, we transform it into a list of `(vertex1, ([vertex2],vertex2)`. This trick enables us to keep track of the minimum in the second value by simply comparing the two running minima of the arrays in the first value. We believe that this method has similar space performance to the secondary sorting one, but is has the advantage of being much simpler to implement since we do not need to worry about the partitioning of the data.

Python limitation You may notice that, in the functions `concat_and_min` and `ccf_iterate_online_min`, we use Python list concatenation and list comprehension instead of `itertools.chain` and `itertools.starmap`. This has two major drawbacks, that make this method a lot less effective:

- it forces the evaluation of the arrays;
- it keeps rewriting the arrays in different places of the memory when they are concatenated.

Using iterators would solve these two issues, since they are lazily evaluated. But this raises another issue, which is a fundamental limitation in Python: to do this, we would need to stack a lot of iterators, and this raises a `RecursionError` when the size of the connected component gets bigger. However, we would like to point out that this is not a limitation of the *algorithm*, but of the language[2]. Using another language (such as Scala) or a smart way to chain and map functions over iterators should solve this issue. The consequence is that we do not expect this method to be better than the previous one. Worse: with all the concatenations happening, this method should actually be slower than the offline minimum one.

With secondary sorting This is the vanilla implementation of Algorithm 3 from [2]. The code can be found in Appendix A.2.3. We first give a brief overview of the secondary sorting programming paradigm in MapReduce before explaining how we implemented it. For a more thorough introduction to this technique, please see [4]. For another example of a PySpark implementation of secondary sorting, see [1].

The secondary sorting design pattern The goal of secondary sorting is, given a list of `(key,value)` pairs, group the values with the same key to get a list of `(key, [list of values])` where the list of values is ordered. MapReduce - or Spark - does not provide such a built-in functionality. However, both are very efficient in sorting items by their *keys* (but don't guarantee anything about the order of the values). Secondary sorting uses this efficient sorting of the keys to sort the values, by creating a *composite key* that contains the value so MapReduce or Spark can natively sort them. In practice, implementing secondary sorting breaks down to four steps:

1. create the composite key;
2. repartition the data so all the entries with the same *original* key end up in the same partition[3];
3. sorting each partition with the desired order;
4. grouping the data in a list, in a `groupByKey` fashion that preserves the order and the partitioning (to avoid a shuffle).

[2]Solving this problem should be possible in Python, but this is not the goal of this project.

[3]This is similar to what `groupByKey` followed by `flatMap` would do.

Implementation in PySpark PySpark provides a nice function to do steps 2 and 3 simultaneously called `repartitionAndSortWithinPartitions`. In our case, the order we want is simply the natural lexicographical order on the keys, so the default parameter works. For step 4, we use the `mapPartitions` function and map the current estimate of the minimum with an iterator over the connected component, so the whole array is not loaded in memory at once. By being this cautious about how the values are grouped, the minimum of the current estimate of the connected component is simply the first item of the iterator.

2.3 Main loop

The CCF algorithm is iterative: we need to repeat operations `ccf-iterate` and `ccf-dedup` until no new pairs are generated. In Python, this is simply done by doing a `for` loop. However, such loops do not get along well with the lazy evaluation feature of Spark. Indeed, the stopping condition depends on the result of the computation, so we need to force Spark to evaluate the RDD at the end of each iteration. It is then absolutely critical to raise `persist` flags so the whole computation graph is not recomputed at each iteration. The code doing this can be found in Appendix B

3 Results

3.1 On a toy example

The authors provide a simple toy example in Figure 5 of [2], which is very convenient to test our implementation. All methods correctly find the connected components. Moreover, we suspect that there is a typo in Figure 5.a of [2], since the vertex H should be also mapped to vertex G after the reducer, since we have $G < H$. It is the case in our implementation.

