RESEARCH PROJECT

DGPS: Towards Large-scale Distributed Graph Processing System

Name	Dong Liu
Email	ldtenacity@gmail.com
Institute	Peking University DAIR Lab
Date	October 18, 2022

Abstract

Analyzing large graphs provides valuable insights for social networking and web companies in content ranking and recommendations. However, the connectivity of nodes and edges in the graph structure makes it difficult to run many graph algorithms on large graphs in a distributed way. In this research project, we propose a new distributed graph data processing system: DGPS. DGPS is based on the distributed computing framework Hadoop, which implements parallel computing of graph algorithms. Specifically, in practice, we split the graph into several smaller graphs and put them on different machines for computation. Based on DGPS, we have implemented three graph algorithms: Small Dense Subgraphs Finding, PageRank and Graph Connected Components. We choose large scale graphs for our experiments and obtained efficient running speed and good performance no less than the single machine algorithm. These results confirm the efficiency and effectiveness of our framework. It also reveals the potential of distributed graph processing systems.

We provide the complete framework design, algorithm implementation, and experimental procedure and results. We use text descriptions, frame diagrams, pseudo-code and screenshots to show our results and workload in a variety of ways.

Keywords: Distributed system; Graph algorithm; Mapreduce

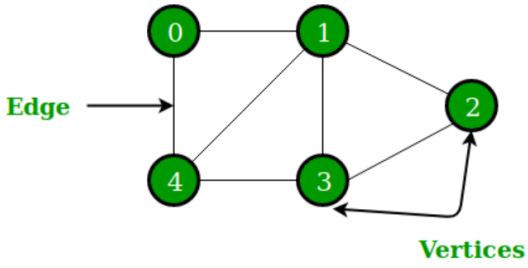
Contents

1. Introduction

1.1 Graph-Structure Data

A graph is an abstract data structure. It holds nodes that are usually related to each other. A node is a dataset, typically in the form of ordered pairs. Nodes are either connected or not connected to another node. The relation between nodes is usually defined as an Edge. Graphs are useful for their ability to associate nodes with other nodes.

A graph is often represented by an Adjacency matrix, A. If a graph has N nodes, then A has a dimension of (NxN). People sometimes provide another feature matrix to describe the nodes in the graph. If each node has F numbers of features, then the feature matrix X has a dimension of (NxF).



1.2 Large Graph-Structure Data

Nowadays the graph-structure data from the Internet is quite large. The web graph is a dramatic example of a large-scale graph. Google estimates that the total number of web pages exceeds 1 trillion; experimental graphs of the World Wide Web contain more than 20 billion nodes (pages) and 160 billion edges (hyperlinks). Graphs of social networks are another example. Facebook reportedly consists of more than a billion users (nodes) and more than 140 billion friendship relationships (edges) in 2012. The LinkedIn network contains almost 8 million nodes and 60 million edges. (Social network graphs are growing rapidly. Facebook went from roughly 1 million users in 2004 to 1 billion in 2012.) In the Semantic Web context, the ontology of DBpedia (derived from Wikipedia), currently contains 3.7 million objects (nodes) and 400 millions facts (edges).

1.3 Whole Graph Processing v.s. Distributed Graph Processing

As for whole graph processing, all computation will be completed in one server. The graph information will be stored on a single machine and all calculation will be executed in one

server.

As for distributed graph processing, a graph will be spiltted into multiple partitions and the several worker nodes will process the partitions in parallel.

1.4 Some Typical Distributed Graph Processing System and their frameworks

1.4.1 Pregel

Pregel is a data flow paradigm and system for large-scale graph processing created at Google to solve problems that are hard or expensive to solve using only the MapReduce framework. While the system remains proprietary at Google, the computational paradigm was adopted by many graph-processing systems, and many popular graph algorithms have been converted to the Pregel framework.

Pregel is essentially a message-passing interface constrained to the edges of a graph. The idea is to ”think like a vertex”. A Pregel computation takes a graph and a corresponding set of vertex states as its inputs. At each iteration, referred to as a superstep, each vertex can send a message to its neighbors, process messages it received in a previous superstep, and update its state. Thus, each superstep consists of a round of messages being passed between neighbors and an update of the global vertex state.

1.4.2 Giraph

Giraph is a framework for performing offline batch processing of semi-structured graph data on massive scale. Giraph is loosely based upon Google's Pregel graph processing framework. Giraph performs iterative calculation on top of an existing Hadoop cluster. Giraph uses Apache Zookeeper to enforce atomic barrier waits and perform leader election

Apache Giraph is an iterative graph processing system built for high scalability. For example, it is currently used at Facebook to analyze the social graph formed by users and their connections. Giraph originated as the open-source counterpart to Pregel, the graph processing architecture developed at Google and described in a 2010 paper. Both systems are inspired by the Bulk Synchronous Parallel model of distributed computation introduced by Leslie Valiant. Giraph adds several features beyond the basic Pregel model, including master computation, sharded aggregators, edge-oriented input, out-of-core computation, and more. With a steady development cycle and a growing community of users worldwide, Giraph is a natural choice for unleashing the potential of structured datasets at a massive scale.

2. Basics of Math and Graph

2.1 Graph

A graph is often denoted by $\mathcal{G} = (V, E)$, where V is the set of vertices and E is the set of edges. An edge $e = u, v$; v has two endpoints u and v, which are said to be joined by e. In this case, u is called a neighbor of v, or in other words, these two vertices are adjacent. Note that an edge can either be directed or undirected. A graph is called a directed graph if all edges are directed or undirected graph if all edges are undirected. The degree of vertice v, denoted by $d(v)$, is the number of edges connected with v. There are a few useful algebra representations for graphs, which are listed as follows.

- **Adjacency matrix**: for a simple graph $G = (V, E)$ with n-vertices, it can be described by an adjacency matrix $A \in \mathbb{R}^{n\times n}$, where

$$A_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \text{ and } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

  It is obvious that such matrix is a symmetric matrix when G is an undirected graph.

- **Degree matrix**: for a graph $G = (V, E)$ with n-vertices, its degree matrix $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix, where

$$D_{ii} = d(v_i).$$

- **Laplacian matrix**: for a simple graph $G = (V, E)$ with n-vertices, if we consider all edges in G to be undirected, then its Laplacian matrix $L \in \mathbb{R}^{n\times n}$ can be defined as

$$L = D - A.$$

  Thus, we have the elements:

$$L_{ij} = \begin{cases} d(v_i) & \text{if } i = j, \\ -1 & \text{if } \{v_i, v_j\} \in E \text{ and } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

  Note that the graph is considered to be an undirected graph for the adjacency matrix.

2.2 Graph algorithms

2.2.1 Breadth-first search

Traversing or searching is one of the fundamental operations which can be performed on graphs. In breadth-first search (BFS), we start at a particular vertex and explore all of its neighbours at the present depth before moving on to the vertices in the next level. Unlike trees, graphs can contain cycles (a path where the first and last vertices are the same). Hence, we have to keep track of the visited vertices. When implementing BFS, we use a queue data structure.

2.2.2 Depth-first search

In depth-first search (DFS) we start from a particular vertex and explore as far as possible along each branch before retracing back (backtracking). In DFS also we have to keep track of the visited vertices. When implementing DFS, we use a stack data structure to support backtracking.

2.2.3 Shortest path

The shortest path from one vertex to another vertex is a path in the graph such that the sum of the weights of the edges that should be travelled is minimum.

- Dijkstra's shortest path algorithm
- Bellman–Ford algorithm

2.2.4 Cycle detection

A cycle is a path in a graph where the first and last vertices are the same. If we start from one vertex, travel along a path and end up at the starting vertex, then this path is a cycle. Cycle detection is the process of detecting these cycles.

- Floyd cycle detection algorithm
- Brent's algorithm

2.2.5 Minimum spanning tree

A minimum spanning tree is a subset of the edges of a graph that connects all the vertices with the minimum sum of edge weights and consists of no cycles.

- Prim's algorithm
- Kruskal's algorithm

2.2.6 Strongly connected components

A graph is said to be strongly connected if every vertex in the graph is reachable from every other vertex.

- Kosaraju's algorithm
- Tarjan's strongly connected components algorithm

2.2.7 Topological sorting

Topological sorting of a graph is a linear ordering of its vertices so that for each directed edge (u, v) in the ordering, vertex u comes before v.

- Kahn's algorithm
- The algorithm based on depth-first search

2.2.8 Graph colouring

Graph colouring assigns colours to elements of a graph while ensuring certain conditions. Vertex colouring is the most commonly used graph colouring technique. In vertex colouring, we try to colour the vertices of a graph using k colours and any two adjacent vertices should not have the same colour. Other colouring techniques include edge colouring and face colouring. The chromatic number of a graph is the smallest number of colours needed to colour the graph.

- Algorithms using breadth-first search or depth-first search
- Greedy colouring

3. Dilemmas and Motivation

3.1 The Dilemma of Processing on Large-scale Graph

With explosion of data volume generated and collected from ubiquitous sensors, portable devices and the Internet, we are now moving into the "Big Data" era. There exist various modern Big Data applications relying on graph computing, including social networks, Internet of things, and neural networks. For example, Facebook has 1.44 billions monthly active users during the first quarter of 2015. Both user data and relationship among them are modeled by graphs for further exploration. To this end, it has become a very important problem to process, analyze, and understand these graphs.

To achieve high performance, existing graph systems aim to exploit massive parallelism using either distributed or shared memory models. Such graph systems process graphs in a repeated-relaxing manner (e.g., using Bellman-Ford algorithm variants) rather than in a sequential but work-optimal order. This introduces a fundamental trade-off between available parallelism and redundant computations.

