Finding connected components in graphs with Map Reduce

Florent Brissaud, Abdoulaye Doucoure, Chabane Debiche, Marie Coly

{florent.brissaud, abdoulaye.doucoure, chabane.debiche, marie.coly}@dauphine.eu

PSL - Université Paris Dauphine *Executive Master IA et Science des Données* (IASD)

Pr Dario Colazzo, cours *Traitement distribué de données massives sur Hadoop, Spark et Kafka*

18 octobre 2022

Table of Contents

1. Description of MapReduce and the CCF algorithm

1.1. MapReduce for finding connected components in graphs

MapReduce is a programming model for expressing distributed computations on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity servers[1]. The basic idea is to partition a large problem into smaller subproblems, given that the sub-problems are independent and can therefore be proceeded in parallel. The result is then obtained by combining the intermediate outputs from the split problems.

The structure of the basic data of MapReduce is *key-value* pairs. For example, *keys* are ids of objects and *values* are characteristics of these objects.

The MapReduce approach then consists in two main steps:

- **Mapper**, which applies a common function to every *key-value* pair that is given as an input, and generates intermediate *key-value* pairs (less, equal, or more than the number of input pairs) that are "emitted" to the reducer, i.e. (k_1, v_1) -> $[(k_2, v_2)]$;
- **Reducer**, which groups all received *key-value* pairs by *keys*, to apply a common function to each *key* and its associated *values*, and emitting outputs *key-value* pairs, i.e. $(k_2, [v_2])$ -> $[(k_3, v_3)]$.

Note that the MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer.

MapReduce is therefore a promising tool for implementing graph algorithms since:

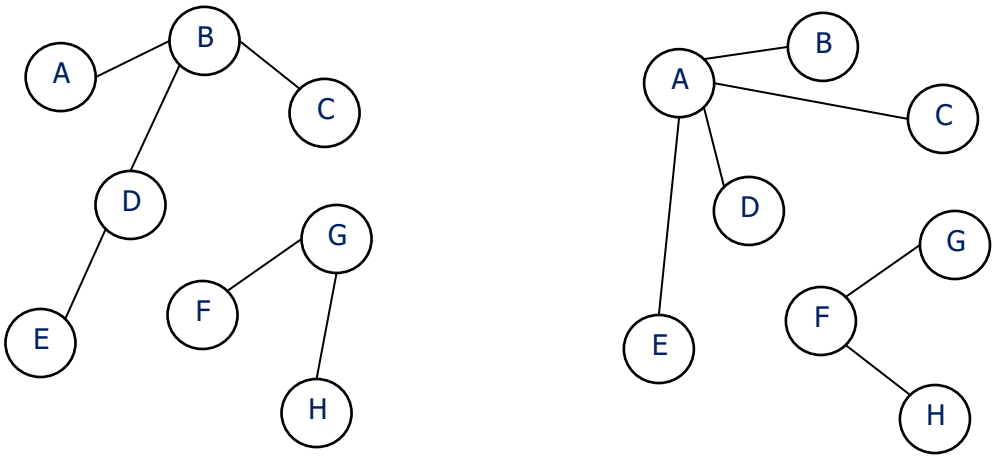
- Graphs can be used to model several networks (e.g. web, social networks…) that can contain millions of nodes and edges, requiring an important power of computations for analyses;
- Graphs can be simply modelled by links (for directed graphs) or edges (for undirected graphs), which are defined by two parameters (ids of the connected nodes).

In the following, the graphs are undirected and modelled by a list of connections between nodes, such as a *key-value* that corresponds to *node1-node2* where *node1* and *node2* are two different nodes connected within the graph. Such a graph can, for example, depict connections within a social network where nodes are people.

The studied algorithm, named *Connected Component Finder (CCF)*[2], aims at finding the connected components in a graph, that is, the sets of nodes that are connected to each other, directly or indirectly. The output of the algorithms is a graph where each node is only connected to the node with the lowest value of the component where the node is contained. For example, the graph in Figure 1 (a) produces the graph in Figure 1 (b) as a result. (Note that this example is the same that is used by Figure 5 in [Kardes *et al.*, 2014][2].)

In terms of *key-value* pairs, the results are sets of $(k_i, v_{i,1})$, $(k_i, v_{i,2})$, …, $(k_i, v_{i,ni})$, with $i = 1, …, n$, where n is the number of connected components, and n_i is the number of nodes in the i^{th} component, which contains node k_i (the node with the smallest value of the component) and nodes $v_{i,1}$, $v_{i,2}$, …, $v_{i,ni}$. Note that all k_i's and $v_{i,j}$'s are different and the sum of n_i's plus n is equal to the number of nodes in the graph.

[1] J. Lin, C. Dyer. *Data-Intensive Text Processing with MapReduce*. University of Maryland, 2010.
[2] H. Kardes, S. Agrawal, X. Wang, A. Sun. *CCF: Fast and scalable connected component computation in MapReduce*. International Conference on Computing, Networking and Communications (ICNC), 2014.



(a) (b)

Figure 1. Input graph (a) and resulting CCF graph (b)

1.2. The CCF algorithms using MapReduce

The main idea of the CCF algorithms proposed by [Kardes *et al.*, 2014][2] is to process by sub-components where each sub-component is defined by a node plus its directly connected nodes.

For example, in Figure 1 (a), the sub-component defined by node D contains nodes D, E, and B. In terms of MapReduce formalism, the inputs are (D, E) and (D, B) and the algorithm applied to this sub-component produces (B, E) and (B, D) as outputs (since B represents the smallest id of the three nodes), that is, connection (D, E) is "changed" by connection (B, E). As another example, applying the algorithm to the sub-component defined by node B would not produce any change because node B is already directly connected to nodes with only higher ids.

This proposed approach is suitable for MapReduce due to the following properties:

- Each sub-component can be processed independently of the others;
- Processing a sub-component never "breaks" a component (i.e. all the nodes of the sub-component that is processed remain part of the same component);
- If processing a sub-component has an effect on the graph, it always tends to the final expected results (i.e. breaking a connection between two nodes is always for creating connections to another node with a smaller id);
- While the final expected result is not obtained, it is always possible to process at least a sub-component affecting the graph.

For the first point above, it is however noticed that some issues must be handled to apply accurately the algorithm to several sub-components at the same time:

- To process sub-components having nodes in common, it is required to define each connection in both "directions" (e.g. (A, B) and (B, A)). This is performed during the mapper by emitting both *key-value* and *value-key* for each connection.
- The processing of different sub-components can produce same connections as outputs. To avoid duplicate connections, a "CCF-Dedup" algorithm is used, while all the identical *key-value*'s are merged together to emit a single one.

Another issue of the CCF algorithm is during the reducer, while all the *values* of any *key* (i.e. all the nodes connected to the one used to define the sub-component) must be read to find the lowest id. In fact, the iteration must then be performed twice: once for finding the node with the lowest id, and once for creating connections between this node and each of the other nodes. This double iteration can be avoided using certain MapReduce implementations. For example, Google's implementation allows the programmer to specify a secondary sort key for ordering the *values*[1]. However, with the Hadoop implementation of MapReduce, the *values* are arbitrarily ordered, and the sorting process can only be done with a dedicated function.

Finally, based on the above properties, the CCF algorithm is stopped when applying the process to each sub-component (defined by each node) does not produce any change in the graph. This condition is checked by incrementing a counter within the CCF algorithm after each creation of a new connection. When this counter is not incremented during the run of the algorithm, the final expected result is obtained (cf. graph (b) in Figure 1 obtained from graph (a) in Figure 1).

1.3. Designed Algorithms

1.3.1. Algorithm A: Single sorting

CCF-Iterate

```
map(key, value)
    emit(key, value)
    emit(value, key)

reduce(key, < iterable > values)
    min ← key
    for each (value ∈ values)
        if(value < min)
            min ← value
        valueList.add(value)
    if(min < key)
        emit(key, min)
        for each (value ∈ valueList)
            if(min ≠ value)
                Counter.NewPair.increment(1)
                emit(value, min)
```

CCF-Dedup

```
map(key, value)
    temp.entity1 ← key
    temp.entity2 ← value
```

```
    emit(temp, null)

reduce(key, < iterable > values)
    emit(key.entity1, key.entity2)
```

1.3.2. Algorithm B: Secondary sorting

CCF-Iterate

```
map(key, value)
    emit(key, value)
    emit(value, key)

reduce(key, < iterable > values)
    min Value ← values.next()
    if(min Value < key)
        emit(key, min Value)
        for each (value ∈ values)
            Counter.NewPair.increment(1)
            emit(value, min Value)
```

CCF-Dedup

```
map(key, value)
    temp.entity1 ← key
    temp.entity2 ← value
    emit(temp, null)

reduce(key, < iterable > values)
    emit(key.entity1, key.entity2)
```

2. Proposed implementation

"We want to obtain [...] the mapping from each node in the graph to its corresponding componentID (smallest node id in each connected component as the identifier of that component)."

"The need to distribute the work informs the design."

"The module should output a mapping table from each node in the graph to the smallest node id in its corresponding connected component."

2.1. Spark environment

```
from pyspark import SparkContext, SparkConf
# gives options to provide configuration parameters passed to spark
context, entry point for spark environment
```

```
from pyspark.rdd import RDD # for RDDs

from pyspark.sql import SparkSession # for Dataframes
```

```
#Init sparkSession

warehouseLocation = "file:${system:user.dir}/spark-warehouse"
spark = SparkSession.builder.appName("Projet IASD - Spark MapReduce") \
.config("spark.sql.warehouse.dir", warehouseLocation) \
.enableHiveSupport() \
.getOrCreate()
```

i.

2.2. Datasets

2.2.1. Data scraping

```
path = "/FileStore/tables/web_Google-1.txt"
dbutils.fs.ls(path)
```

```
dbutils.fs.ls("/FileStore/tables/") # list files and folders in
directory
```

2.2.2. Data transformation

```
# move web_Google.txt from dbfs to local file system
dbutils.fs.cp(path, 'file:/tmp/web_Google.txt') # copy path towards
file:/tmp

localPath = 'file:/tmp/web_Google.txt'
```

```
# transform dataset files into RDD
data = sc.textFile(localPath)
```

We notice that the file starts with commented lines describing the file structure of 5105039 Edges, linking 875713 Nodes in the textual format 'FromNodeId\tToNodeId', as a tuple.

2.3. Mapper

The first 4 lines must be ignored because they only characterize the schema.

These data appear to be entered manually.

```
# d.split("\t")[:2] templates the split of an edge into an isolated
ordered pair of nodes
# map(lambda d: int(d), d.split("\t")[:2]) formats each node of the pair
into sorts-optimized integer
# tuple(map(lambda d: int(d), d.split("\t")[:2])) creates a reducible
tuple from a pair of nodes w/ the origin node as key and the destination
node as value
```

```
# map(lambda p : tuple(map(lambda d: int(d), d.split("\t")[:2]))) apply
to each edge the tuple mapping

# data.filter(lambda d: d[0] != '#') # gets rid of comments

optimized_data = data.filter(lambda lines: lines[0] != '#').map(lambda
edge : tuple(map(lambda node: int(node), edge.split("\t")[:2])))
```

```
optimized_data.take(10)
```

```
Out[15]: [(0, 11342),
 (0, 824020),
 (0, 867923),
 (0, 891835),
 (11342, 0),
 (11342, 27469),
 (11342, 38716),
 (11342, 309564),
 (11342, 322178),
 (11342, 387543)]
```

```
Command took 2.16 seconds
```

2.4. Reducer

"CCF-Iterate job generates adjacency lists AL = (a1, a2, ..., an) for each node v."