3.2 On real graphs

We have tested our algorithm on two real-world graphs taken from [3]: the "Arxiv High Energy Physics paper citation network" and the "Web-Google" dataset. The first one represents the graph of scientific citations in the field of high energy physics on ArXiv, where two papers are connected if one cites the other, and the second one is the dataset published by Google in 2002 as part of the Google Programming Contest. The latter is also used in the original

	Toy graph	Citations	Web-Google
Offline	1.94	17.2	351
Online	2.16	27.7	5660
Secondary	2.07	26.1	538
Iterations	4	7	8

Table 1: Results. The values in the first three lines are the execution times, in seconds. The last line is the number of iterations required for convergence (all methods require the same number of iterations for a given dataset). All computations were performed on DataBricks, Community Edition.

article as a benchmark, so we will be able to compare our performances with the authors'. All results are presented in Table 1.

3.3 Comments

As expected, the Online algorithm gets absolutely terrible as the size of the graph increases. Surprisingly, the Secondary sorting algorithm is slower than the Offline one, contrary to what is stated in the article (both algorithms should be comparable for small graphs, with Secondary outperforming Offline when the graph gets bigger). We interpret this as being caused by the hardware architecture we are using, since DataBricks only allocates one node for our computations.

We also achieve comparable execution times as the ones that the authors got. At first, this is deceiving, because Spark is supposed to be much faster than MapReduce. However, one should bear in mind that our tests were realised on a *single* node with two cores and 15 GB of RAM, whereas the authors could use a 80 nodes cluster, each with 8 cores. Consequently, our implementation is indeed approximately a hundred times faster than the one of the authors[4], as stated in the introduction.

4 Conclusion

In this project, we have implemented in PySpark the Connected Component Finder algorithm from its MapReduce specification in [2]. The authors present two variants, with one using the secondary sorting design pattern

[4]Such a simple reasoning on orders of magnitude cannot give a precise value for the speed increase, since more nodes means more computer power available, but also increased communication times.

and both of which we have implemented. We also propose a third variant to solve the same issue as the secondary sorting technique, but our variant does not scale well due to Python limitations on chaining and composing iterators. At comparable computer power available, our vanilla implementations are much faster than the ones of the authors thanks to Spark's in-memory computation paradigm. However, our secondary-sorting method seems to be systematically slower than the other one, which is not the expected result. We suspect that this discrepancy stems from the hardware we are using for our tests.

A Algorithms

A.1 Imports

The following imports are required to make the algorithms work:

```
from pyspark.rdd import portable_hash
from itertools import groupby, starmap, chain, filterfalse
```

The following auxiliary functions should also be defined:

```
def partition_by_first_value(key):
  return portable_hash(key[0])
def get_sorted_values(lines):
  groups_with_key_duplicates = groupby(lines, key=lambda x: x[0])
  return groups_with_key_duplicates
def concat_and_min(listval1, listval2):
  return (listval1[0] + listval2[0], min(listval1[1], listval2[1]))
```

A.2 CCF-Iterate

We provide three implementations of CCF-Iterate.

A.2.1 CCF-Iterate with offline minimum computation

This code is an implementation of Algorithm 1 from [2], with the minimum being computed after the grouping by keys. In the original algorithm, the computation of the minimum requires two iterations over the values, but we rely on the fact that Spark handles Python arrays instead of iterables to compute it with the `min` function.

```
def ccf_iterate_offline_min(graph):
  sym = graph.flatMap(lambda edge : [edge, edge[::-1]])
  grps = sym.groupByKey()
  grps_mins = grps.map(
    lambda keyval : (keyval[0], (keyval[1], min(keyval[1])))
  )
  fltr = grps_mins.filter(
    lambda keyval : keyval[1][1] < keyval[0]
  )

  new_pairs = sc.accumulator(0)
  def pair_and_count(keyval):
    minval = keyval[1][1]
    out = [
      (value, minval) for value in keyval[1][0] if value != minval
    ]
    new_pairs.add(len(out))
    return [(keyval[0], minval)] + out

  return fltr.flatMap(pair_and_count), new_pairs
```