3.2 Graph Processing Based on Existing Distributed Data Processing Systems.

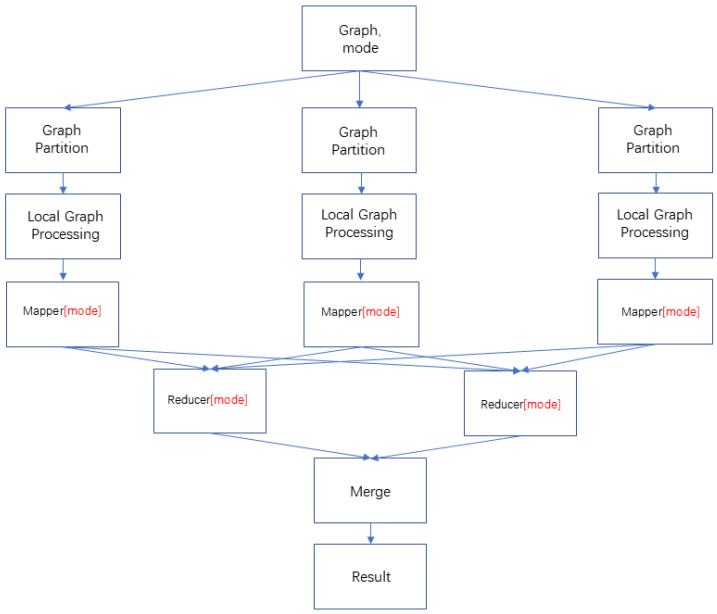
Building systems that process vast amounts of data has been made simpler by the introduction of the MapReduce framework, and its open-source implementation Hadoop. These systems offer automatic scalability to extreme volumes of data, automatic fault-tolerance, and a simple programming interface based around implementing a set of functions. However, it has been recognized that these systems are not always suitable when processing data in the form of a large graph. A framework similar to MapReduce—scalable, fault-tolerant, easy to program—but geared specifically towards graph data, would be of immense use. Google's proprietary Pregel system was developed for this purpose. Pregel is a distributed message-passing system, in which the vertices of the graph are distributed across compute nodes and send each other messages to perform the computation.

4. Framework of DGPS

Our system provide a distributed processing system which can make queries to be splited into several distributed jobs and to be executed on the set of machines in which the intermediate result is stored in the execution graphs.

In this system we provided three useful algorithms to execute the corresponding graph processing jobs on large graphs.

- Graph Partition
- Several mode (Several Algorithms)->Different Mapper & Reducer for different mode
- Parallel Processing



4.1 Platform and Tools

4.1.1 Hadoop

The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.

It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

4.1.2 Hadoop MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a _map_ function that processes a key/value pair to generate a set of intermediate key/value pairs, and a _reduce_ function that merges all intermediate values associated with the same intermediate key.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

4.1.2.1 MapReduce Programming Model

The computation takes a set of _input_ key/value pairs, and produces a set of _output_ key/value pairs. The user of the MapReduce library expresses the computation as two functions: _Map_ and _Reduce_.

Map, takes an input pair and produces a set of _intermediate_ key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the _Reduce_ function.

The _Reduce_ function, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per _Reduce_ invocation. The intermediate values are supplied to the user's reduce function via an iterator.

4.2 Implementation

4.2.1 MapReduce Implementation

1. MapReduce library in the user program first splits the input file into M pieces. It then starts up many copies of the program on a cluster of machines.

2. One of copies of the program is master, and the rest are workers, which assigned by the master. There are M map tasks and R reduce tasks to assign. Master picks idle workers and assigns each one a map task or a reduce tasks.

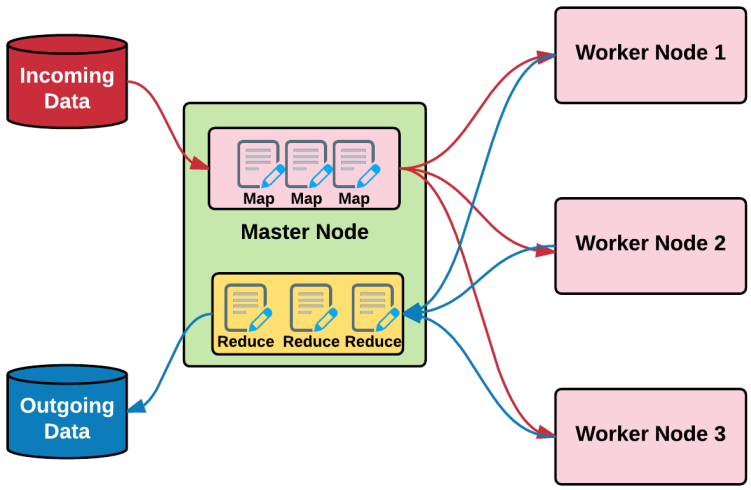
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

4. The buffered pairs are written to local disk, partitioned into R regions by the user-defined partitioning function. The locations of these buffered pairs on the local disk are passed back to the master. Master is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. Reduce worker then sorts it by the intermediate keys.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the reduce function is appended to a final output file for this reduce partition.

7. When all map tasks have been completed, master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.



4.2.2 Graph Processing System Implementation

Hadoop will distribute the entire graph onto multiple devices. When the graph size exceeds a single virtual machine memory, it will assign the extra part of graph to a new virtual machine.

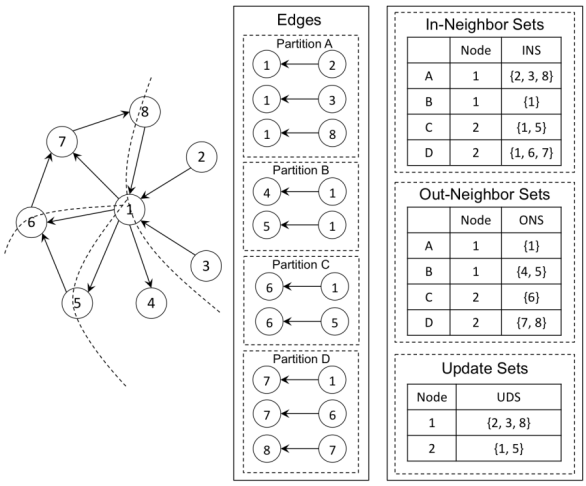
There are mainly two graph partitioning methods, some distributed graph frameworks like PowerGraph and GraphX use a vertex-cut partitioning to optimize for inter-node communication, while others use edge partitioning, an efficient graph partitioning strategy that assigns a roughly equal number of edges to each partition.

As for a typical implementation for graph partitioning (here we talk about directed graph), for each partition, the system will maintain two sets for it, the first one is the in-neighbor set, which contains all the outgoing edges from a specific node to the other nodes in the graph and the second one is out-neighbor set, which contains all the ingoing edges from nodes in the rest of the graph to a specific node.

$INS(P_i) = u|(u, v) \in P_i$

$ONS(P_i) = v|(u, v) \in P_i$

The figure below shows that a graph is divided into four partition, and assigned to 4 workers on 2 virtual machines.

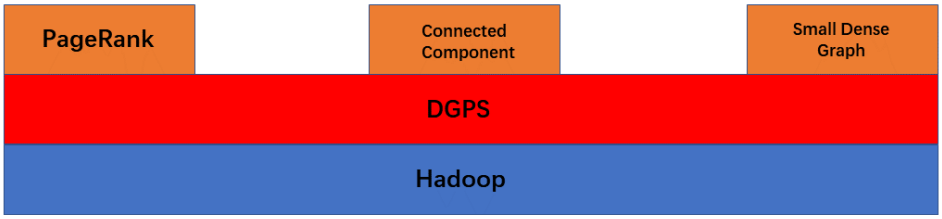


4.3 Architecture

4.3.1 Several Algorithms Application

- Algorithm I:Small Dense Subgraphs Finding
- Algorithm II: PageRank
- Algorithm III: Graph Connected Components

The details of implementation of these algorithms will be mentioned in the later chapter.



4.3.2 Master-Slave Architecture

Our system is based on hadoop to realize the graph distributed computing. The figure below shows the architecture of DGPS.

There are several Key points in the System:

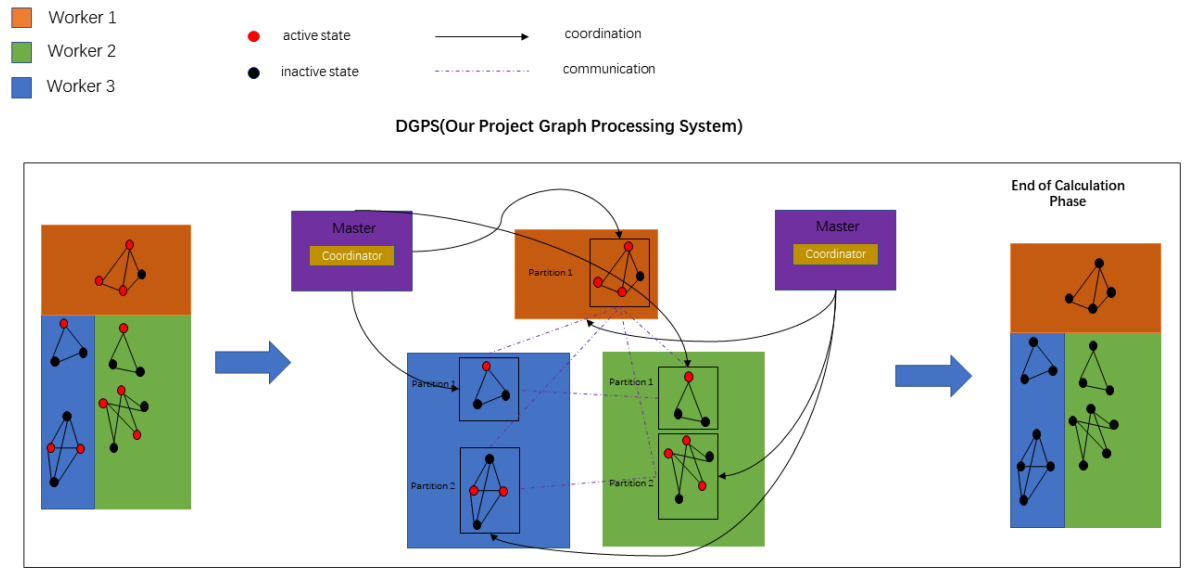
1. Graph is splitted into several partitions