2.4.1. User-defined UDF reducer

"*The output of the map phase is put back together for each key to independently process the values for each key in parallel.*"

The "*.next()*" keyword in pseudocode suggests using an iterator. Furthermore, it allows to indicate with precision, that this iterator is traversed by a native method "*next()*" to be sorted.

The pyspark "*yield*" keyword remains particularly suitable to do so because "*They're also useful in the same cases where list comprehensions are used, with an added benefit: you can create them without building and holding the entire object in memory before iteration. In other words, you'll have no memory penalty when you use generator expressions*"[3].

[3] https://realpython.com/introduction-to-python-generators/#understanding-the-python-yield-statement

2.4.2. Single sorting reducer

The following code represents the Figure 2 related multiple passes.

```
# break the adjacency list into pairs
def singleSortingReduce( # 'start with the initial edge list to
construct the first degree neighbourhood of each node'
    edgesList): # 'For the first iteration, this job takes the initial
edge list as input'
  # abstract 'key and value pairs' to be reduced by UDF
  key, iterable_values = edgesList # 'key, < iterable > values'
  # assume that the key's node is the minimal value of its adjacency
list
  min = key # 'min ← key'

  # 'go over all the values to find the minValue and store all the
values in a list'

  # initialize an non iterated adjacency list to search for the min
valued edge from
  valueList = [] # 'If multiple passes are needed, the values should be
stored in a list'
  # copy each node's adjacency list to look for the min node inside
  for value in iterable_values: # 'for each (value ∈ values)'
    # search for the minimum valued edge
    if value < min: # 'if(value < min)'
      # update min value
      min = value # 'min ← value'
    # add each value of the adjacency list into the copy
    valueList.append(value) # 'valueList.add(value)'
    # nothing is emitted if the pair's value is not < key

  # in case the root, being the key of the adjency list, is not the
minimal value:
  if min < key: # 'if the node id of this node v_id is larger than the
min node id a_min in the adjacency list: if(min < key)'
    # emit a spark-optimized reduceable pair with min node as value for
key as key
    yield( # 'In MapReduce, values can be iterated just once without
loading all of them into memory'
        ( # 'emit(key, min)'
```

```
            key, min)) # 'emit the < key; minValue > pair [...] (v_id,
a_min)'
    # check for each values in the adjacency list to be compared to its
confirmed min
    for value in valueList: # 'where a_i ∈ AL': 'for each (value ∈
valueList)'
      # compare this confirmed min with the value being checked to
prepare a symmetric tuple so that the node on the right side of the pair
will be mapped as well
      if min != value: # 'where [...] a_i <> a_min: if(min <> value)'
        # make sure the stop criterion is not activated, with its value
initiated as false, until the adjacency list root is not starting with
the minimal value
        NewPairCounter1.add(1) # 'increase the global NewPair counter by
1: Counter.NewPair.increment(1)'
        # emit a second spark-optimized distribution from the adjacency
list, with min as the MapReduce value so that it can be mapped for CCF
again
        yield( # 'In MapReduce, values can be iterated just once without
loading all of them into memory'
            ( # 'emit(value, min) [...] for all other values as < value;
minValue >'
                value, min)) # 'creates [...] a pair for each (a_i,
a_min)' w/ node value, from the adjacency list, to be mapped as the
minimal value MapReduce key, so that the minimal value can be mapped for
CCF again
      # 'if v_id is smaller than a_min, we do not emit any pair [...]
and there is no need for further iterations'
    # 'If the minValue is larger than key, we do not emit anything'
```

2.4.3. Secondary sorting reducer

The following code represents Figure 3 related Adjusting memory utilization.

```
# break the adjacency list into pairs
def secondarySortingReduce( # 'start with the initial edge list to
construct the first degree neighborhood of each node'
    edges_list): # 'For the first iteration, this job takes the initial
edge list as input'
  # abstract adjacency list to be reduced by UDF
  key, values = edges_list  # 'key, < iterable > values'
  # Add minimal value to key to be considered in sortByKey
  minValue = sort(values)[0] # minValue ← values.next() complexity is
optimized in this algorithm by a transformation in memory
  # copy each node's adjacency list to look for the min node inside
  if minValue < key: # if(minValue < key)
    # emit a spark-optimized reduceable pair with min node ‘minValue’ as
value and ‘key’ as key
    yield( # 'In MapReduce, values can be iterated just once without
loading all of them into memory'
        (
            key, minValue)) # emit(key, minValue)
    # check for each values in the adjacency list to be compared to its
confirmed min
    for value in valueList: # 'where a_i ∈ AL': 'for each (value ∈
values)'
      # # make sure the stop criterion is not activated, with its value
initiated as false, until the adjacency list root is not starting with
the minimal value
      NewPairCounter2.add(1) # 'increase the global NewPair counter by
1: Counter.NewPair.increment(1)'
      yield( # 'In MapReduce, values can be iterated just once without
loading all of them into memory'
          ( # 'emit(value, min) [...] for all other values as < value;
minValue >'
              value, minValue)) # emit(value, minValue) with node value,
from the adjacency list, to be mapped as the minimal value MapReduce
key, so that this minimal value can be mapped for CCF again
      # 'if v_id is smaller than a_min, we do not emit any pair [...]
and there is no need for further iterations'
    # 'If the minValue is larger than key, we do not emit anything'
```

2.4.4. CCF-Dedup

CCF-Dedup 'just deduplicates the output of the CCF-Iterate job' so we use RDD.distinct() to do so. Indeed its Pyspark implementation[4], as described below, matches the article's pseudocode:

```
    def distinct(self, numPartitions=None):
        """
        Return a new RDD containing the distinct elements in this RDD.

        Examples
        --------
        >>> sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())
        [1, 2, 3]
        """
        return self.map(lambda x: (x, None)) \ # map(key, value) | temp.entity1 ← key
| temp.entity2 ← value | emit(temp, null)
                   .reduceByKey(lambda x, _: x, numPartitions) \ # reduce(key, <
iterable > values)
                   .map(lambda x: x[0]) # emit(key.entity1, key.entity2)
```

2.5. Map-reduce job

```
# Spark Context Accumulator initialized outside the loop
NewPairCounter1 = sc.accumulator(0) # 'global NewPair counter variable'
NewPair1 = True
singleSortingData = optimized_data
```

We turn Spark readable objects into MapReduce readable tuples so that "*the input which is given as (key,value) pairs is split up among multiple machines to be worked on in parallel*".

```
# check CCF-Iterate stop criterion
while NewPair1 == True:
    # Iterate
    # Stop criterion
    NewPair1 = False
    # accumulator's value to be initialized to 0 to be in a Boolean
format
    NewPairCounter1.value = 0
```

[4] https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd.html#RDD.distinct

```
    # CCF-Iterate

    # Map to create a symmetry of each tuple so both elements can be
grouped by key
    iterateMap1 = singleSortingData.flatMap(
        lambda pair: ( # ‘for each edge <a; b >, mapper emits both < a;
b >, and < b; a > pairs’
        pair, pair[ # ‘so that a should be in the adjacency list of b
and vice versa’
          ::-1])) # ‘map(key, value) | emit(key, value) | emit(value,
key)’
    # Iterate-Reducer
    iterateReduce1 = iterateMap1.groupByKey().flatMap(
      lambda adjacencyList: singleSortingReduce( # increments
NewPairCounter, within the boolean format, if a new pair has to be
iterated on
          adjacencyList)).sortByKey() # 'all the adjacent nodes will be
grouped together for each node':  # sortByKey() is a stable
transformation that’s going to calculate a part in order to sort

    # CCF-dedup Map Reduce job
    deduplicatedEdges1 = iterateReduce1.distinct() # 'just deduplicates
the output of the CCF-Iterate job'

    # Replace edges with the version without the doubles
    singleSortingData = deduplicatedEdges1 # 'In later iterations, the
input is the output of CCF-Dedup from the previous iteration'
    # 'If the counter is 0 at the end of the job, it means that we found
all the components'
    NewPair1 = bool(NewPairCounter1.value) # check whether a pair had to
be created or not
```

```
NewPairCounter2 = sc.accumulator(0) # 'global NewPair counter'
NewPair2 = True
secondarySortingData = optimized_data # for comparison within databricks
notebook
```

```
# check CCF-Iterate stop criterion
while NewPair2 == True:
    # Iterate

    # Stop criterion
    NewPair2 = False
    # accumulator's value to be initialized to 0 to be in a Boolean
format
    NewPairCounter2.value = 0

    # CCF-Iterate

    # Map to create a symmetry of each tuple so both elements can be
grouped by key
    iterateMap2 = secondarySortingData.flatMap(
        lambda pair: ( # 'for each edge <a; b >, mapper emits both < a;
b >, and < b; a > pairs'
        pair, pair[ # 'so that a should be in the adjacency list of b
and vice versa'
          ::-1])) # 'map(key, value) | emit(key, value) | emit(value,
key)'
    # Iterate-Reducer
    iterateReduce2 = iterateMap2.groupByKey().flatMap(
      lambda adjacencyList: secondarySortingReduce( # increments
NewPairCounter if a new pair has to be iterated on
          adjacencyList)).sortByKey() # 'all the adjacent nodes will be
grouped together for each node'

    # CCF-dedup Map Reduce job
    deduplicatedEdges2 = iterateReduce2.distinct() # 'just deduplicates
the output of the CCF-Iterate job'

    # Replace edges with the version without the doubles
    secondarySortingData = deduplicatedEdges2 # 'In later iterations,
the input is the output of CCF-Dedup from the previous iteration'
    # 'If the counter is 0 at the end of the job, it means that we found
all the components'
    NewPair2 = bool(NewPairCounter2.value) # check whether a pair had to
be created or not
```

The objective is to represent in a mapping table (because CCF is only an intermediate step to identify relationships in a pipeline): "*We want to obtain [...] the mapping from each node in the graph to its corresponding componentID [...]. The module should output a mapping table from each node in the graph to the smallest node id in its corresponding connected component.*" Therefore, we're going to invert the tuples, since the output of CCF-Iterate returns them in the opposite direction to that sought, as can be seen in the diagram of the 4th iterations.

```
mappingTable = singleSortingData.map(
        lambda pair: ( # for each edge <b; a >, reverses CCF-Iterate by
emitting < a; b >
        pair[ # 'so that a should be in the adjacency list of b and vice
versa'
          ::-1])) # 'map(key, value) | emit(key, value) | emit(value,
key)'
```

```
mappingTable.sortByKey().take(10)
```

3. Experimental analysis

We turn Spark readable objects into MapReduce readable tuples so that "*the input which is given as (key,value) pairs is split up among multiple machines to be worked on in parallel*".