A.2.2 CCF-Iterate with online minimum computation

This code is very similar to the one in Section A.2.1, except that the minimum is computed online as the group is being created in the `reduceByKey`. See Section 2 for the practical difference with the previous algorithm. Also note that the function passed to the `reduceByKey` is *not* rigorously associative (as it should be), since arrays are ordered. In practice, this order does not matter, so this is not a problem.

```
def ccf_iterate_online_min(graph):
  sym = graph.flatMap(lambda edge : [edge, edge[::-1]])
  frmt_for_reduce = sym.map(
    lambda keyval : (keyval[0], ([keyval[1]], keyval[1]))
  )
  grps_mins = frmt_for_reduce.reduceByKey(concat_and_min)
  fltr = grps_mins.filter(
    lambda keyval : keyval[1][1] < keyval[0]
  )
```

```
  new_pairs = sc.accumulator(0)
  def pair_and_count(keyval):
    minval = keyval[1][1]
    key_min_pair = (keyval[0], minval)
    other_pairs = [(val, minval) for val in keyval[1][0] if minval != val]
    new_pairs.add(len(other_pairs))
    return chain(key_min_pair, other_pairs)

  return fltr.flatMap(pair_and_count), new_pairs
```

A.2.3 CCF-Iterate with secondary sorting

In this code, we make sure that the group of edges is ordered after the shuffle. To do so, we use a technique called *secondary sorting*, as explained in Section 2. The extra indexing for `keymingroup` is simply caused by the fact that `groupby` returns something of the form `(key,[list of (key,value)])` instead of `(key,[list of values])`.

```
def ccf_iterate_secondary_sorting(graph):
  sym = graph.flatMap(lambda edge: [edge, edge[::-1]])
  composite_key = sym.map(lambda edge: (edge,None))
  partition_sorted_composite = composite_key.\
    repartitionAndSortWithinPartitions(
      partitionFunc=partition_by_first_value
    )
  partition_sorted = partition_sorted_composite.map(
    lambda compkey:tuple(compkey[0])
  )
  sorted_groups = partition_sorted.mapPartitions(
    get_sorted_values, preservesPartitioning=True
  )
  groups_with_min = sorted_groups.mapValues(
    lambda group: (next(group), group)
  )
  fltr = groups_with_min.filter(
    lambda keymingroup: keymingroup[1][0][1] < keymingroup[0]
  )

  new_pairs = sc.accumulator(0)
  def pair_and_count(keymingroup):
```

```
    minval = keymingroup[1][0][1]
    key_min_pair = zip([keymingroup[0]],[minval])

    def pair_and_increment(duplicated_key, value):
      new_pairs.add(1)
      return (value, minval)
    other_pairs = starmap(
      pair_and_increment,
      keymingroup[1][1]
    )
    return chain(key_min_pair, other_pairs)
  return fltr.flatMap(pair_and_count), new_pairs
```

A.3 CCF-Dedup

The code of CCF-Dedup is much simpler:

```
def ccf_dedup(graph):
  temp = graph.map(lambda keyval : (keyval, None))
  reduced = temp.reduceByKey(lambda v1,v2 : v1)
  return reduced.map(lambda keyval:keyval[0])
```

B Main code

The main loop of the program is the following:

```
def find_connected_components(graph, method):
  if method=='online':
    ccf_iterate = ccf_iterate_online_min
  elif method=='offline':
    ccf_iterate = ccf_iterate_offline_min
  elif method=='secondary':
    ccf_iterate = ccf_iterate_secondary_sorting
  new_pairs = -1
  n_loops = 0
  while new_pairs != 0:
    graph, acc = ccf_iterate(graph)
    graph = ccf_dedup(graph)
    graph.persist()
```

```
    graph.foreach(lambda x:x)
    new_pairs = acc.value
    n_loops += 1
  return graph, n_loops
```

References

[1] Sebastian Brestin. Spark secondary sort. `https://www.qwertee.io/blog/spark-secondary-sort/`, April 2019.

[2] Hakan Kardes, Siddharth Agrawal, Xin Wang, and Ang Sun. Ccf: Fast and scalable connected component computation in mapreduce. In *2014 International Conference on Computing, Networking and Communications (ICNC)*, pages 994–998. IEEE, 2014.

[3] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[4] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.