2. Two states of nodes (active, inactive)

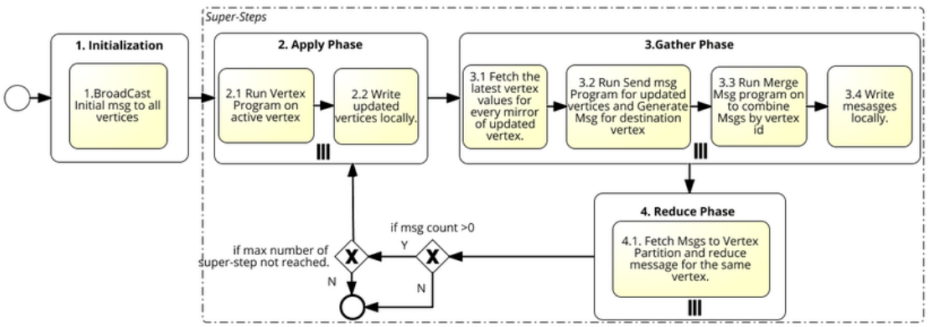
3. Several super-steps

4. Two states of nodes: active & inactivate

5. Master-Slave Architecture, two types of commands from master node: coordination & communication



4.3.3 The details function of each step



4.3.4 Overview of the System

1. Input: A Whole Graph

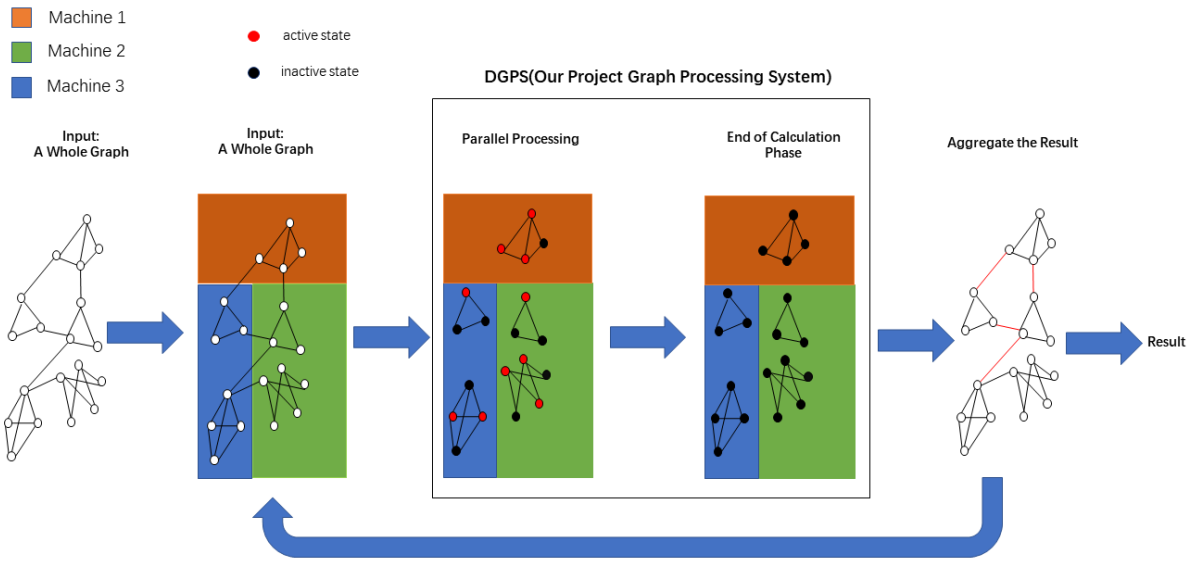
2. Partitioning: Split the Graph into Several Partitions

3. Distributed Computing: Several Worker compute in parallel, each worker processes one or more partitions.

4. Supersteps: Each calculation iteration is referred to as a superstep, each vertex can send a message to its neighbors, process messages it received in a previous superstep, and update its state.

5. Aggregation: An aggregation of the results computed by different worker nodes.

6. Output: The result of aggregation step.



5. Experiments

Environment: IDEA:2021.2.3, hadoop:3.1.3, VMware:15, linux:CentOS7, VMs: Hadoop102 Hadoop103 Hadoop104.

Dataset: http://snap.stanford.edu/biodata/datasets/10008/10008-PP-Decagon.html

This is a protein-protein association network that includes direct (physical) protein-protein interactions, as well as indirect (functional) associations between human proteins. Nodes represent proteins and edges represent associations between them.

dataset	statistics
Nodes	19081
Edges	715612
Nodes in largest SCC	19065
Fraction of nodes in largest SCC	1.000000
Edges in largest SCC	715602
Fraction of edges in largest SCC	0.999986
Average clustering coefficient	0.233544
Number of triangles	157277164
Fraction of closed triangles	0.082030
Diameter (longest shortest path)	8
90-percentile effective diameter	3.199186

5.1 Small Dense Subgraphs Finding

5.1.1 Setup and Code Analysis

Let $\mathcal{G} = (V, E)$ be an undirected graph. For any subset of vertices V' from V, the density "rho" of the subgraph induced by V' is defined as: $rho(V') = |E[V']|/V'$, where $|E[V']|$ subset of E is the set of edges whose endpoints lie in V'. In this problem we are asked to design and implement a MapReduce algorithm that, given a density rho, finds a subgraph of $\mathcal{G}$ with density >= rho, i.e., a set V' such that $rho(V') >= rho$. The challenge is to find a subgraph that is as small as possible. While the problem of finding the smallest dense-enough subgraph is hard, we should attempt at finding a subgraph with density >= rho as small as we can.

The problem is divided into 3 basic steps:

- Compute the density of the whole original graph, "ComputeGraphDensity" class.
- Use the just computed density as a threshold for both pruning and partitioning of the graph, "GraphPartitioning" class.
- Find the smallest subgraph with the corresponding degree threshold (rho value), "ComputeMinDensitySubgraph" class.

Each step is translated in a corresponding MapReduce round:

- **1.** For the first round the Mapper just parses the input graph file, line by line (edge by edge). All the edges are being transferred to only one Reducer. The Reducer computes the density of the whole graph, iterating over the edges and keeping track of the nr of edges and nodes. The output file produced by this Reducer is of the following form:

```
                                #
#    [src]<tab>[dst]<tab>[graph_density]    #
# ... [src]<tab>[dst]<tab>[graph_density] ... #
# ..................................................... #
                                #
```

map phase

input:

“‘ [src]<tab>[dst] “‘

output:

“‘ <1, [src]<tab>[dst]> “‘

code:

```
1 public void map(LongWritable key, Text edge, Context
    context) throws IOException {
2   if (edge.toString().isEmpty()) return;
3   try {
```

```
            context.write(new IntWritable(1), edge);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
                    }
```

reduce phase

input:

“‘ <1, list[src]<tab>[dst]> “‘

output

“‘ <[src]<tab>[dst], [graph_density]> “‘

code:

```
public void reduce(IntWritable key, Iterable<Text> edges,
    Context context) throws IOException {
    HashSet<Integer> hashSet = new HashSet<>();
    HashSet<String> edgeSet = new HashSet<>();

    Iterator<Text> iter = edges.iterator();
    Queue<String> queue = new LinkedList<>();
    String hashEdge;
    //while there are edges left
    while (iter.hasNext()) {
        Text line = (Text) iter.next();
        String[] edge = line.toString().split("\t");
        int src = Integer.parseInt(edge[0]);
        int dst = Integer.parseInt(edge[1]);

        if (src < dst) hashEdge = src + " - " + dst;
        else hashEdge = dst + " - " + src;

        if (edgeSet.contains(hashEdge)) continue;
        edgeSet.add(hashEdge);

        queue.add(line.toString());
        hashSet.add(src);
        hashSet.add(dst);
    }

    //computing density of the whole graph as |E|/|V|
    int edgeCount = queue.size();
    int nodeCount = hashSet.size();
    double graph_density = (double) edgeCount / (double)
       nodeCount;

    Text K_out = new Text(), V_out = new Text();
    while (!queue.isEmpty()) {
```

```
33
34          K_out.set(queue.poll());
35          V_out.set(String.valueOf(graph_density));
36          try {
37              context.write(K_out, V_out);
38          } catch (InterruptedException e) {
39              e.printStackTrace();
40          }
41      }
42                  }
```

- **2.** In the second round the Mapper parses the output file, line by line, produced by the previous MapReduce round. For each parsed line, emits a <subgraphID, edge> keyvalue pair, randomly choosing the "subgraphID" in [0 graph_density) range. The Reducer receives as input a <subgraphID, List<edge» keyvalue pair, which corresponds to the specific "subgraphID" subgraph of the original graph. For each edge of the corresponding subgraph, it explores the degrees (in the subgraph's context) of both edge's endpoints, and emits the edge only if the degrees of the both endpoints are greater than graph_density. This round is very important to the success of the entire algorithm because: first, the algorithm takes advantage of parallelized computation (subgraph pruning is done in each graph partition) and second, it throws away edges which endpoints have very few links (endpoint_degree < graph_density), and thus nodes with small degree that don't contribute as much to the final desired result, are pruned. In sum, the purpose of the round is to cross out those nodes that don't contribute to the result, and, the output are the edges which endpoints are more suitable to help us in the task.

  map phase

  input:

  "' same as the output of previous reduce phase "'

  output:

  "' <[subgraphID], [edge]> subgraphID is randomly picked. "'

  code:

```
1  public void map(LongWritable key, Text record/*edge "[src
     ]<tab>[dst]"*/, Context context) throws IOException {
2      if (record.toString().isEmpty()) return;
3
4      String[] line = record.toString().split("\t");
5      String edge = line[0] + "\t" + line[1];
6      double graphDensity = Double.parseDouble(line[2]);
7      int partitions = (int) Math.round(graphDensity);
8      int partition = new Random().nextInt(partitions);
9
10     System.out.println("Mapper: [graphDensity " +
         graphDensity + "] [partitions " + partitions +
11           "] [subgraph " + partition + "] edge - " +
               edge);
```

```
12
13      //<key, value> pair as <subgraphID, (edge, density)>
14      try {
15          context.write(new IntWritable(partition), new
                Text(edge + "\t" + graphDensity));
16      } catch (InterruptedException e) {
17          e.printStackTrace();
18      }
19  }
```

reduce phase

input:

“‘ <[subgraphID], List[edge]> “‘

output:

“‘ <NULL, [src]<tab>[dst]> “‘

code:

```
1   public void reduce(IntWritable key, Iterable<Text> edges,
        Context context) throws IOException {
2       int graphDensity = 0;
3       HashMap<Integer, Integer> hashMap = new HashMap<>();
4
5       Iterator<Text> iter = edges.iterator();
6       Queue<String> queue = new LinkedList<>();
7       //while there are edges left
8       while (iter.hasNext()) {
9           Text line = iter.next();
10          String[] edge = line.toString().split("\\t");
11          queue.add(edge[0] + "\t" + edge[1]);
12
13          int src = Integer.parseInt(edge[0]);          //first
                node of this edge
14          int dst = Integer.parseInt(edge[1]);          //
                second node of this
                edge
15
16          if (hashMap.containsKey(src)) hashMap.put(src,
                hashMap.get(src) + 1);      //update degree of this
                node
17          else hashMap.put(src, 1);
18
19          if (hashMap.containsKey(dst)) hashMap.put(dst,
                hashMap.get(dst) + 1);      //update degree of this
                node
20          else hashMap.put(dst, 1);
21
22          if (!iter.hasNext()) graphDensity = (int) Math.
                round(Double.parseDouble(edge[2]));
23      }
```

```
24
25      Text output_edge = new Text();
26      while (!queue.isEmpty()) {
27          String[] edge_ = queue.poll().split("\\t");
28          int src = Integer.parseInt(edge_[0]);
29          int dst = Integer.parseInt(edge_[1]);
30
31          if (hashMap.get(src) > graphDensity && hashMap.
                get(dst) > graphDensity) {
32              output_edge.set(src + "\t" + dst);
33              System.out.println("Reducer: [subgraph " +
                    key.toString() + "] OUT(null, "
34                          + output_edge.toString() + ")");
35              try {
36                  context.write(null, output_edge);
37              } catch (InterruptedException e) {
38                  e.printStackTrace();
39              }
40          }
41              }
42          }
```