3.1. RDD

3.1.1. Single sort

3.1.1.1. Sort By Key

```
while NewPair1 == True:

    NewPair1 = False

    NewPairCounter1.value = 0

    iterateMap1 = singleSortingData.flatMap(
        lambda pair:
        pair, pair[
          ::-1]))

    iterateReduce1 = iterateMap1.groupByKey().flatMap(
      lambda adjacencyList: singleSortingReduce(
          adjacencyList)).sortByKey()


    deduplicatedEdges1 = iterateReduce1.distinct()


    singleSortingData = deduplicatedEdges1


    NewPair1 = bool(NewPairCounter1.value)
```

Command took 8.00 minutes.

3.1.1.2. Reparted and sorted within partition

```
singleSortingData.getNumPartitions()
```

```
NewPairCounter1P = sc.accumulator(0)
NewPair1P = True
singleSortingDataP = optimized_data
```

```
while NewPair1P == True:

    NewPair1P = False
    NewPairCounter1P.value = 0

    iterateMap1P = singleSortingDataP.flatMap(
        lambda pair: (
        pair, pair[
          ::-1]))
    iterateReduce1 = iterateMap1P.groupByKey().flatMap(
      lambda adjacencyList: singleSortingReduce(
          adjacencyList)).repartitionAndSortWithinPartitions(3)
    deduplicatedEdges1P = iterateReduce1.distinct()
    singleSortingDataP = deduplicatedEdges1P
    NewPair1P = bool(NewPairCounter1P.value)
```

Command took 0.15 seconds

3.1.2. Secondary sort

3.1.2.1. Sort by key

```
while NewPair2 == True:
    NewPair2 = False
    NewPairCounter2.value = 0
    iterateMap2 = secondarySortingData.flatMap(
        lambda pair: (pair, pair[::-1]))
    iterateReduce2 = iterateMap2.groupByKey().flatMap(
      lambda adjacencyList: secondarySortingReduce(adjacencyList)).sortByKey()
    deduplicatedEdges2 = iterateReduce2.distinct()

    secondarySortingData = deduplicatedEdges2
```

```
    NewPair2 = bool(NewPairCounter2.value)
```

Command took 1.04 minutes

3.1.2.2. Reparted and sorted within partition optimized

```
while NewPair2P == True:
    NewPair2P = False
    NewPairCounter2P.value = 0
    iterateMap2P = secondarySortingDataP.flatMap(
        lambda pair: (
        pair, pair
          ::-1]))
    iterateReduce2P = iterateMap2P.groupByKey().flatMap(
      lambda adjacencyList: secondarySortingReduce(
          adjacencyList)).repartitionAndSortWithinPartitions(3)
    deduplicatedEdges2P = iterateReduce2P.distinct()
    secondarySortingDataP = deduplicatedEdges2P
    NewPair2P = bool(NewPairCounter2P.value)
```

Command took 0.18 seconds

3.2. Sanity check

```
singleSortingData.take(10)

results = map(lambda e: e[::-1], singleSortingData.collect())

mappingTable = singleSortingData.map(
        lambda pair: ( # for each edge <b; a >, reverses CCF-Iterate emitting
< a; b >
        pair[ # 'so that a should be in the adjacency list of b and vice
versa'
          ::-1])) # 'map(key, value) | emit(key, value) | emit(value, key)'
```

Command took 0.02 seconds

3.3. Dataframes

```
DFNewPairCounter1 = sc.accumulator(0)
DFNewPair1 = True

DFNewPairCounter2 = sc.accumulator(0)
DFNewPair2 = True
```

```
DFsingleSortingData = optimized_data.toDF()
DFsecondarySortingData = optimized_data.toDF()
```

```
while DFNewPair1 == True:
    DFNewPair1 = False
    DFNewPairCounter1.value = 0
    DFiterateMap1 = DFsingleSortingData.explode(
        lambda pair: (
        pair, pair[
          ::-1]))
    DFiterateReduce1 = DFiterateMap1.groupByKey().flatMap(
      lambda adjacencyList: DFsingleSortingReduce(
          adjacencyList)).sortByKey()
    DFdeduplicatedEdges1 = DFiterateReduce1.distinct()
    DFsingleSortingData = DFdeduplicatedEdges1
    DFNewPair1 = bool(DFNewPairCounter1.value)
```

Command took 0.04 seconds (and 0.02 s when sortByKey is replaced with repartitionAndSortWithinPartitions(3))

```
while DFNewPair2 == True:
    DFNewPair2 = False
    DFNewPairCounter2.value = 0
    DFiterateMap2 = DFsecondarySortingData. explode(
        lambda pair: (
        pair, pair[
          ::-1]))
    DFiterateReduce2 = DFiterateMap2.groupByKey().flatMap(
      lambda adjacencyList: secondarySortingReduce(adjacencyList)).sortByKey()
    DFdeduplicatedEdges2 = DFiterateReduce2.distinct()
    DFsecondarySortingData = DFdeduplicatedEdges2
```

```
    DFNewPair2 = bool(DFNewPairCounter2.value)
```

Command took 0.02 seconds (and 0.02 s when sortByKey is replaced with repartitionAndSortWithinPartitions(3))

3.4. Hadoop Cluster code

```
import string
from pyspark.rdd import RDD
from pyspark.sql import SparkSession
from pyspark import StorageLevel
import time
import sys

def main():
    """ Main get connected components script definition.
    : return: None
    """

    args = sys.argv[1:]
    print(len(args), args.__class__)
    print(args)
    #ifRunOnLocal = bool(args[0])
    ifRunOnLocal = True

    if ifRunOnLocal:
        __MASTER__ = "local[*]"
        NB_PARTITIONS = 4
    else:
        __MASTER__ = "yarn"
        #NB_PARTITIONS = args[1]
        NB_PARTITIONS = 4

    # Define sparksession
    spark = SparkSession.builder \
        .appName("IASD3 - Connected Components - MapReduce Implementation") \
        .master(__MASTER__) \
        .getOrCreate()
    """
    rdd = buildTrainRDD(spark, NB_PARTITIONS)
    print("rdd : ", rdd.__class__, rdd.count())
    print("\n")
```

```
    """
    path = "C:\\Users\\HP LUXE\\Documents\\JOBS\\FORMATION CONTINUE\\Paris " \
           "Dauphine\\Cours\\Traitement_distribuee_donnees_massives_hadoop-
Spark_Kafka\\web-Google.txt "

    rdd = spark.sparkContext.textFile(path)
    print("rdd : ", rdd.__class__, rdd.count())

    print("ccfJobLauncher running!!!")
    start_time_ccf = time.time()
    ccfRDD, nb_iterations = ccfIterateJobLauncher(rdd)
    print("ccfRDD : ", ccfRDD.__class__, ccfRDD.count())
    end_time_ccf = time.time()
    print(f"Execution time - ccfIterate              : {end_time_ccf -
start_time_ccf}")
    print(f"Execution time - ccfIterate              : {round((end_time_ccf -
start_time_ccf) / 60)} mn")
    print(f"ccfIterate converge in {nb_iterations} itérations \n")

    print("ccfRDD : ", ccfRDD.__class__, ccfRDD.count())
    ccf = ccfRDD.flatMap(lambda v: [str(v[0]) + " " + str(v[1]), str(v[1]) + "
" + str(v[1])]).distinct()
    print("\n")

    print("ccfIterate2ndSortJobLauncher running!!!")
    start_time_2ndS = time.time()
    ccf2ndSRDD, nb_iterations = ccfIterate2ndSortJobLauncher(rdd)
    end_time_2ndS = time.time()
    print(f"Execution time - ccfIterate2ndSort : {end_time_2ndS -
start_time_2ndS}")
    print(f"Execution time - ccfIterate2ndSort in mn : {round((end_time_2ndS -
end_time_2ndS) / 60)}")
    print("ccfIterate2ndSort converge with ", nb_iterations, " iterations! \n")
    ccf2nd = ccf2ndSRDD.flatMap(lambda v: [str(v[0]) + " " + str(v[1]),
str(v[1]) + " " + str(v[1])]).distinct()

    spark.stop()

    return None

"""------------------------ Utils Functions RDD ----------------------------
-- """
def rddMapper(rdd: RDD):
    return rdd.flatMap(lambda item: [(emitter(item[0], item[1])),
(emitter(item[1], item[0]))]) #.sortByKey()
```

```
def emitter(x, y):
   return int(x), int(y)

def ccfIterateJobLauncher(rdd: RDD):
   nb_iterations = 0
   neighbors = rddMapper(rdd)
   while True:
       nb_iterations = nb_iterations + 1
       """
         reduce(key,< iterable > values)
         min = key
         for each (value in values)
           if(value < min)
           min   value
           valueList.add(value)
       """
       m = neighbors.reduceByKey(lambda k, v: min(k, v)).map(lambda v:
(int(v[0]), min(int(v[0]), int(v[1]))))
       graphEdRDD = neighbors.map(lambda e: e if int(e[0]) < int(e[1]) else
None).filter(lambda k: k is not None)
       newEdgesRDD = graphEdRDD.join(m).map(lambda e: (e[1][0],
e[1][1])).distinct()
       if ifStopLoop(newEdgesRDD, neighbors):
           break
       else:
           neighbors = newEdgesRDD.flatMap(lambda item: [(emitter(item[0],
item[1])), (emitter(item[1], item[0]))]).persist(StorageLevel.MEMORY_AND_DISK)
   return newEdgesRDD, nb_iterations

def ccfIterate2ndSortJobLauncher(neighbors: RDD):
   nb_iterations = 0
   while True:
       nb_iterations = nb_iterations + 1
       neig = neighbors.reduceByKey(lambda a, b: min(a, b)).map(lambda v:
(int(v[0]), min(int(v[0]), int(v[1]))))
       lgEdgesRDD = neighbors.map(lambda e: e if int(e[0]) < int(e[1]) else
None).filter(lambda k: k is not None)
       newEdgesRDD = lgEdgesRDD.join(neig).map(lambda e: (e[1][0],
e[1][1])).distinct()

       # 2nd sort and deduplicate
       secondSRDD = newEdgesRDD.map(lambda e: (emitter(e[0], e[1])) if
int(e[1]) <= int(e[0]) else (emitter(e[1], e[0])))
       neigs = secondSRDD.reduceByKey(lambda a, b: min(a, b)).map(lambda v:
```

```
(int(v[0]), min(int(v[0]), int(v[1]))))
        secondSNewEdges = secondSRDD.join(neigs).map(lambda e: emitter(e[1][0],
e[1][1]) if (int(e[1][0]) != int(e[1][1])) else emitter(e[0],
e[1][0])).distinct()
        secondSNewEdgesRDD = secondSRDD.flatMap(lambda e: [(emitter(e[0],
e[1])), (emitter(e[1], e[0]))])
        if ifStopLoop(secondSNewEdgesRDD, neighbors):
            break
        else:
            neighbors = secondSNewEdges.flatMap(lambda e: [(emitter(e[0],
e[1])), (emitter(e[1], e[0]))]).persist(StorageLevel.MEMORY_AND_DISK)
    return secondSRDD, nb_iterations


def ifStopLoop(newEdgesRDD: RDD, neighbors: RDD):
    if newEdgesRDD.subtract(neighbors).isEmpty() and
neighbors.subtract(newEdgesRDD).isEmpty():
        print("ifBreak == TRUE")
        ifBreak = True
    else:
        ifBreak = False
    return ifBreak

if __name__ == '__main__':
    main()
```

4. Comments on the results

The exercise was not so much to count the graphs but to follow the steps of the article: in fact, the article is a method of counting which replicates exactly the steps of map reduce.

For example, the reading of the nodes corresponds to the way in which the RDDs are created, the transformation of the adjacency lists corresponds to that of Map, the intermediate sort of the second algorithm to the merge sort (cf algorithmics course) and to the 'distinct' method abstraction within Spark, etc.

Counting nodes is a simple exercise to implement Map Reduce steps, and comment on the difference between Hadoop and Spark (hence the 2 algorithms).

We preferred using a global accumulator native to spark rather than a local counter that would be returned.

It seems that pre-processing in full makes it possible to maintain a complexity of O(n) rather than $O(n^2)$ (object of comparison of the CCF).

Otherwise, it could slow down even more during the secondary sort (it seems to me that the complexity would go to $O(2n^2)$) so the effect would theoretically be contrary to what we hope.

The article suggests using an iterator (each of which has its own implementation of "next").

In explanation, the article specifies "*The space complexity of this approach is O(N) where N is the size of largest connected component as we store the values in a list in the reducer. In order to improve the space complexity further, we implemented another version of CCF-Iterate, presented in Figure 3. A secondary sort approach can be used to pass the values to the reducer in a sorted way with custom partitioning. First value will be the minValue. We will just iterate over the values once to emit the necessary values.*"

Yield is particularly suitable for the second algorithm because as you can read at Understanding the Python Yield Statement[5], "*They're also useful in the same cases where list comprehensions are used, with an added benefit: you can create them without building and holding the entire object in memory before iteration. In other words, you'll have no memory penalty when you use generator expressions.*"

In addition to being adapted to the parallelization of Spark (thanks to the asynchronous concurrency of tasks[6]) and its lazy evaluation, "*yield*" also optimizes (just like Spark does) the path of the iterator the pseudocode refers to as "*unlike lists, lazy iterators do not store their contents in memory*"[7].

The presence of the "*.next*" in the pseudo code (which would not be reduced simply to the use of an iterator as I thought previously), it makes it possible to indicate with precision, that it is iterated upon by a native method "*next()*" to be sorted.

But whatever the implementation, it should be a priori ordered (without using for example a loop as in the other version of the algo), with a spatial complexity linear in the worst case[8].

However, this is sufficient to be theoretically at least as efficient as the previous default algorithm according to the article ("*The space complexity of this approach is O(N) where N is the size of largest connected component as we store the values in a list in the reducer*"), doubly optimized not only by the lazy evaluation (acceleration which we assume to be of the same order of magnitude as the ratio between Hadoop and Spark) suggested by the article ("values can be iterated just once without loading all of them into

[5] https://realpython.com/introduction-to-python-generators/#understanding-the-python-yield-statement
[6] *https://zestedesavoir.com/articles/1568/decouvrons-la-programmation-asynchrone-en-python/#2-concurrence- and-parallelism*
[7] https://realpython.com/introduction-to-python-generators/#using-generators
[8] https:/ /en.wikipedia.org/wiki/Algorithm_of_sorting#Comparison_of_algorithms

memory") of the generator but also by parallelization (spatial complexity in O(log(n)) and temporal complexity in O(n.log(n)) if we are in the best case with all the elements almost sorted).

In the 1st map of the algo, a groupBy is not justified before the flatMap.

The goal of the article is by no means to count the nodes but rather to represent them in a mapping table (which makes sense because the introduction of the article explains that CCF is only an intermediate step for identify relationships in a pipeline, which can better leverage node IDs than a count): "*We want to obtain […] the mapping from each node in the graph to its corresponding componentID [...]. The module should output a mapping table from each node in the graph to the smallest node id in its corresponding connected compone"t.*"

Spark RDD APIs (Resilient Distributed Datasets) is Read-only partition collection of records. RDD is the fundamental data structure of Spark.

Unfortunately, it's not optimized by default, hence some actions take too much time on Databricks.

RDDs contains the collection of records which are partitioned. The basic unit of parallelism in an RDD is called partition. Each partition is one logical division of data which is immutable and created through some transformation on existing partitions. Immutability helps to achieve consistency in computations.

No inbuilt optimization engine is available in RDD. When working with structured data, RDDs cannot take advantages of sparks advance optimizers.

You can use RDDs When you want low-level transformation and actions on your data set to optimize computations.

We moved from RDD to DataFrame (If RDD is in tabular format) by toDF() method but we had to redesign the code to do so for tests on the cluster.

DataFrame in Spark allows higher-level abstraction.

RDDs take 1000 to 2000 times more time than Dataframes with billions of nodes.

Optimization takes place using catalyst optimizer. Dataframes use catalyst tree transformation framework in four phases: a) Analyzing a logical plan to resolve references. b) Logical plan optimization. c) Physical planning. d) Code generation to compile parts of the query to Java bytecode so we didn't need to do the low level optimization ourselves.

We can compare the 1st algorithm that takes by default longer than the 2nd one takes.

When optimized at the RDD level, the first algorithm still is more efficient than the 2nd because this optimization basically does not only the secondary sort in a logical way, but without sampling action.

Indeed, sortByKey, takes an extra action of sampling data first, to foundariesies for range.

On the contrary, repartitionAndSortWithinPartitions can find the maximum for each key by means of a single pass on each partition, without charging it in the RAM.

As repartitionAndSortWithPartitions and secondary both do the sorting, using them both together is redundant, hence the extra seconds.

With dataframes the 2nd algorithm is twice as fast as the 1st with a sort By Key but both take the same time when optimized within partitions.

We can conclude that optimisations within partitions and dataframes are the fastest strategy for large datasets.

5. Appendix

5.1. Code[9]

```
# -*- coding: utf-8 -*-
"""Connected Component Finder (DF) (1).ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1XeP6poMpySCT8AqoWuZo5jk1NMlYR1Ym

# CCF: Fast and Scalable Connected Component Computation in MapReduce

"Connected Component Finder (CCF): finding all connected components in a graph"<br>
* [Guidelines](https://github.com/neohack22/IASD/blob/master/indications-
Project.pdf)
* [Article](https://www.cse.unr.edu/~hkardes/pdfs/ccf.pdf)

"We want to obtain [...] the mapping from each node in the graph to its
corresponding componentID (smallest node id in each connected component as the
identifier of that component)."<br>
"The need to distribute the work informs the design."<br>
"The module should output a mapping table from each node in the graph to the
smallest node id in its corresponding connected component."

## Imports

### Spark environment
"""

from pyspark import SparkContext, SparkConf # gives options to provide configuration
parameters passed to spark context, entry point for spark environment

from pyspark.sql import SparkSession
```