- **3.** In this round the Mapper just parses the partitioned input graph file, produced by the previous MapReduce round, line by line (edge by edge), and outputs all the edges with the same key so that they are transferred to only one Reducer. The Reducer iterates over all the nodes, and at each iteration it removes from the graph the node with the minimum degree. Then it computes the new graph density and if the new graph is better, it stores the new best density of at least rho and the new best graph. This will continue until all nodes have been removed from the graph and in the end we will have stored the smallest graph of at least density rho the algorithm could find.

  map phase

  input:

  “‘ same as the output of previous reduce phase “‘

  output:

  “‘ <1, [edge]> “‘

  code:

```
1       public void map(LongWritable key, Text edge/*edge "[
            src]<tab>[dst]"*/, Context context) throws
            IOException {
2           if (edge.toString().isEmpty()) return;
3           //<key, value> pairs as <1, edge>
4           try {
5               context.write(new IntWritable(1), edge);
6           } catch (InterruptedException e) {
7               e.printStackTrace();
8           }
```

```
        }
```

reduce phase

input:

“‘ <1, list[edge]> “‘

output:

“‘ <NULL, [edge]> “‘

code:

```
public void reduce(IntWritable key, Iterable<Text> edges,
    Context context) throws IOException {
    final int targetDensity = 4;      //the corresponding rho value
    int edgeCount = 0, nodeCount = 0;
    Hashtable<Integer, LinkedList<Integer>> graph = new
       Hashtable<>();
    Hashtable<Integer, LinkedList<Integer>> minSubgraph =
        null;
    int maxIdxNode = 0;
    for (Text text : edges) {
        String[] edge = text.toString().split("\t");
        int src = Integer.parseInt(edge[0]);
        int dst = Integer.parseInt(edge[1]);

        if (src > maxIdxNode) maxIdxNode = src;
        if (dst > maxIdxNode) maxIdxNode = dst;

        if (graph.containsKey(src)) graph.get(src).add(
           dst);
        else {
            LinkedList<Integer> src_neighbors = new
               LinkedList<>();
            src_neighbors.add(dst);

            graph.put(src, src_neighbors);
        }

        if (graph.containsKey(dst)) graph.get(dst).add(
           src);
        else {
            LinkedList<Integer> dst_neighbors = new
               LinkedList<>();
            dst_neighbors.add(src);

            graph.put(dst, dst_neighbors);
        }

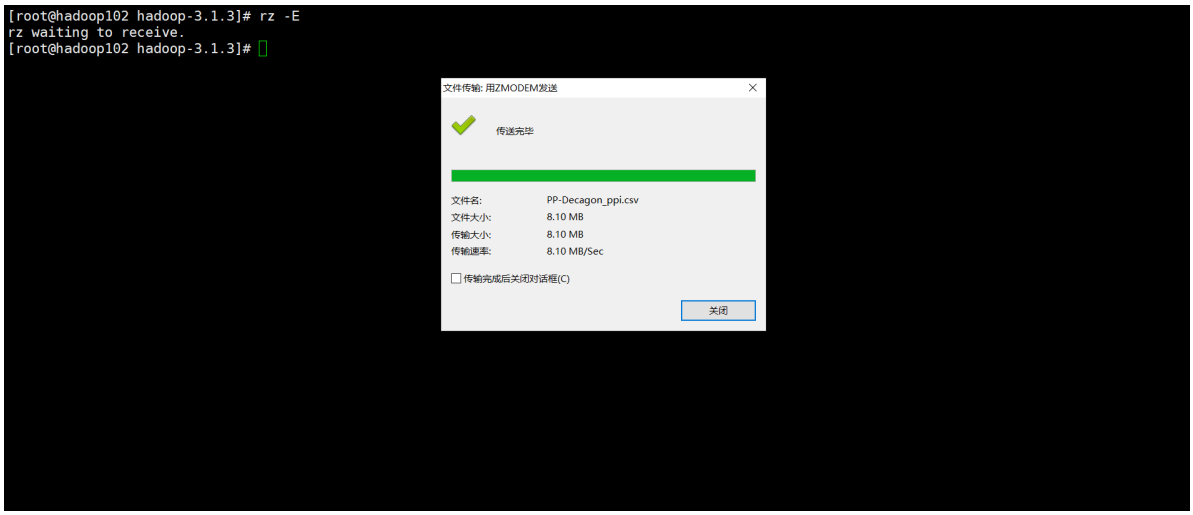
        edgeCount++;
```

```
32      }
33
34      nodeCount = graph.size();
35      IndexMinPQ<Integer> minHeap = new IndexMinPQ<>(
            maxIdxNode + 1);
36
37      for (int node : graph.keySet()) {
38          minHeap.insert(node, graph.get(node).size());
39          System.out.println("node " + node + "\tdegree " +
                graph.get(node).size());
40      }
41
42      int minNode, minNodesSub = 100, minEdgeCount = 0;
43      while (!minHeap.isEmpty())
44      {
45          minNode = minHeap.delMin();          //extract the
                minimum degree node from the heap
46          System.out.println("minNode = " + minNode);
47          for (int neighbor : graph.get(minNode)) {
48              //remove the minimum degree node from the neighbor's
                    adjacency list
49              graph.get(neighbor).remove(Integer.valueOf(
                    minNode));
50              minHeap.decreaseKey(neighbor, graph.get(
                    neighbor).size());      //update the neighbor node
                    degree
51          }
52
53          edgeCount -= graph.get(minNode).size();      //
                update number of edges of the
                graph
54          nodeCount--;                                 //update
                number of nodes of the graph
55          graph.remove(minNode);                       //remove
                the minimum degree node from the graph structure
56
57          if ((((float) edgeCount / (float) nodeCount) >=
                4) && nodeCount < minNodesSub) {
58              minNodesSub = nodeCount;
59              minEdgeCount = edgeCount;
60              minSubgraph = new Hashtable<>(graph);
61
62
63              System.out.println("Reducer: minNodesSub = "
                    + minNodesSub);
64          }
65      }
66
67      if (minSubgraph == null) return;
68
```

```
69      try {
70          float minDensity = (float) minEdgeCount / (float)
                minSubgraph.size();
71          System.out.println("Reducer: density of the
               minimum subgraph = " + minDensity);
72          context.write(null, new Text(String.valueOf(
               minDensity)));
73      } catch (InterruptedException e) {
74          e.printStackTrace();
75      }
76
77      Text output_node = new Text();
78      for (int node : minSubgraph.keySet()) {
79          output_node.set(String.valueOf(node));
80          try {
81              System.out.println("Reducer: OUT(null, " +
                   output_node + ")");
82              context.write(null, output_node);
83          } catch (InterruptedException e) {
84              e.printStackTrace();
85          }
86      }
87
88      String hashEdge;
89      HashSet<String> edgeSet = new HashSet<>();
90      Text output_edge = new Text();
91      for (int src : minSubgraph.keySet()) {
92          for (int dst : minSubgraph.get(src)) {
93              if (src < dst) hashEdge = src + "\t" + dst;
94              else hashEdge = dst + "\t" + src;
95
96              if (!edgeSet.contains(hashEdge)) {
97                  edgeSet.add(hashEdge);
98                  output_edge.set(hashEdge);
99                  try {
100                     System.out.println("Reducer: OUT(null
                           , " + hashEdge + ")");
101                     context.write(null, output_edge);
102                 } catch (InterruptedException e) {
103                     e.printStackTrace();
104                 }
105             }
106         }
107     }
108         }
```

5.1.2 Results

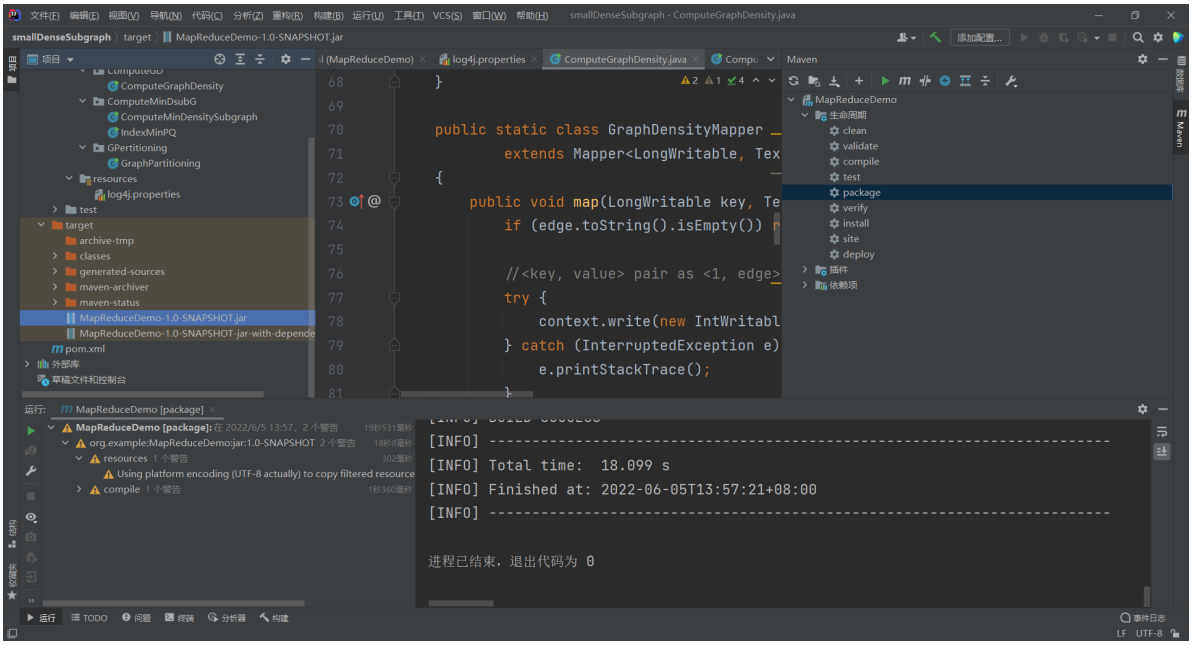
1. load dataset file(PP-Decagon_ppi.csv) to VM