[9] https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/1459143108282575/3650636048450686/661125469820182/latest.html

```
from pyspark.rdd import RDD

#Init sparkSession

warehouseLocation = "file:${system:user.dir}/spark-warehouse"
spark = SparkSession.builder.appName("Projet IASD - Spark MapReduce") \
.config("spark.sql.warehouse.dir", warehouseLocation) \
.enableHiveSupport() \
.getOrCreate()

"""### Datasets

#### Data Scraping
"""

path = "/FileStore/tables/web_Google-1.txt"
dbutils.fs.ls(path)

dbutils.fs.ls("/FileStore/tables/") # liste fichiers et dossiers dans directory

"""## Data transformation"""

# move web_Google.txt from dbfs to local file system
dbutils.fs.cp(path, 'file:/tmp/web_Google.txt') # copie path vers file:/tmp

localPath = 'file:/tmp/web_Google.txt'

# transform dataset file into RDD
data = sc.textFile(localPath)

"""We notice that the file starts with commented lines describing the file structure
of 5105039 Edges, linking 875713 Nodes in the textual format 'FromNodeId\tToNodeId',
as a tuple.

### Preprocessing

### Map

Les 4 1ères lignes doivent être ignorées car caractérisent le schéma.<br>
Ces données semblent être saisies manuellement.
```

```
"""

optimized_data = data.filter(lambda lines: lines[0] != '#').map(lambda edge :
tuple(map(lambda node: int(node), edge.split("\t")[:2])))

optimized_data.take(10)

"""# CCF-Iterate
"CCF-Iterate job generates adjacency lists AL = (a1, a2, ..., an) for each node v"

## User-Defined Reduce Function
"The output of the map phase is put back together for each key to independently
process the values for each key in parallel"

".next()" pseudocode suggests using an iterator.<br>
Furthermore, it allows to indicate with precision, that this iterator is traversed
by a native method "next()" to be sorted.<br>
But whatever the implementation, it should be ordered (without using for example a
loop as in the single sorting version of the algo), [with a spatial complexity in
the worst case linear](https://fr
.wikipedia.org/wiki/Algorithm_of_sorting#Comparison_of_algorithms).<br>
However, this is sufficient to be theoretically at least as efficient as the
previous default algorithm according to the
[article](https://www.cse.unr.edu/~hkardes/pdfs/ccf.pdf) ("The space complexity of
this approach is O(N) where N is the size of largest connected component as we store
the values in a list in the reducer").<br>
What's more, the article specifies "The space complexity of this approach is O(N)
where N is the size of largest connected component as we store the values in a list
in the reducer. In order to improve the space complexity further, we implemented
another version of CCF-Iterate [...]. A secondary sort approach can be used to pass
the values to the reducer in a sorted way with custom partitioning. We don't need to
iterate over the values with this approach as the first value will be the minValue.
We will just iterate over the values once to emit the necessary values."<br>
Hence our implementation of the secondary sorting algorithm would be optimized
(experiments will tell us how much) not only by the lazy evaluation (acceleration we
anticipate being of the same order of magnitude as the ratio between Hadoop and
Spark) suggested by the article ( "values can be iterated just once without loading
all of them into memory") of the generator but also by parallelization (spatial
complexity in O(log n) and temporal complexity in O(n log n) if we are in the best
case with all items almost sorted).<br>
```

```
The 'yield' keyword remains particularly suitable to do so because ["They're also
useful in the same cases where list comprehensions are used, with an added benefit:
you can create them without building and holding the entire object in memory before
iteration. In other words, you'll have no memory penalty when you use generator
expressions."](https://realpython.com/introduction-to-python-
generators/#understanding-the-python-yield-statement). <br>
In addition to being adapted to the parallelization of Spark (thanks to
[asynchronous concurrency](https://zestedesavoir.com/articles/1568/decouvrons-la-
programmation-asynchrone-en-python/#2-concurrence-et -parallelism) of tasks) and its
[lazy evaluation](https://realpython.com/introduction-to-python-generators/#using-
generators), "yield" also optimizes (according to the same principle as Spark) the
traversing the iterator to which the pseudocode refers because "unlike lists, lazy
iterators do not store their contents in memory".

### Single sorting Reducer (Figure 2 multiple passes)
"""

# break the adjency list into pairs
def singleSortingReduce( # 'start with the initial edge list to construct the first
degree neighborhood of each node'
    edgesList): # 'For the first iteration, this job takes the initial edge list as
input'
  # abstract 'key and value pairs' to be reduced by UDF
  key, iterable_values = edgesList # 'key, < iterable > values'
  # assume that the key's node is the minimal value of its adjacency list
  min = key # 'min ← key'

  # 'go over all the values to find the minValue and store all the values in a list'

  # initialize an non iterated adjacency list to search for the min valued edge from
  valueList = [] # 'If multiple passes are needed, the values should be stored in a
list'
  # copy each node's adjacency list to look for the min node inside
  for value in iterable_values: # 'for each (value ∈ values)'
    # search for the minimum valued edge
    if value < min: # 'if(value < min)'
      # update min value
      min = value # 'min ← value'
    # add each value of the adjency list into the copy
```

```
    valueList.append(value) # 'valueList.add(value)'
    # nothing is emitted if the pair's value is not < key

  # if the root, being the key of the adjency list, is not the minimal value:
  if min < key: # 'if the node id of this node v_id is larger than the min node id
a_min in the adjacency list: if(min < key)'
    # emit a spark-optimized reduceable pair with min node as value for key as key
    yield( # 'In MapReduce, values can be iterated just once without loading all of
them into memory'
        ( # 'emit(key, min)'
            key, min)) # 'emit the < key; minV alue > pair [...] (v_id, a_min)'
    # check for each values in the adjacency list to be compared to its confirmed
min
    for value in valueList: # 'where a_i ∈ AL': 'for each (value ∈ valueList)'
      # compare this min with the value to prepare a symetric tuple so that the
right node of the pair will be mapped as well
      if min != value: # 'where [...] a_i <> a_min: if(min <> value)'
        # make sure you the stop criterion is not activated, as long as it's value's
false, as long as the adjencency list root is not starting with the minimal value
        NewPairCounter1.add(1) # 'increase the global NewPair counter by 1:
Counter.NewPair.increment(1)'
        # emit a second spark-optimized distribution from the adjacency list, with
min as the MapReduce value so that it can be mapped for CCF again
        yield( # 'In MapReduce, values can be iterated just once without loading all
of them into memory'
            ( # 'emit(value, min) [...] for all other values as < value; minValue >'
                value, min)) # 'creates [...] a pair for each (a_i, a_min)' w/ node
value, from the adjacency list, to be mapped as the minimal value MapReduce key, so
that the minimal value can be mapped for CCF again
      # 'if v_id is smaller than a_min, we do not emit any pair [...]  and there is
no need for further iterations'
    # 'If the minValue is larger than key, we do not emit anything'

"""### Secondary sorting Reducer (Figure 3 Adjusting memory utilization)"""

# break the adjency list into pairs
def secondarySortingReduce( # 'start with the initial edge list to construct the
first degree neighborhood of each node'
    edges_list): # 'For the first iteration, this job takes the initial edge list as
input'
```

```
  # abstract adjacency list to be reduced by UDF
  key, values = edges_list  # 'key, < iterable > values'
  # Add minimal value to key to be considered in sortByKey
  minValue = sorted(values)[0] # minValue ← values.next() #fait en mémoire, ce qui
ne faut pas faire
  # copy each node's adjacency list to look for the min node inside
  if minValue < key: # if(minValue < key)
    # emit a spark-optimized reduceable pair with min node as value for key as key
    yield( # 'In MapReduce, values can be iterated just once without loading all of
them into memory'
        (
            key, minValue)) # emit(key, minValue)
    # check for each values in the adjacency list to be compared to its confirmed
min
    for value in valueList: # 'where a_i ∈ AL': 'for each (value ∈ values)'
      # make sure you the stop criterion is not activated, as long as it's value's
false, as long as the adjencency list root is not starting with the minimal value
      NewPairCounter2.add(1) # 'increase the global NewPair counter by 1:
Counter.NewPair.increment(1)'
      yield( # 'In MapReduce, values can be iterated just once without loading all
of them into memory'
          ( # 'emit(value, min) [...] for all other values as < value; minValue >'
              value, minValue)) # emit(value, minV alue) w/ node value, from the
adjacency list, to be mapped as the minimal value MapReduce key, so that the minimal
value can be mapped for CCF again
      # 'if v_id is smaller than a_min, we do not emit any pair [...]  and there is
no need for further iterations'
    # 'If the minValue is larger than key, we do not emit anything'

"""### CCF-Dedup

CCF-Dedup 'just deduplicates the output of the CCF-Iterate job' so we use
RDD.distinct() to do so.<br>
Indeed [its Pyspark
implementation](https://spark.apache.org/docs/latest/api/python/_modules/pyspark/rdd
.html#RDD.distinct), as described below, matches the article's pseudocode:

    def distinct(self, numPartitions=None):
        '''
        Return a new RDD containing the distinct elements in this RDD.
```

```
        Examples
        --------
        >>> sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())
        [1, 2, 3]
        '''
        return self.map(lambda x: (x, None)) \ # map(key, value) | temp.entity1 ← 
key | temp.entity2 ← value | emit(temp, null)
                   .reduceByKey(lambda x, _: x, numPartitions) \ # reduce(key, < 
iterable > values)
                   .map(lambda x: x[0]) # emit(key.entity1, key.entity2)

## Map-reduce job
"""

# Spark Context Accumulator initialized outside the loop
NewPairCounter1 = sc.accumulator(0) # 'global NewPair counter'
NewPair1 = True
singleSortingData = optimized_data

"""We turn Spark readable objects into  MapReduce readable tuples so that "the input 
which is given as (key,value) pairs is split up among multiple machines to be worked 
on in parallel"
"""

# check CCF-Iterate stop criterion
while NewPair1 == True: # True: #to b completed w/ the stop criterion
    # Iterate
    # nodes_number = nodes_number + 1 # to be used for sanity check with article toy 
data
    # Stop criterion
    NewPair1 = False
    # accumulator's value to be initialized to 0 to maintain boolean range
    NewPairCounter1.value = 0

    # CCF-Iterate

    # Map to create a symetry of each tuple so both elements can be grouped by key
    iterateMap1 = singleSortingData.flatMap(
        lambda pair: ( # 'for each edge <a; b >, mapper emits both < a; b >, and <
```

```
b; a > pairs'
        pair, pair[ # 'so that a should be in the adjacency list of b and vice
versa'
          ::-1])) # 'map(key, value) | emit(key, value) | emit(value, key)'
    # Iterate-Reducer
    iterateReduce1 = iterateMap1.groupByKey().flatMap(
      lambda adjacencyList: singleSortingReduce( # increments NewPairCounter if a
new pair has to be iterated on
          adjacencyList)).sortByKey() # 'all the adjacent nodes will be grouped
together for each node' #plus stable # sortByKey() est une transformation-action qui
va calculer une partie en fonction de laquelle il va faire un tri

    # CCF-dedup Map Reduce job
    deduplicatedEdges1 = iterateReduce1.distinct() # 'just deduplicates the output
of the CCF-Iterate job'

    # Replace edges with the double removed version
    singleSortingData = deduplicatedEdges1 # 'In later iterations, the input is the
output of CCF-Dedup from the previous iteration'
    # 'If the counter is 0 at the end of the job, it means that we found all the
components'
    NewPair1 = bool(NewPairCounter1.value) # check wether a pair had to be created
or not

NewPairCounter2 = sc.accumulator(0) # 'global NewPair counter'
NewPair2 = True
secondarySortingData = optimized_data # for comparison within databricks notebook

# check CCF-Iterate stop criterion
while NewPair2 == True: # True: #to b completed w/ the stop criterion
    # Iterate
    # nodes_number = nodes_number + 1 # to be used for sanity check with article toy
data
    # Stop criterion
    NewPair2 = False
    # accumulator's value to be initialized to 0 to maintain boolean range
    NewPairCounter2.value = 0

    # CCF-Iterate
```

```
    # Map to create a symetry of each tuple so both elements can be grouped by key
    iterateMap2 = secondarySortingData.flatMap(
        lambda pair: ( # 'for each edge <a; b >, mapper emits both < a; b >, and < b; a > pairs'
        pair, pair[ # 'so that a should be in the adjacency list of b and vice versa'
          ::-1])) # 'map(key, value) | emit(key, value) | emit(value, key)'
    # Iterate-Reducer
    iterateReduce2 = iterateMap2.groupByKey().flatMap(
      lambda adjacencyList: secondarySortingReduce( # increments NewPairCounter if a new pair has to be iterated on
          adjacencyList)).sortByKey() # 'all the adjacent nodes will be grouped together for each node'

    # CCF-dedup Map Reduce job
    deduplicatedEdges2 = iterateReduce2.distinct() # 'just deduplicates the output of the CCF-Iterate job'

    # Replace edges with the double removed version
    secondarySortingData = deduplicatedEdges2 # 'In later iterations, the input is the output of CCF-Dedup from the previous iteration'
    # 'If the counter is 0 at the end of the job, it means that we found all the components'
    NewPair2 = bool(NewPairCounter2.value) # check wether a pair had to be created or not

singleSortingData.take(10)

"""The objective is to represent in a mapping table (because CCF is only an intermediate step to identify relationships in a pipeline): "We want to obtain [...] the mapping from each node in the graph to its corresponding componentID [...]. The module should output a mapping table from each node in the graph to the smallest node id in its corresponding connected component."<br>
Therefore, we're going to invert the tuples, since the output of CCF-Iterate returns them in the opposite direction to that sought, as can be seen in the diagram of the 4th iterations.
"""

results = map(lambda e: e[::-1], singleSortingData.collect())
```

```
mappingTable = singleSortingData.map(
        lambda pair: ( # for each edge <b; a >, reverses CCF-Iterate emitting < a; b
>
        pair[ # 'so that a should be in the adjacency list of b and vice versa'
          ::-1])) # 'map(key, value) | emit(key, value) | emit(value, key)'

mappingTable.take(10)

mappingTable.sortByKey().take(10)

"""Count"""

#deduplicatedEdges.count()
#deduplicated_nodes.count()

#print(deduplicated_nodes[0])
#deduplicated_nodes.take(10)

"""## Dataframes check"""

#checkData = (1,2),(2,3),(2,4),(4,5),(6,7),(7,8)
#checkData = sc.parallelize(checkData)
#creer rdd spark

# Spark Context Accumulator initialized outside the loop
#NewPairCounter_check = sc.accumulator(0) # 'global NewPair counter'
#NewPair_check = True
#singleSortingData = optimized_data

DFNewPairCounter1 = sc.accumulator(0) # 'global NewPair counter'
DFNewPair1 = True

DFNewPairCounter2 = sc.accumulator(0) # 'global NewPair counter'
DFNewPair2 = True

#RDDsingleSortingData = optimized_data.rdd
DFsingleSortingData = optimized_data.toDF()
DFsecondarySortingData = optimized_data.toDF()

DFsingleSortingData
```

```
# check CCF-Iterate stop criterion
while DFNewPair1 == True: # True: #to b completed w/ the stop criterion
    # Iterate
    # nodes_number = nodes_number + 1 # to be used for sanity check with article toy
data
    # Stop criterion
    DFNewPair1 = False
    # accumulator's value to be initialized to 0 to maintain boolean range
    DFNewPairCounter1.value = 0

    # CCF-Iterate

    # Map to create a symetry of each tuple so both elements can be grouped by key
    DFiterateMap1 = DFsingleSortingData.flatMap(
        lambda pair: ( # 'for each edge <a; b >, mapper emits both < a; b >, and <
b; a > pairs'
        pair, pair[ # 'so that a should be in the adjacency list of b and vice
versa'
          ::-1])) # 'map(key, value) | emit(key, value) | emit(value, key)'
    # Iterate-Reducer
    DFiterateReduce1 = DFiterateMap1.groupByKey().flatMap(
      lambda adjacencyList: DFsingleSortingReduce( # increments NewPairCounter if a
new pair has to be iterated on
          adjacencyList)).sortByKey() # 'all the adjacent nodes will be grouped
together for each node' #plus stable # sortByKey() est une transformation-action qui
va calculer une partie en fonction de laquelle il va faire un tri

    # CCF-dedup Map Reduce job
    DFdeduplicatedEdges1 = DFiterateReduce1.distinct() # 'just deduplicates the
output of the CCF-Iterate job'

    # Replace edges with the double removed version
    DFsingleSortingData = DFdeduplicatedEdges1 # 'In later iterations, the input is
the output of CCF-Dedup from the previous iteration'
    # 'If the counter is 0 at the end of the job, it means that we found all the
components'
    DFNewPair1 = bool(DFNewPairCounter1.value) # check wether a pair had to be
created or not
```

```
# check CCF-Iterate stop criterion
while DFNewPair2 == True: # True: #to b completed w/ the stop criterion
    # Iterate
    # nodes_number = nodes_number + 1 # to be used for sanity check with article toy
data
    # Stop criterion
    DFNewPair2 = False
    # accumulator's value to be initialized to 0 to maintain boolean range
    DFNewPairCounter2.value = 0

    # CCF-Iterate

    # Map to create a symetry of each tuple so both elements can be grouped by key
    DFiterateMap2 = DFsecondarySortingData.flatMap(
        lambda pair: ( # 'for each edge <a; b >, mapper emits both < a; b >, and <
b; a > pairs'
        pair, pair[ # 'so that a should be in the adjacency list of b and vice
versa'
          ::-1])) # 'map(key, value) | emit(key, value) | emit(value, key)'
    # Iterate-Reducer
    DFiterateReduce2 = DFiterateMap2.groupByKey().flatMap(
      lambda adjacencyList: secondarySortingReduce( # increments NewPairCounter if a
new pair has to be iterated on
          adjacencyList)).sortByKey() # 'all the adjacent nodes will be grouped
together for each node'

    # CCF-dedup Map Reduce job
    DFdeduplicatedEdges2 = DFiterateReduce2.distinct() # 'just deduplicates the
output of the CCF-Iterate job'

    # Replace edges with the double removed version
    DFsecondarySortingData = DFdeduplicatedEdges2 # 'In later iterations, the input
is the output of CCF-Dedup from the previous iteration'
    # 'If the counter is 0 at the end of the job, it means that we found all the
components'
    DFNewPair2 = bool(DFNewPairCounter2.value) # check wether a pair had to be
created or not

# end of tests
```

5.2. Jupyter notebook local implementation initial draft

Etape 1 : IMPORTATION du DATASET et DATA PREP (Preprocessing Dataset)

Entrée [1]:
```
import pandas as pd
```

Entrée [2]:
```
# Il faut d'abord télécharger le dataset web-google.txt
# et mettre son chemin d'accès dans read_csv() ci-dessous

df = pd.read_csv('/Users/a33679/Documents/!0-SparkProject/web-Google.txt')

# Nb de Noeuds : 875.713

# Nb d'Arêtes (Edges) : 5.105.039
# = Nb de Lignes dans le fichier de data
# = Nb de Paires "noeud de départ" -> "noeud d'arrivée")

# df : nom de la variable où l'on va stocker toutes les données
```

Entrée [3]:
```
df
```

Out[3]:

	# Directed graph (each unordered pair of nodes is saved once): web-Google.txt
# Webgraph from the Google programming contest	2002.0
# Nodes: 875713 Edges: 5105039	NaN
# FromNodeId\tToNodeId	NaN
0\t11342	NaN
0\t824020	NaN
...	...
916425\t547323	NaN
916425\t604833	NaN
916425\t632916	NaN
916425\t637936	NaN
916425\t837379	NaN

5105042 rows × 1 columns

Entrée [4]:
```
df=df[3:]

# on ne prend pas en compte les 3 premières lignes de commentaires qui ne sont pas utiles
# la 3è ligne n'est pas incluse dans le DataSet que l'on retient, la 3è ligne comprend le titre des colonnes

# Contenu des Premières Lignes du fichier Data, avant traitement
#1) x# Directed graph (each unordered pair of nodes is saved once): web-Google.txt
#2) # Webgraph from the Google programming contest, 2002
#3) # Nodes: 875713 Edges: 5105039
#4) # FromNodeId⟶ToNodeId
```

Entrée [5]:
```
df
```

Out[5]:

	# Directed graph (each unordered pair of nodes is saved once): web-Google.txt
0\t11342	NaN
0\t824020	NaN
0\t867923	NaN
0\t891835	NaN
11342\t0	NaN
...	...
916425\t547323	NaN
916425\t604833	NaN
916425\t632916	NaN
916425\t637936	NaN
916425\t837379	NaN

5105039 rows × 1 columns

Entrée [6]:
```
df.index

# afficher les index, les id de chaque ligne
# \t : désigne dans le fichier txt la séparation entre le noeud de départ et le noeud d'arrivée
```

Out[6]:
```
Index(['0\t11342', '0\t824020', '0\t867923', '0\t891835', '11342\t0',
       '11342\t27469', '11342\t38716', '11342\t309564', '11342\t322178',
       '11342\t387543',
       ...
       '916425\t218829', '916425\t229195', '916425\t367498', '916425\t379452',
       '916425\t521152', '916425\t547323', '916425\t604833', '916425\t632916',
       '916425\t637936', '916425\t837379'],
      dtype='object', length=5105039)
```

Entrée [7]:
```
datasetEntier=list()

# création d'une variable "datasetEntier" vide que l'on utilisera après
```

Entrée [8]:
```
datasetEntier
```

Out[8]:
```
[]
```

Entrée [9]:
```
len(df)
```

Out[9]:
```
5105039
```

Entrée [10]:
```
range(len(df))
```

Out[10]:
```
range(0, 5105039)
```

Entrée [11]:
```
for i in range(len(df)):
    datasetEntier.append(df.index[i])


# on s'est rendu compte plus haut que les données ont été stockées dans les index
# on va sortir les index mais en tant que datasetEntier
# ajout dans dataserEntier les index les uns après les autres, de la ligne i à la dernière ligne
```

Entrée [12]:
```
datasetEntier

# avant on affichait les Index
# maintenant on affiche les Data
# ce qui correspondait à Index dans "df", correspond maintenant aux Data dans datasetEntier
```

Out[12]:
```
['0\t11342',
 '0\t824020',
 '0\t867923',
 '0\t891835',
 '11342\t0',
 '11342\t27469',
 '11342\t38716',
 '11342\t309564',
 '11342\t322178',
 '11342\t387543',
 '11342\t427436',
 '11342\t538214',
 '11342\t638706',
 '11342\t645018',
 '11342\t835220',
 '11342\t856657',
 '11342\t867923',
 '11342\t891835',
 '824020\t0',
```

Entrée [13]:
```
# affichage de la 1ère ligne : noeud de départ "0", noeud d'arrivée "11342"

datasetEntier[0]
```

Out[13]:
```
'0\t11342'
```

Entrée [14]:
```
# affichage de la dernière ligne : noeud de départ "916425", noeud d'arrivée "837379"

datasetEntier[5105038]
```

Out[14]:
```
'916425\t837379'
```

Entrée [15]:
```
# Préparation du Dataset avec 2 colonnes,
# la colonne [0] contiendra le noeud de départ
# la colonne [1] contiendra le noeud de d'arrivée

datasetEntier2=list()
for i in datasetEntier:
    datasetEntier2.append(
        [int(i.split('\t')[0]),int(i.split('\t')[1])]
        )

# au départ liste vide
# 1ère ligne : '0\t11342',
# 2è ligne   : '0\t824020',
# on va spliter au niveau de ""\t'",
# 1ère ligne
# à gauche -> mettre dans colonne [0] : soit 0
# à droite -> mettre dans colonne [1] :11342
```

Entrée [16]:
```
# Affichage du Dataset après le split

datasetEntier2
```

Out[16]:
```
[[0, 11342],
 [0, 824020],
 [0, 867923],
 [0, 891835],
 [11342, 0],
 [11342, 27469],
 [11342, 38716],
 [11342, 309564],
 [11342, 322178],
 [11342, 387543],
 [11342, 427436],
 [11342, 538214],
 [11342, 638706],
 [11342, 645018],
 [11342, 835220],
 [11342, 856657],
 [11342, 867923],
```

Entrée [17]:
```
datasetEntier = datasetEntier2

# datasetEntier reprend le contenu de datasetEntier2 (dataset auxiliaire)
```

Entrée [18]:
```
# Un datasetEntier avec plus de 5 millions de ligne ne peut être traité avec un unique ordinateur.
# On choisit donc de restreindre aux 1000 premières lignes datasetEntier

datasetEntier=datasetEntier[:1000]

# 1000 est ici un exemple, du début jusqu'à 1000 exlus
# avec Spark (Databricks ou Cluster Dauphine, on pourra traitera l'intégralité des lignes )
```

Entrée [19]:
```
datasetEntier[0]
```

Out[19]:
```
[0, 11342]
```

Entrée [20]:
```
datasetEntier[999]
```

Out[20]:
```
[579432, 115712]
```

Entrée [21]:
```
# creation dataset représentant le graphe
datasetEntier
```

Out[21]:
```
[[0, 11342],
 [0, 824020],
 [0, 867923],
 [0, 891835],
 [11342, 0],
 [11342, 27469],
 [11342, 38716],
 [11342, 309564],
 [11342, 322178],
 [11342, 387543],
 [11342, 427436],
 [11342, 538214],
 [11342, 638706],
 [11342, 645018],
 [11342, 835220],
 [11342, 856657],
```

Entrée [22]:
```
# Création de 2 sousdatasets (datasetVert et datasetOrange)
# pour simuler un environnement distribué


# 1er SousDataSet de 500 lignes, de 0 à 499
datasetOrange = datasetEntier[:500]

# 2è SousDataSet de 500 lignes, de 500 à 999
datasetVert = datasetEntier[500:1000]
```

Entrée [23]:
```
datasetOrange[0]
```

Out[23]:
```
[0, 11342]
```

Entrée [24]:
```
datasetOrange[499]
```

Out[24]:
```
[173976, 778296]
```

Entrée [25]:
```
datasetVert[0]
```

Out[25]:
```
[173976, 782447]
```

Entrée [26]:
```
datasetVert[499]
```

Out[26]:
```
[579432, 115712]
```

Entrée [27]:
```
datasetOrange
```

```
 [0, 891835],
 [11342, 0],
 [11342, 27469],
 [11342, 38716],
 [11342, 309564],
 [11342, 322178],
 [11342, 387543],
 [11342, 427436],
 [11342, 538214],
 [11342, 638706],
 [11342, 645018],
 [11342, 835220],
 [11342, 856657],
 [11342, 867923],
 [11342, 891835],
 [824020, 0],
 [824020, 91807],
 [824020, 322178],
 [824020, 387543],
 [824020, 417728],
```