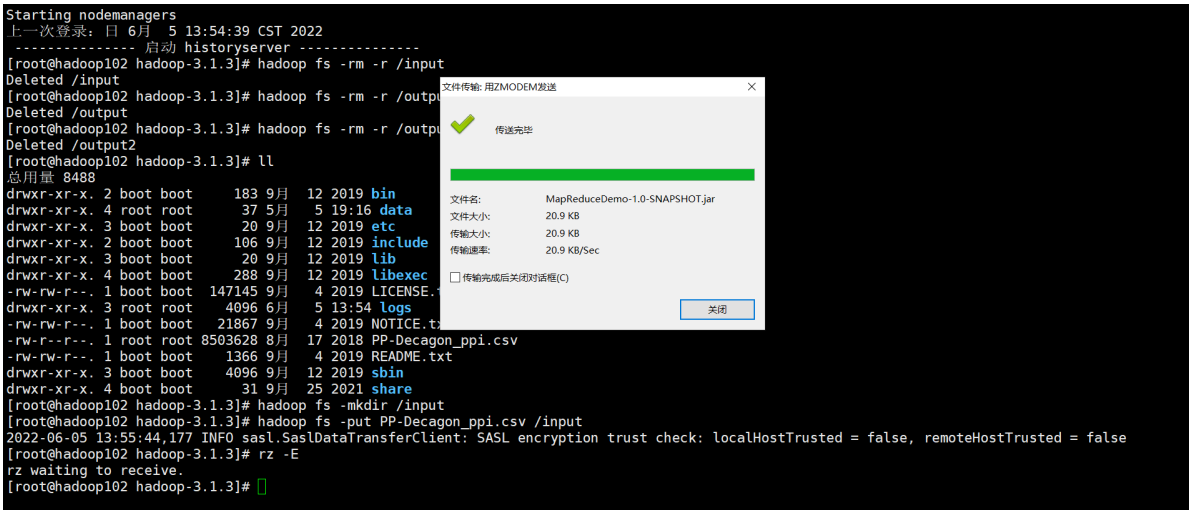
2. put dataset file(PP-Decagon_ppi.csv) to HDFS

```
[root@hadoop102 hadoop-3.1.3]# ll
总用量 8488
drwxr-xr-x. 2 boot boot     183 9月  12 2019 bin
drwxr-xr-x. 4 root root      37 5月   5 19:16 data
drwxr-xr-x. 3 boot boot      20 9月  12 2019 etc
drwxr-xr-x. 2 boot boot     106 9月  12 2019 include
drwxr-xr-x. 3 boot boot      20 9月  12 2019 lib
drwxr-xr-x. 4 boot boot     288 9月  12 2019 libexec
-rw-rw-r--. 1 boot boot  147145 9月   4 2019 LICENSE.txt
drwxr-xr-x. 3 root root    4096 6月   5 13:54 logs
-rw-rw-r--. 1 boot boot   21867 9月   4 2019 NOTICE.txt
-rw-r--r--. 1 root root 8503628 8月  17 2018 PP-Decagon_ppi.csv
-rw-rw-r--. 1 boot boot    1366 9月   4 2019 README.txt
drwxr-xr-x. 3 boot boot    4096 9月  12 2019 sbin
drwxr-xr-x. 4 boot boot      31 9月  25 2021 share
[root@hadoop102 hadoop-3.1.3]# hadoop fs -mkdir /input
[root@hadoop102 hadoop-3.1.3]# hadoop fs -put PP-Decagon_ppi.csv /input
2022-06-05 13:55:44,177 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
[root@hadoop102 hadoop-3.1.3]#
```

3. compile program to Jar file



4. load Jar file to VM

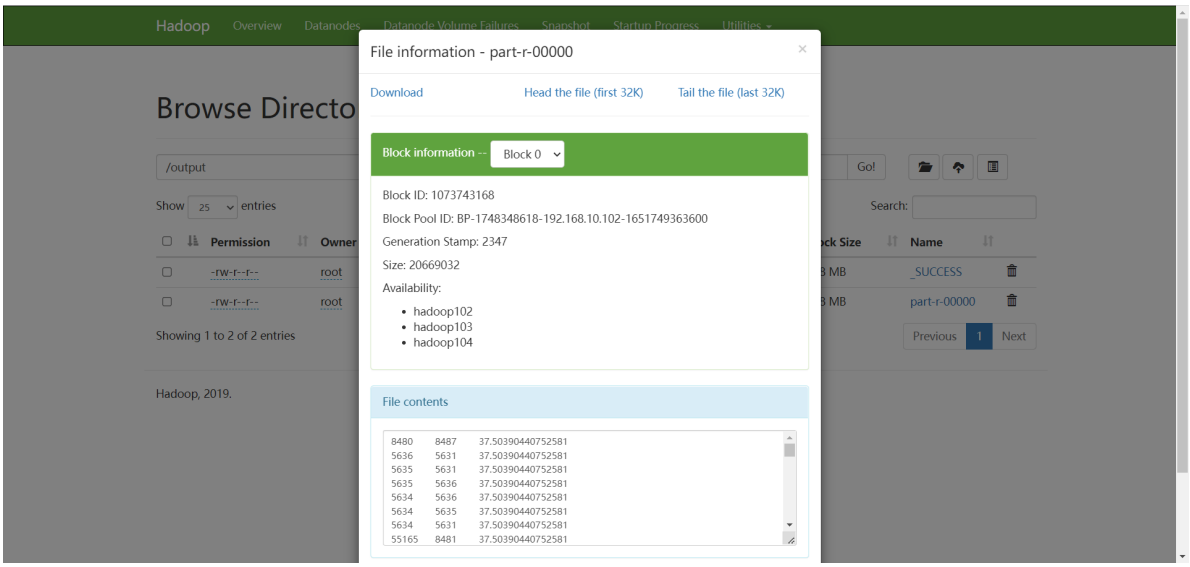


5. run program in VM

task1: compute graph density

```
-rw-rw-r--. 1 boot boot   21867 9月   4 2019 NOTICE.txt
-rw-r--r--. 1 root root 8503628 8月  17 2018 PP-Decagon_ppi.csv
-rw-rw-r--. 1 boot boot    1366 9月   4 2019 README.txt
drwxr-xr-x. 3 boot boot    4096 9月  12 2019 sbin
drwxr-xr-x. 4 boot boot      31 9月  25 2021 share
[root@hadoop102 hadoop-3.1.3]# hadoop fs -mkdir /input
[root@hadoop102 hadoop-3.1.3]# hadoop fs -put PP-Decagon_ppi.csv /input
2022-06-05 13:55:44,177 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
[root@hadoop102 hadoop-3.1.3]# rz -E
rz waiting to receive.
[root@hadoop102 hadoop-3.1.3]# hadoop jar MapReduceDemo-1.0-SNAPSHOT.jar ComputeGD.ComputeGraphDensity /input/PP-Decagon_ppi.csv /output
2022-06-05 14:00:23,869 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 14:00:24,257 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and ex
ecute your application with ToolRunner to remedy this.
2022-06-05 14:00:24,275 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1654408
482149_0001
2022-06-05 14:00:24,348 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 14:00:24,454 INFO input.FileInputFormat: Total input files to process : 1
2022-06-05 14:00:24,487 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 14:00:24,524 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 14:00:24,550 INFO mapreduce.JobSubmitter: number of splits:1
2022-06-05 14:00:24,642 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 14:00:24,676 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654408482149_0001
2022-06-05 14:00:24,677 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 14:00:24,814 INFO conf.Configuration: resource-types.xml not found
2022-06-05 14:00:24,815 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2022-06-05 14:00:24,973 INFO impl.YarnClientImpl: Submitted application application_1654408482149_0001
2022-06-05 14:00:25,002 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654408482149_0001/
2022-06-05 14:00:25,002 INFO mapreduce.Job: Running job: job_1654408482149_0001
```

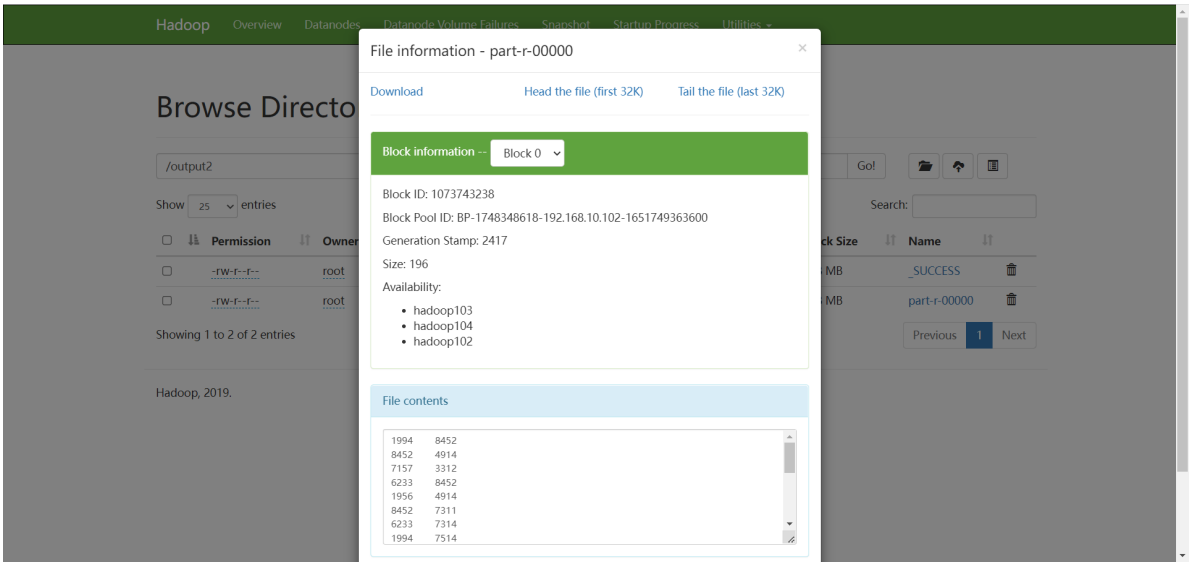
result:



task2: graph partitioning

```
[root@hadoop102 hadoop-3.1.3]# hadoop jar MapReduceDemo-1.0-SNAPSHOT.jar GPartitioning.GraphPartitioning /output/part-r-00000 /output2
2022-06-05 16:54:40,006 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 16:54:40,338 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and ex
ecute your application with ToolRunner to remedy this.
2022-06-05 16:54:40,353 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1654419
132285_0002
2022-06-05 16:54:40,415 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 16:54:40,510 INFO input.FileInputFormat: Total input files to process : 1
2022-06-05 16:54:40,532 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 16:54:40,561 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 16:54:40,582 INFO mapreduce.JobSubmitter: number of splits:1
2022-06-05 16:54:40,670 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 16:54:40,697 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654419132285_0002
2022-06-05 16:54:40,697 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 16:54:40,826 INFO conf.Configuration: resource-types.xml not found
2022-06-05 16:54:40,826 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2022-06-05 16:54:40,865 INFO impl.YarnClientImpl: Submitted application application_1654419132285_0002
2022-06-05 16:54:40,892 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654419132285_0002/
2022-06-05 16:54:40,892 INFO mapreduce.Job: Running job: job_1654419132285_0002
2022-06-05 16:54:46,013 INFO mapreduce.Job: Job job_1654419132285_0002 running in uber mode : false
2022-06-05 16:54:46,016 INFO mapreduce.Job:  map 0% reduce 0%
2022-06-05 16:54:55,114 INFO mapreduce.Job:  map 100% reduce 0%
2022-06-05 16:55:01,200 INFO mapreduce.Job:  map 100% reduce 100%
2022-06-05 16:55:02,224 INFO mapreduce.Job: Job job_1654419132285_0002 completed successfully
2022-06-05 16:55:02,286 INFO mapreduce.Job: Counters: 53
        File System Counters
                FILE: Number of bytes read=24962710
                FILE: Number of bytes written=50361849
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
```

result:

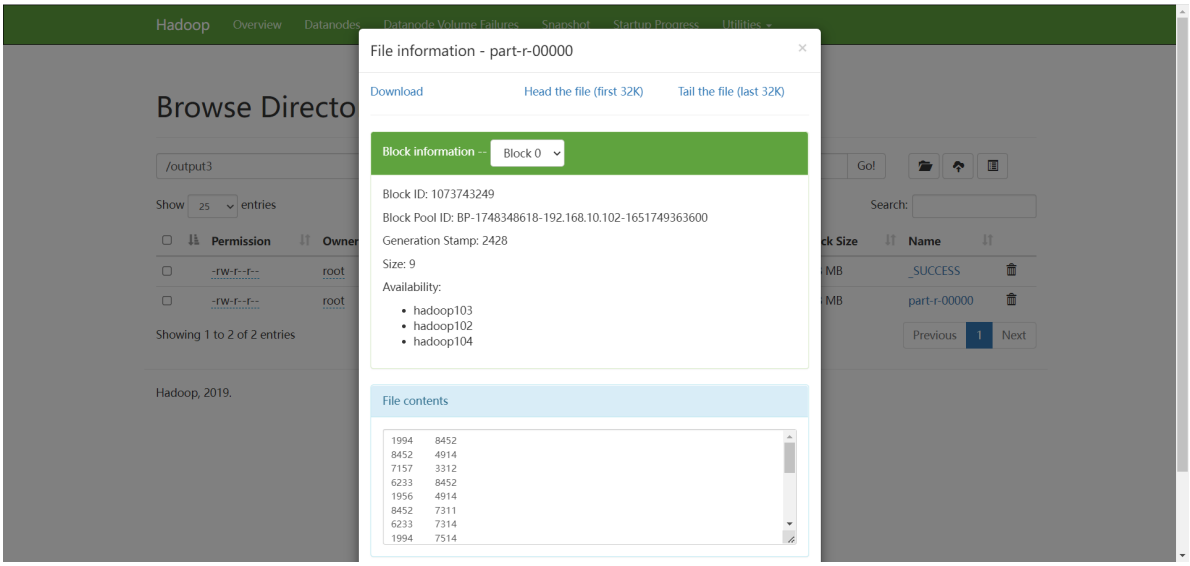


task3: compute min subgraph density

```
                Bytes Written=196
[root@hadoop102 hadoop-3.1.3]# hadoop jar MapReduceDemo-1.0-SNAPSHOT.jar ComputeMinDsubG.ComputeMinDensitySubgraph /output2/part-r-00000 /output3

2022-06-05 16:55:44,497 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 16:55:44,831 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and ex
ecute your application with ToolRunner to remedy this.
2022-06-05 16:55:44,843 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1654419
132285_0003
2022-06-05 16:55:44,910 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 16:55:45,003 INFO input.FileInputFormat: Total input files to process : 1
2022-06-05 16:55:45,034 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 16:55:45,066 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 16:55:45,091 INFO mapreduce.JobSubmitter: number of splits:1
2022-06-05 16:55:45,188 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 16:55:45,617 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654419132285_0003
2022-06-05 16:55:45,617 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 16:55:45,743 INFO conf.Configuration: resource-types.xml not found
2022-06-05 16:55:45,744 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2022-06-05 16:55:45,781 INFO impl.YarnClientImpl: Submitted application application_1654419132285_0003
2022-06-05 16:55:45,807 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654419132285_0003/
2022-06-05 16:55:45,808 INFO mapreduce.Job: Running job: job_1654419132285_0003
2022-06-05 16:55:49,889 INFO mapreduce.Job: Job job_1654419132285_0003 running in uber mode : false
2022-06-05 16:55:49,890 INFO mapreduce.Job:  map 0% reduce 0%
2022-06-05 16:55:54,940 INFO mapreduce.Job:  map 100% reduce 0%
2022-06-05 16:55:59,005 INFO mapreduce.Job:  map 100% reduce 100%
2022-06-05 16:55:59,021 INFO mapreduce.Job: Job job_1654419132285_0003 completed successfully
2022-06-05 16:55:59,085 INFO mapreduce.Job: Counters: 53
        File System Counters
                FILE: Number of bytes read=316
                FILE: Number of bytes written=437155
```

result:



5.2 PageRank

5.2.1 Setup and Code Analysis

We can state the Page Rank algorithm in a more general way. We want to assign scores so that

$$x_i = \sum_{j \in L_i} \frac{x_j}{n_j}$$

where L_i are the indices such that v_j links to v_i if $j \in L_i$, and n_j is equal to outdegree of v_j. Note that L_i contains $i \deg(v_i)$ elements. The set L_i is called the set of *backlinks* of vertex v_i. This can be rewritten as a vector equation

$$x = Ax,$$

where A is the matrix $A = (a_{ij})$ given by

$$a_{ij} = \begin{cases} \frac{1}{n_j} & \text{if } j \in L_i \\ 0 & \text{otherwise} \end{cases}.$$

This matrix is called the *link matrix*. We note that in the example, the matrix A was

$$A = \begin{bmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix}.$$

The problem of solving for the scores x then amounts to finding an eigenvector with eigenvalue 1 for the matrix A.

We can consider the link matrix as giving the probabilities of traversing a link from the page represented by the column to the page representing the row. Thus it makes sense that the sum of the values of the columns are equal to one.

The basic idea is that we can try to compute an eigenvector iteratively like

$$x_{k+1} = Mx_k = M^k x_0.$$

Certainly, if $Mx_0 = x_0$, then this procedure fixes x_0. In general, if we replace this method with

$$x_{k+1} = \frac{Mx_k}{\|Mx_k\|}$$

for any vector norm, we will generally find an eigenvector for the largest eigenvalue.

In this research projecct, for each graph, we list the top ten nodes in order of descending PageRank value. In addition, list a summary of each graph property, e.g., number of nodes, number of edges, and (min, max, avg) of out-degree for each node; list the total number of MapReduce iterations for convergence.

The MapReduce method for PageRank we use is shown below:

Map:

- Input:
  - key: index.html
  - value: <pagerank> 1.html 2.html...
  - Output for each outlink:
  - key: "1.html"
  - value: "index.html <pagerank> <number of outlinks>"

Reduce

- Input:
  - Key: "1.html"
  - Value: "index.html 0.5 23"
    Value: "2.html 2.4 2"
    Value: ...
- Output:
  - Key: "1.html"
  - Value: "<new pagerank> index.html 2.html..."

When the ranks converge, we uses also MapReduce to sort the ranks and outputs only the top 10 nodes. The MapReduce method is shown below: Reduce is simply outputting them. After that, the graph information is output using another simple MapReduce.

1. preprocess the dataset "Driver" class.
2. iteratively rank the pages until they converge, "PageRank" class.
3. sort the ranks and outputs only the top 10 nodes, "SortRank" class.
4. output graph information, "AddInfo" class.

1. Preprocess

In this phase, dataset will be transformed to target format, using simple mapreduce operation.

Mapper phase does not do any thing, reducer phase will gather all the adjacent nodes into one simple Text, in the format of following.

input:

"' [src]<tab>[dst] "'

output:

Map:

- Input:
  - Key: “index.html”
  - Value: “<pagerank> <outlinks>”
- Output:
  - Key: “<pagerank>”
  - Value: “index.html”

“‘ <[node], [adjacent node]<tab>[adjacent node]...> “‘

code:

```
    protected void map(LongWritable key, Text value, Context
        context)
            throws IOException, InterruptedException {

        context.write(new Text(key.toString()),value);

    }
protected void reduce(Text key, Iterable<Text> values,
   Context
            context) throws IOException, InterruptedException
                {
        StringBuilder v= new StringBuilder();
        for (Text value : values) {
            v.append(value).append('\t');
        }
        context.write(key,new Text(v.toString()));
    }
```

2. PageRank

The PageRank.java will use MapReduce to iteratively rank the pages until they converge.