Entrée [28]:
```
datasetVert
```

```
 [5, 579655],
 [5, 581741],
 [5, 608321],
 [39733, 15455],
 [39733, 82098],
 [39733, 109781],
 [39733, 173876],
 [39733, 300279],
 [39733, 332494],
 [39733, 407045],
 [39733, 477876],
 [39733, 477995],
 [39733, 518248],
 [39733, 579655],
 [39733, 581741],
 [39733, 597093],
```

Etape 2 : DECLARATION DES FONCTIONS

fonction #1 : Liste des Noeuds d'un Dataset

Entrée [29]:
```
# Définir l'ensemble des noeuds d'un graphe à partir d'un dataset

def noeuds(dataset):
    noeuds = set()
    for n in range(len(dataset)):
        for l in range(2):
            noeuds.add(dataset[n][l])
    return noeuds

# 1 dataset en entrée => la liste des noeuds du graphe en sortie
# la fonction s'appelle noeuds
# en entrée : dataset
# en sortie "return" : noeuds
```

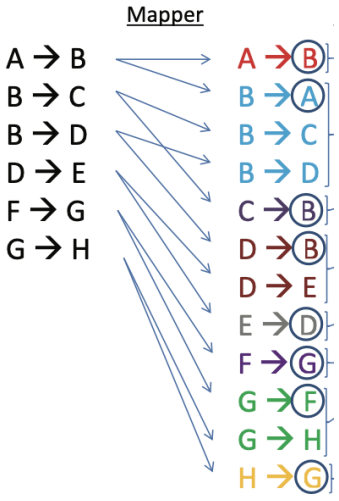
```
datasetEntier2
```

```
[[0, 11342],
 [0, 824020],
 [0, 867923],
 [0, 891835],
 [11342, 0],
 [11342, 27469],
 [11342, 38716],
 [11342, 309564],
 [11342, 322178],
 [11342, 387543],
 [11342, 427436],
 [11342, 538214],
 [11342, 638706],
 [11342, 645018],
 [11342, 835220],
 [11342, 856657],
 [11342, 867923],
 [11342, 891835],
 [824020, 0],
```

fonction #2 : Liste des Paires d'un Dataset

Entrée [ ]:

```python
# Créer à partir d'un DataSet, les couples
# dans l'ordre croissant (par exemple A vers B ; ou bien [0] vers [11342])
# et décroissant (B vers A ; ou bien [11342] vers [0])

def SymGen(dataset):
    symdataset = dataset.copy()
    for n in range(len(dataset)):
        symdataset.append([dataset[n][1],dataset[n][0]])
    return symdataset

# dans un dataset, on a des noeuds de départ et d'arrivée
# Sym : Symétrique
# Gen : Génère
# Générer le dataset symétrique
# 10 -> 20 : on va générer 10 -> 20 et 20 -> 10, couples aller et retour
# symdataset.append([dataset[n][1] : dans colonne 1 Mais aussi dataset[n][0]]) : colonne 0
# génère un dataset 2 fois plus grand
# cf. papier de recherche figure 5.
```



(a) Iteration - 1

fonction #3 : Sous-Dataset d'un Noeud Courant

(Paires liées au Noeud Courant)

Entrée [32]:
```
# Créer le sous-dataset du graphe constitué des couples contenant un noeud Courant donné

def sousdataset(dataset, noeudCourant):
    ssdataset = []
    for n in range(len(dataset)):
        if noeudCourant in set(dataset[n]):
            ssdataset.append(dataset[n])
    return ssdataset

# renvoyer toutes les flèches qui partent ou qui arrivent vers le noeud courant
# générer tous les couples qui contiennent le noeud courant
# cf. papier de recherche figure 5.

# ex  Noeud courant D : couple D -> B, D -> D, D -> E, E -> D (itération 1)
# Permet d'indiquer tous les couples ou D apparait
```

Si Nœud Courant « D »

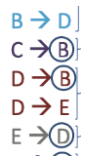
Couples ou Paires contenant D :

B → D
D → E

Alors la fonction renvoie ce sous dataset de D :

B ☐ D
D ☐ B
D ☐ E
E ☐ D

Cf. Figure 5, itération 1 dans le papier de recherche
(ignorer juste la ligne C ☐ B, qui s'intercale dans l'image)



fonction #4: Noeuds Adjancents d'un Noeud Courant

Renvoie la liste de NOEUDS d'un Noeud Courant

Entrée [32]:
```
# Définir l'ensemble des noeuds adjacents d'un Noeud Courant donné à partir de son sousdataset
def adj(dataset, noeudCourant):
    ssdataset = sousdataset(dataset, noeudCourant)
    adjacents = set()
    for n in range(len(ssdataset)):
        for i in range(2):
            if ssdataset[n][i]!=noeudCourant:
                adjacents.add(ssdataset[n][i])
    return adjacents

# une fois déterminé tous les couples qui contiennent le noeud courant, souhait d'avoir la liste des noeuds

# ex  Noeud Courant D : B, E (cf. dans l'article de recherche, figure 5, itération 1)
# Cela revient à indiquer avec qui le Noeud Courant D est en lien direct
```

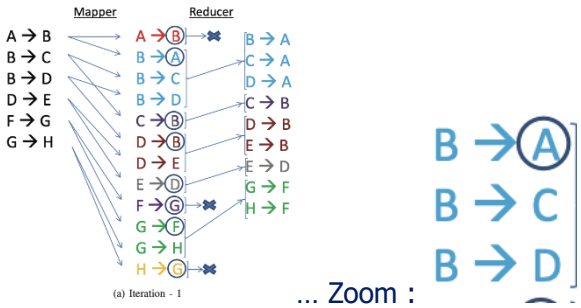
Lors du déroulement de l'algorithme présenté précédemment, il a nécessaire de déterminer les Nœuds Adjacents du Nœud Courant B :

// Pour le nœud courant B, ses voisins sont A, C, D et E
cf. schéma résultat du Reducer après itération 1

B → A
B → C
B → D
B → E

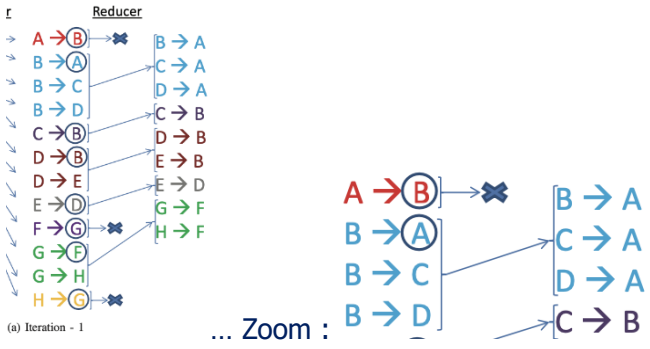
fonction #5 : Noeud Racine de l'ensemble des noeuds adjacents d'un Noeud Courant

Entrée [ ]:
```
# Définir le Noeud Racine de l'ensemble des noeuds adjacents d'un Noeud Courant donné

def rootadj(dataset, noeudCourant):
    ssdataset = sousdataset(dataset, noeudCourant)
    adjacents = set()
    for n in range(len(ssdataset)):
        for i in range(2):
            if ssdataset[n][i]!=noeudCourant:
                adjacents.add(ssdataset[n][i])
    return min(adjacents)

# rootadj : nom de la fonction
# parmi tous les noeuds adjacents au noeud courant,
# on identifie le min qui correspond à la racine

# la fonction qui va encercler, pour B : c'est A, parmi A, C et D
```



fonction #6 : Nouveau DataSet d'un Noeud Courant

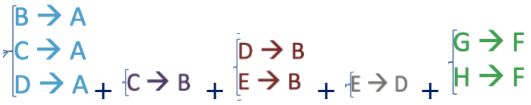
Entrée [34]:
```
# fonction qui remplit un nouveau dataset quand 'la' racine d'un Noeud Courant donné
# lui est inférieure
def newdataset(dataset, noeudCourant):
    count = 0
    ndataset = []
    racine = rootadj(dataset, noeudCourant)
    if noeudCourant>racine and len(adj(dataset, noeudCourant))!=1:
        ndataset.append([noeudCourant,racine])
        for x in adj(dataset, noeudCourant):
            if x!=racine:
                ndataset.append([x,racine])
        count = count + 1
    elif noeudCourant>racine and len(adj(dataset, noeudCourant))==1:
        ndataset.append([noeudCourant,racine])
    return ndataset, count

# Si la racine est plus petite que le noeud courant,
# dans ce cas on va créer un nouveau dataset
# ndataset.append([noeudCourant,racine]) : on va fabriquer d'autres couples

# Sinon, on ne crée pas de nouveau noeud. pas de code nécessaire.
# cf. le cas A -> B dans le papier de recherche

# len(adj(dataset, noeudCourant))!=1: : il faut qu'il y ait au moins 2 voisins

# Quand le noeud encerclé est bien racine par rapport au noeud courant, partie REDUCER
# cf. cas de B :-> A Noeud Racine, Itération 1
# On crée alors le sousdataset, avec l'ensemble des voisins de B qui vont vers A
```



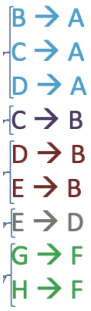
(nouveau sous DataSet en **Bleu** du Nœud Courant B)

fonction #7 : Concatenne les Nouveaux DataSet des Noeuds Courants

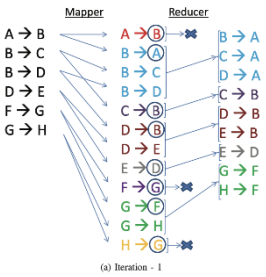
Entrée [35]:
```
def AllNewDataset(dataset):
    resultat = list()
    countNewPair = 0
    for x in noeuds(dataset):
        resultat.extend(newdataset(dataset, x)[0])
        countNewPair = newdataset(dataset, x)[1]
    return resultat

# cf. article "(a) Iteration - 1" on est après le Reducer
# pour chaque noeud courant, on va créer un nouveau Dataset
# => Tous ces noueaux Dataset sont alors concaténés dans "AllNewDataset"
# countNewPair = 0 : comptage des nouvelles paires
# extend : concaténation
# AllNewDataset : on a tous les couples générés et on sait si le compteur est à zéro


# Il va coller les sous-dataset, permet de réunir les sous dataset créés préxédement,
# Fabrique donc le dataset en sortie de l'itération 1 par exemple
```



Résultat :



Cf. article de recherche :



fonction #8 : CCF Iterate => Liste de Paires de Noeuds, Compteur Nouvelles Paires

Entrée [36]:
```
# Lister les Paires de Noeuds et Compter les Nouvelles Paires

def ccfIterate(datasetInput):
    datasetOutput = list()
    countNewPair=0
    for x in noeuds(datasetInput):
        datasetOutput.extend(newdataset(datasetInput, x)[0])
        countNewPair=countNewPair+newdataset(datasetInput, x)[1]
    return datasetOutput, countNewPair

# datasetOutput.extend(newdataset(datasetInput, x)[0]) :
# le résultat est une liste de couples de noeuds
# que l'on postionne dans la 1ère colonne (0)
#        countNewPair=countNewPair+newdataset(datasetInput, x)[1] :
# ... résultat : un compteur = nouveaux couples (nouvelles paires)
# que l'on positionne dans la 2è colonne (1)
# quand il n'y aura plus de nouvelle paire, on arrête la fonction ccfIterate

# Cette fonction va faire appel à d'autres fonctions vues précédemnent comme newdataset
```

fonction #9 : DEDUP => Retrait des Doublons

Entrée [37]:
```
# Dedoublonner une liste

def deduple(liste):
    listeSansDoublons = list()
    for element in liste:
        if element not in listeSansDoublons:
            listeSansDoublons.append(element)
    return listeSansDoublons

# Si element pas présent dans la liste, alors on l'ajoute. Sinon on ne l'ajoute pas
```

fonction #10 : Noeuds d'un Sous-Graphe

1 Sous-Graphe marqué par son Noeud Racine

Entrée [38]:
```
# Afficher les noeuds du sous-graphe associé à son noeud racine donné
# Indice du Noeud le plus petit dans un sous-graphe

def Noeuds_sousgraphe(noeudRacine):
    NoeudsSousGraphe=set()
    for i in range(len(datasetOutputFinal)):
        if datasetOutputFinal[i][1]==noeudRacine:
            NoeudsSousGraphe.add(datasetOutputFinal[i][0])
    return NoeudsSousGraphe

# A la fin du programme, identification par exemple de 3 sous-graphes
# Chaque sous-graphe contient un noeud racine
# La fonction ici donne tous les noeuds de chaque sous graphe identifié
#       if datasetOutputFinal[i][1]==noeudRacine:
# Le noeud racine est à droite
# NoeudsSousGraphe.add(datasetOutputFinal[i][0]) -> récupération du noeud de gauche
# Varible locale non créée avant : datasetOutputFinal


# Apres le travail de CCFiterate, le but est de déterminer les sous graphes séparés,
# chacun aura un noeud racine.
# Cette fdonction permet d'indiquer tous les sous graphes nommés par leurs noeuds racines
```

ETAPES d'EXECUTION

Etape 3 : ITERATION INITIALE ordi VERT (ccfIterate)

Entrée [39]:
```
# simulation d'un traitement distribué, ici ordi Vert

datasetOutputVert = ccfIterate(datasetVert)

# datasetOutputVert : variable
# ccfIterale avec en entrée : le datasetvert

# On fait ici une 1ère itération par exemple pour voir le nb de nouvelles paires si différent de Zéro
```

Entrée [40]:
```
datasetOutputVert

# affichage des couples générés après la 1ere itération
```

Out[40]:
```
([[873984, 579655],
  [902658, 39733],
  [85506, 10],
  [310676, 10],
  [393685, 10],
  [507927, 10],
  [181848, 10],
  [556796, 10],
  [407045, 39733],
  [846855, 8],
  [808449, 8],
  [331202, 8],
  [882714, 8],
  [636166, 8],
  [704230, 8],
  [333064, 8],
  [776489, 8],
  [807179, 8],
  [55948, 8],
```

Entrée [54]:
```
countNewPair = ccfIterate(datasetVert)[1]

# nb de nouvelles paires créées en sortir
```

Entrée [55]:
```
countNewPair

# Tant que le compteur est différent de Zéro, je refais des ccfIterate
# Il faut à nouveau itérer
# Ici 69 dans cet exemple => on continue
```

Out[55]: 69

Etape 5 : ITERATIONS COMPLEMENTAIRES ordi VERT

Entrée [66]:
```
countIteration=1
while countNewPair!=0:
    datasetInputVert=datasetOutputVert
    datasetOutputVert=deduple(ccfIterate(datasetInputVert)[0])
    countNewPair=ccfIterate(datasetInputVert)[1]
    countIteration=countIteration+1

# On réalise plusieurs Itérations
# ... Jusqu'à ce que le compteur = 0 "while countNewPair!=0:"
# Boucle de CCF Itérate + Dédoublonage
```

Entrée [67]:
```
countIteration

# Compteur du nb de Cycles d'Itération pour arriver à un compteur de nouvelles paires = 0
# Si compteur itération = 5 cela signifie 4 de plus après l'itération initiale
# Si countIteration = 5, nb d'itérations avant d'avoir 1 nombre de nouvelles paires = 0
```

Out[67]: 5

Entrée [61]:
```
datasetOutputVert
```

Out[61]:
```
[[873984, 5],
 [808449, 8],
 [902658, 5],
 [85506, 10],
 [115712, 11],
 [407045, 5],
 [846855, 8],
 [86022, 5],
 [735251, 9],
 [507927, 10],
 [140825, 4],
 [882714, 8],
 [747035, 7],
 [310299, 9],
 [280605, 11],
 [900126, 4],
 [145948, 5],
```

Etape 6 : ITERATION INITIALE ordi ORANGE (ccfIterate)

Entrée [70]:
```
datasetOutputOrange = ccfIterate(datasetOrange)
```

Entrée [71]:
```
datasetOutputOrange
```

Out[71]:
```
([[223236, 1],
  [777734, 1],
  [276233, 1],
  [203402, 1],
  [822292, 1],
  [552600, 1],
  [610457, 1],
  [150176, 1],
  [189348, 1],
  [578091, 1],
  [635575, 1],
  [484795, 1],
  [893884, 1],
  [521022, 1],
  [867774, 1],
  [425667, 1],
  [8517, 1],
  [748615, 1],
  [522701, 1],
```

Etape 7 : DEDOUBLONNAGE ordi ORANGE (deduple)

Entrée [72]:
```
datasetOutputOrange = deduple(ccfIterate(datasetOrange)[0])
```

Entrée [73]:
```
datasetOutputOrange
```

Out[73]:
```
[[223236, 1],
 [777734, 1],
 [276233, 1],
 [203402, 1],
 [822292, 1],
 [552600, 1],
 [610457, 1],
 [150176, 1],
 [189348, 1],
 [578091, 1],
 [635575, 1],
 [484795, 1],
 [893884, 1],
 [521022, 1],
 [867774, 1],
 [425667, 1],
 [8517, 1],
 [748615, 1],
 [522701, 1],
```

Entrée [74]:
```
countNewPair = ccfIterate(datasetOrange)[1]
```

Entrée [75]:
```
countNewPair
```

Out[75]:
```
62
```

Etape 8 : ITERATIONS COMPLEMENTAIRES ordi ORANGE

Entrée [76]:
```
countIteration=1
while countNewPair!=0:
    datasetInputOrange=datasetOutputOrange
    datasetOutputOrange=deduple(ccfIterate(datasetInputOrange)[0])
    countNewPair=ccfIterate(datasetInputOrange)[1]
    countIteration=countIteration+1
```

Entrée [77]:
```
countIteration
```

Out[77]:
```
3
```

Entrée [78]:
```
datasetOutputOrange
```

Out[78]:
```
[[223236, 1],
 [10245, 2],
 [777734, 1],
 [539143, 1],
 [528391, 2],
 [451592, 0],
 [760842, 0],
 [600077, 4],
 [654863, 2],
 [648208, 1],
 [822292, 1],
 [385050, 2],
 [124446, 4],
 [91681, 4],
 [412196, 4],
 [397349, 2],
 [412198, 4],
 [430119, 2],
 [789543, 1],
```

Etape 9 : TRAITEMENT FINAL

```
# Aggrégation des résultats obtenus avec les traitements vert et orange.

# récup des traitements fait par les 2 ordinateurs par simulation dans notre cas (traitements vert et orange)
```

Entrée [81]:
```
# On réunit les deux datasetOutput vert et orange
datasetOutput = list()
datasetOutput.extend(datasetOutputVert)
datasetOutput.extend(datasetOutputOrange)
```

Entrée [82]:
```
datasetOutput
```

Out[82]:
```
[[873984, 5],
 [808449, 8],
 [902658, 5],
 [85506, 10],
 [115712, 11],
 [407045, 5],
 [846855, 8],
 [86022, 5],
 [735251, 9],
 [507927, 10],
 [140825, 4],
 [882714, 8],
 [747035, 7],
 [310299, 9],
 [280605, 11],
 [900126, 4],
 [145948, 5],
 [42021, 11],
 [801318, 7],
```

Entrée [88]:
```
# Exécution de l'itération initiale

datasetOutputFinal = ccfIterate(datasetOutput)[0]

# CCF Iterate + Dédoublonage
```

Entrée [89]:
```
datasetOutputFinal
```

Out[89]:
```
[[86022, 5],
 [600077, 4],
 [528391, 2],
 [735251, 9],
 [507927, 10],
 [882714, 8],
 [385050, 2],
 [280605, 11],
 [397349, 2],
 [430119, 2],
 [852008, 11],
 [464936, 4],
 [571440, 7],
 [747569, 2],
 [149560, 4],
 [778296, 4],
 [133180, 5],
 [622656, 11],
 [608321, 5],
```

Entrée [91]:
```
# Exécution du dédoublonnage

datasetOutputFinal=deduple(ccfIterate(datasetOutput)[0])
```

Entrée [92]:
```
datasetOutputFinal
```

Out[92]:
```
[[86022, 5],
 [600077, 4],
 [528391, 2],
 [735251, 9],
 [507927, 10],
 [882714, 8],
 [385050, 2],
 [280605, 11],
 [397349, 2],
 [430119, 2],
 [852008, 11],
 [464936, 4],
 [571440, 7],
 [747569, 2],
 [149560, 4],
 [778296, 4],
 [133180, 5],
 [622656, 11],
 [608321, 5],
```

Entrée [93]:
```
# Exécution des itérations complémentaires

countIteration=1
while countNewPair!=0:
    datasetInputFinal=datasetOutputFinal
    datasetOutputFinal=deduple(ccfIterate(datasetInputFinal)[0])
    countNewPair=ccfIterate(datasetInputFinal)[1]
    countIteration=countIteration+1
```

Entrée [94]:
```
countIteration
```

Out[94]:
```
1
```

Entrée [97]:
```
# Affichage des sous-graphes et de leur nomnbre

countSousGraphes=0
SousGraphes=set()
for i in range(len(datasetOutputFinal)):
    SousGraphes.add(datasetOutputFinal[i][1])

# Création d'une liste de tous les sous-graphes, en ajoutant les éléments de la colonne 1
# Les doublons ne sont pas ajoutés
```

Entrée [98]:
```
# Affichage du nombre de sous-graphes :

countSousGraphes = len(list(SousGraphes))
countSousGraphes

# count devant SousGraphes : indique le nb d'élements
```

Out[98]:
```
11
```

Entrée [99]:
```
# Les sous-graphes sont identifiés par les indices suivants :

SousGraphes
```

Out[99]:
```
{0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11}
```

Entrée [100]:
```
# Noeuds d'un sous-graphe donné

Noeuds_sousgraphe(6)
```

Out[100]:
```
{119755, 177631, 188708, 546100, 668992}
```

Entrée [101]:
```
# Noeuds d'un sous-graphe donné?

Noeuds_sousgraphe(7)
```

Out[101]:
```
{9925,
 13593,
 17517,
 19576,
```