How many iterations are needed to reach convergence

For me, convergence is defined as **1) the top 10 ranks don’t change for 3 consecutive iterations and 2) The rank values don’t change more than 0.01 in 3 consecutive iterations.**

For current dataset, it takes 18 iterations to converge.

map phase

input:

“‘ same as the output of preprocessing phase “‘

output:

“‘ <[node], info [degree] [adjacent node] [adjacent node]...> <[adjacent node], [node] [rank] [degree]> rank = 1 if first time <[adjacent node], [node] [rank] [degree]> same as up ... <[node], info 0> if node alone “‘

code:

```
public void map(LongWritable key, Text value, OutputCollector
   <Text, Text> output, Reporter reporter)
                throws IOException {

String line = value.toString();
String[] parts = line.split("[ \t]");

thisLink.set(parts[0]);
double rank = 0.0;
int degree = 0;
if (parts.length >= 2) {
        if (parts[1].length() < 10) { // A rank b c...
                rank = Double.parseDouble(parts[1]);
                degree = parts.length - 2;

                StringBuilder builder = new StringBuilder();
                if (parts.length >= 3) {
                        for (int i = 2; i < parts.length; i
                           ++) {
                                builder.append(parts[i]).
                                   append(" ");
                        }
                }
                outputValue.set("info " + degree + " "
                                + builder);
                output.collect(thisLink, outputValue);

                outputValue.set(thisLink + " "
                                + rank + " "
                                + degree);
                if (parts.length >= 3) {
                        for (int i = 2; i < parts.length; i
                           ++) {
                                outputKey.set(parts[i]);
                                output.collect(outputKey,
                                   outputValue);
                                reporter.incrCounter(Counters
                                   .INPUT_WORDS, 1);
                        }
                }
        } else { // A (no rank) b c ...
                rank = 1.0;
                degree = parts.length - 1;
```

```
38
39                  StringBuilder builder = new StringBuilder();
40                  for (int i = 1; i < parts.length; i++) {
41                          builder.append(parts[i]).append(" ");
42                  }
43                  outputValue.set("info " + degree + " "
44                                  + builder);
45                  output.collect(thisLink, outputValue);
46
47                  outputValue.set(thisLink + " "
48                                  + rank + " "
49                                  + degree);
50                  for (int i = 1; i < parts.length; i++) {
51                          outputKey.set(parts[i]);
52                          output.collect(outputKey, outputValue
                                );
53                          reporter.incrCounter(Counters.
                                INPUT_WORDS, 1);
54                  }
55          }
56  } else {// only A
57          rank = 1.0;
58
59          outputValue.set("info 0");
60          output.collect(thisLink, outputValue);
61  }
62
63                  }
```

reduce phase

input:

“‘ <[node], listinfo... [node]... info 0> the output of map phase combined together. “‘

output:

“‘ <[node], [rank] [adjacent node] [adjacent node]...> “‘

code:

```
1  public void reduce(Text key, Iterator<Text> values,
     OutputCollector<Text, Text> output, Reporter reporter)
2                   throws IOException {
3
4          double sum = 0.00;
5          String outlinks = "";
6
7          while (values.hasNext()) {
8                   String line = values.next().toString();
9                   String[] parts = line.split("[ \t]");
10
11                  if (!parts[0].equals("info")) {//Not info message
```

```
                        sum += Double.parseDouble(parts[1]) /
                            Double.parseDouble(parts[2]);
                } else {
                        if (parts.length > 2) {
                                for (int i = 2; i < parts.
                                    length; i++) {
                                        outlinks = outlinks.
                                            concat(parts[i] +
                                            " ");
                                }
                        }
                }
        }
        double factor = 0.85;
        sum = factor * sum + 1 - factor;
        sum = Math.round(sum * 10000.0) / 10000.0;
        outputValue.set(sum + " " + outlinks);
        output.collect(key, outputValue);
                }
```

3. sortRank

sort the nodes according to its rank.

map phase

input:

“‘ same as the last time of the reduce phase “‘

output:

“‘ <[rank], [node]> rank = 100 - rank, in order to find the top 10 “‘

code:

```
    public static class Map extends MapReduceBase implements
        Mapper<LongWritable, Text, Text, Text> {

                private final Text outputKey = new Text();
                private final Text outputValue = new Text();

                public void map(LongWritable key, Text value,
                    OutputCollector<Text, Text> output,
                    Reporter reporter)
                                throws IOException {

                        String line = value.toString();
                        String[] parts = line.split("[ \t]");
                        outputValue.set(parts[0]);
                        outputKey.set(String.valueOf(100.0 -
                            Double.parseDouble(parts[1])));
                        output.collect(outputKey, outputValue
                            );
                }
```

```
15          }
```

reduce phase

input:

“‘ <[rank], list[node]> “‘

output:

“‘ <[node], [rank]> “‘

code:

```
1     public void reduce(Text key, Iterator<Text> values,
          OutputCollector<Text, Text> output, Reporter reporter)
2       throws IOException {
3     while (values.hasNext()) {
4       double thekey = Math.round((100.0 - Double.parseDouble(
            key.toString())) * 10000.0) / 10000.0;
5       Text newkey = new Text(String.valueOf(thekey));
6       if(numOutput < 10) { //Number of displayed records
7           output.collect(newkey, values.next());
8           numOutput++;
9       } else {
10          values.next();
11      }
12
13    }
14  }
```

4. AddInfo

this class will output the related infomation about the dataset and the output.

input:

“‘ same as the output of preprocess “‘

output:

“‘ Info: Number of Node: [numNodes] Number of Edges: [numEdges] Average Degree: [numEdges/numNodes] Max Degree: [maxDegree] Min Degree: [minDegree] “‘

code:

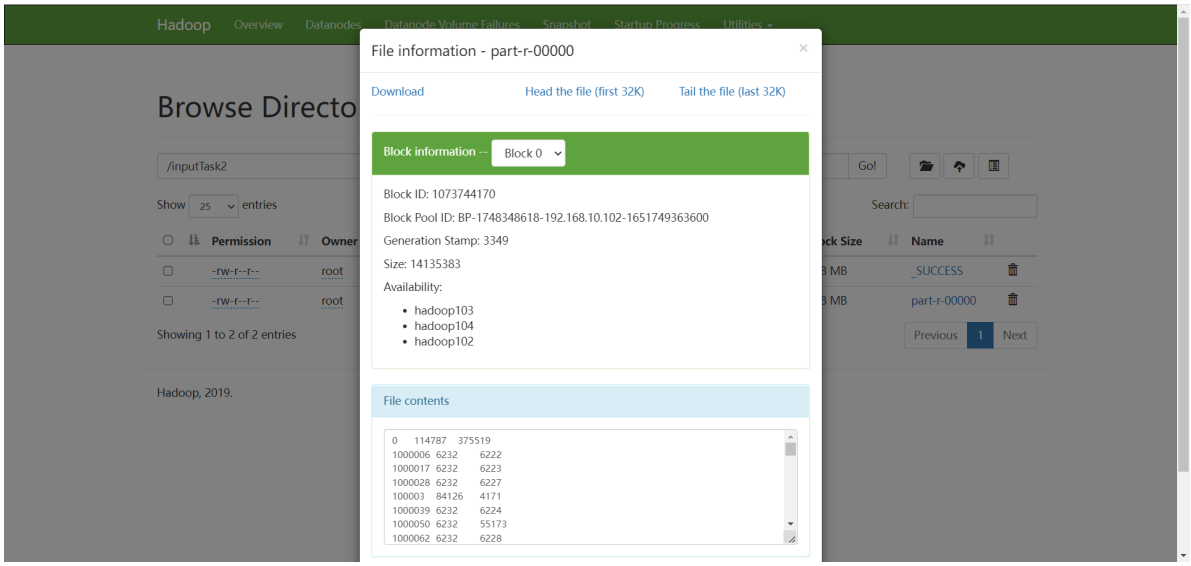
```
1   public void map(LongWritable key, Text value, OutputCollector
        <Text, Text> output, Reporter reporter)
2                   throws IOException {
3
4           String line = value.toString();
5           String[] parts = line.split("[ \t]");
6
7           int degree = parts.length - 1;
8
9           StringBuilder builder = new StringBuilder();
10          if (parts.length >= 3) {
11                  for (int i = 2; i < parts.length; i++) {
12                          builder.append(parts[i]).append(" ");
```

```
13                  }
14          }
15          outputKey.set("Info:");
16          outputValue.set(String.valueOf(degree));
17          output.collect(outputKey, outputValue);
18  }
19
20  public void reduce(Text key, Iterator<Text> values,
        OutputCollector<Text, Text> output, Reporter reporter)
21                  throws IOException {
22          while (values.hasNext()) {// key.toString().equals("Info:"))
23                  String line = values.next().toString();
24                  String[] parts = line.split("[ \t]");
25                  numNodes++;
26
27                  int degree = Integer.parseInt(parts[0]);
28                  numEdges += degree;
29                  if(degree<minDegree)
30                          minDegree = degree;
31                  if(degree > maxDegree)
32                          maxDegree = degree;
33          }
34          if(key.toString().equals("Info:")){
35                  Text out = new Text();
36                  out.set("\nNumber of Node: "+numNodes+ "\
                        nNumber of Edges: "+ numEdges + "\nAverage
                         Degree: " + numEdges/numNodes + "\nMax
                        Degree: "+ maxDegree + "\nMin Degree: "+
                        minDegree);
37                  output.collect(key, out);
38          } else {
39                  output.collect(key, values.next());
40          }
41  }
```

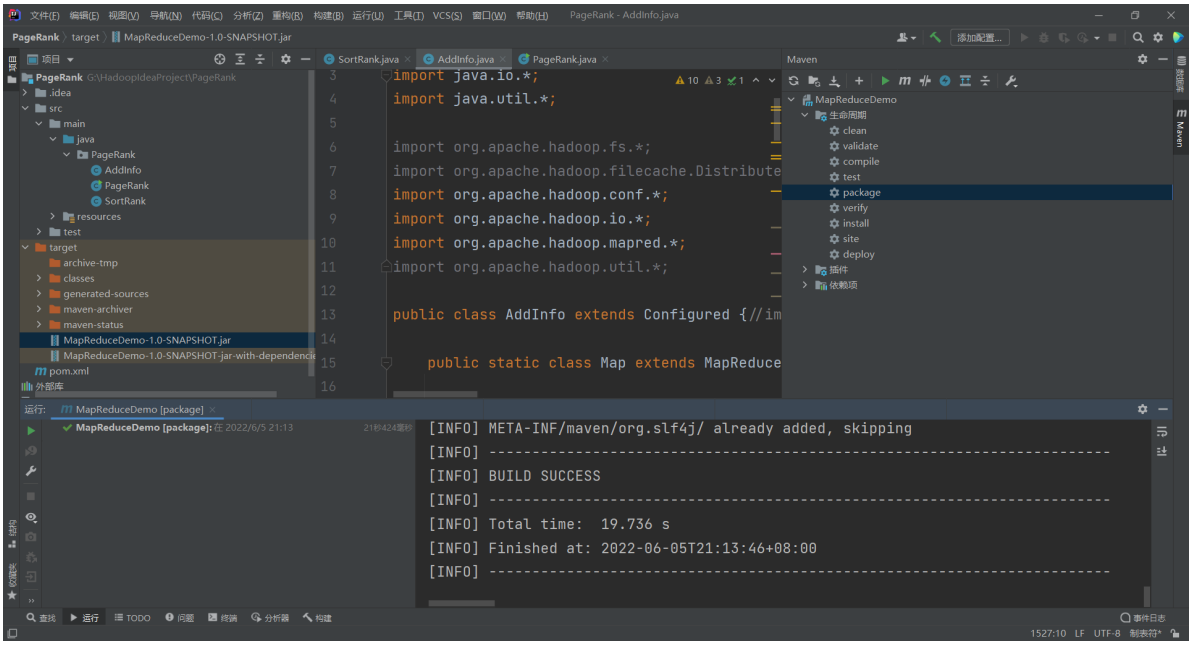
5.2.2 results

1. preprocess the dataset

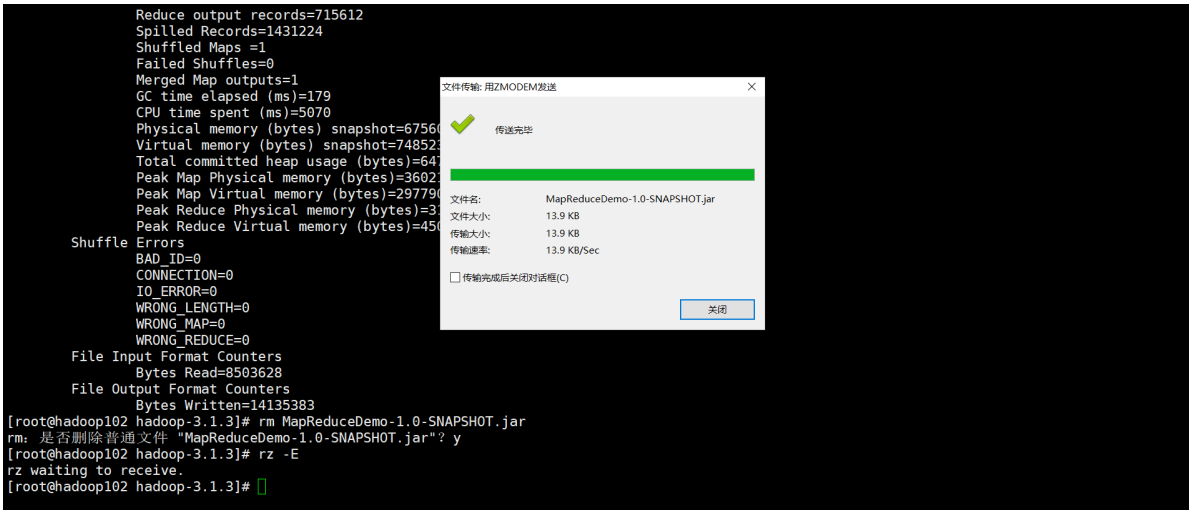
```
[root@hadoop102 hadoop-3.1.3]# hadoop jar MapReduceDemo-1.0-SNAPSHOT.jar example.org.mapreduce.lab.driver /input/PP-Decagon_ppi.csv /inputTask2
2022-06-05 21:22:06,495 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 21:22:06,884 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and ex
ecute your application with ToolRunner to remedy this.
2022-06-05 21:22:06,897 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1654434
395401_0002
2022-06-05 21:22:06,970 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 21:22:07,074 INFO input.FileInputFormat: Total input files to process : 1
2022-06-05 21:22:07,107 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 21:22:07,147 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 21:22:07,168 INFO mapreduce.JobSubmitter: number of splits:1
2022-06-05 21:22:07,264 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 21:22:07,288 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654434395401_0002
2022-06-05 21:22:07,289 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 21:22:07,434 INFO conf.Configuration: resource-types.xml not found
2022-06-05 21:22:07,435 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2022-06-05 21:22:07,891 INFO impl.YarnClientImpl: Submitted application application_1654434395401_0002
2022-06-05 21:22:07,953 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654434395401_0002/
2022-06-05 21:22:07,953 INFO mapreduce.Job: Running job: job_1654434395401_0002
```



2. compile program to Jar file



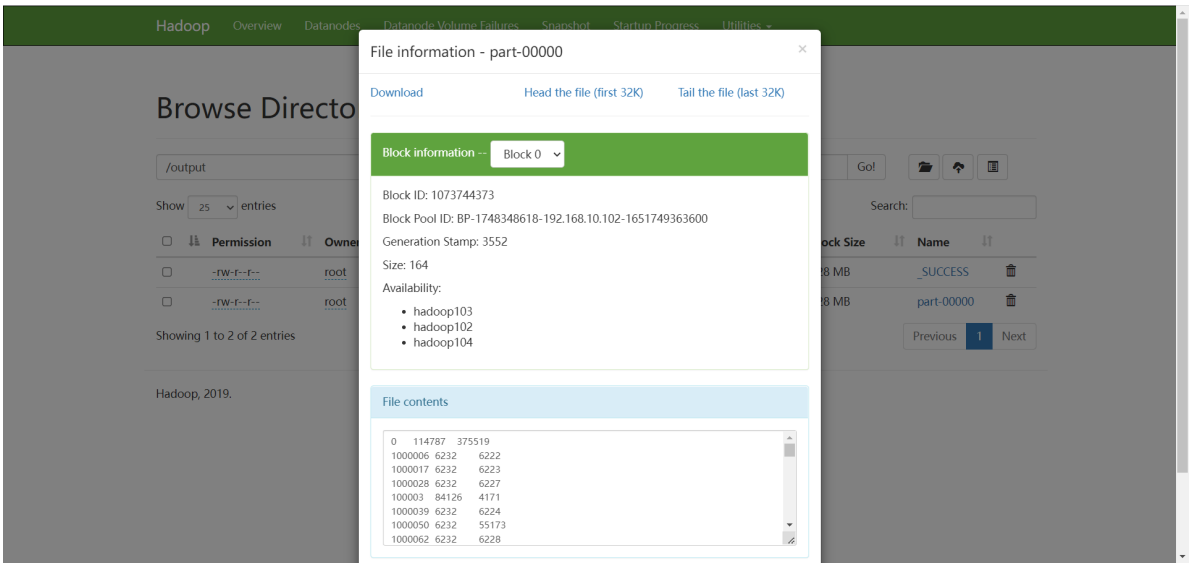
3. load Jar file to VM

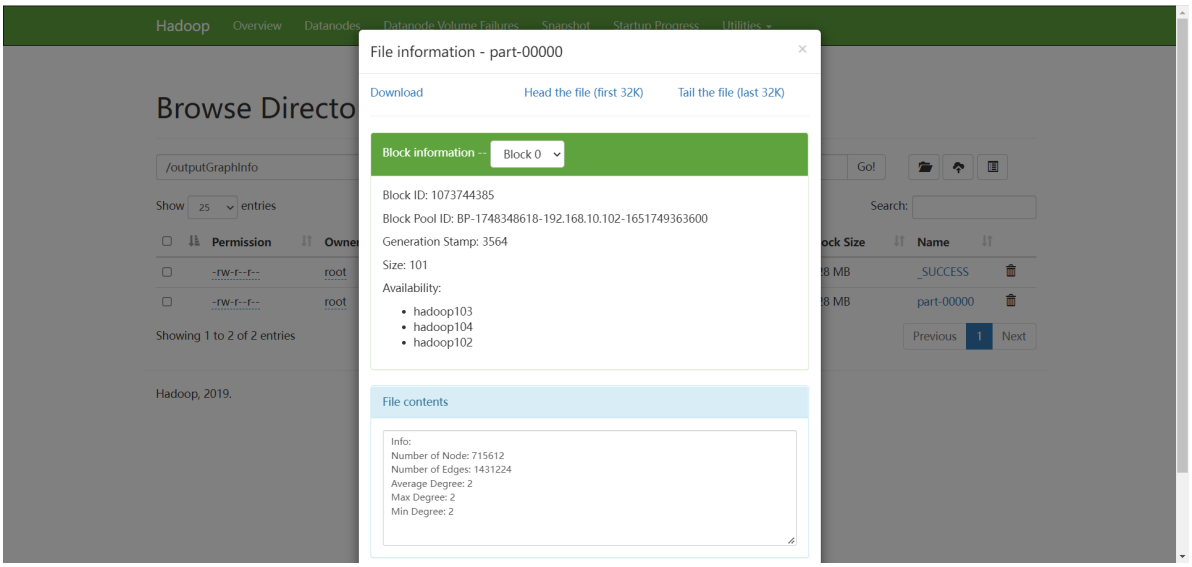


4. run program in VM

```
[root@hadoop102 hadoop-3.1.3]# hadoop jar MapReduceDemo-1.0-SNAPSHOT.jar PageRank.PageRank /inputTask2/part-r-00000 /output
2022-06-05 21:24:29,395 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 21:24:29,515 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 21:24:29,769 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and ex
ecute your application with ToolRunner to remedy this.
2022-06-05 21:24:29,786 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1654434
395401_0003
2022-06-05 21:24:29,848 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 21:24:29,947 INFO mapred.FileInputFormat: Total input files to process : 1
2022-06-05 21:24:29,972 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 21:24:30,008 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 21:24:30,029 INFO mapreduce.JobSubmitter: number of splits:2
2022-06-05 21:24:30,121 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 21:24:30,146 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654434395401_0003
2022-06-05 21:24:30,146 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 21:24:30,275 INFO conf.Configuration: resource-types.xml not found
2022-06-05 21:24:30,275 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2022-06-05 21:24:30,318 INFO impl.YarnClientImpl: Submitted application application_1654434395401_0003
2022-06-05 21:24:30,343 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654434395401_0003/
2022-06-05 21:24:30,344 INFO mapreduce.Job: Running job: job_1654434395401_0003
```

result:





5.3 Graph Connected Components

5.3.1 Setup

Given a graph, this algorithm identifies the connected components. In the following, a connected component will be called as "cluster". We have tried to implement the "The Alternating Algorithm" proposed in the paper "Connected Components in MapReduce and Beyond: https://dl.acm.org/doi/10.1145/2670979.2670997". Below, we have reported the pseudo-code of the algorithm.

Algorithm 1 The Alternating Algorithm

INPUT: Edges (u, v) as a set of key-value pairs $\langle$u; v$\rangle$.
INPUT: A unique label ℓ_v for every node v $\in$ V.

```
repeat
    Large-Star
    Small-Star
until Convergence
```

Algorithm 2 The Large-Star operation

```
procedure MAP( u; v )
    Emit ⟨u; v⟩
    Emit ⟨v; u⟩
end procedure
procedure REDUCE( u; Γ(u) )
    m ← arg min_{v∈Γ+(u)} ℓ_v
    Emit ⟨v; m⟩ ∀v where ℓ_v > ℓ_u
end procedure
```

Algorithm 3 The Small-Star operation

```
procedure MAP( u; v )
    if ℓ_v ≤ ℓ_u then
        Emit ⟨u; v⟩
    else
        Emit ⟨v; u⟩
    end if
end procedure
procedure REDUCE( u; N ⊆ Γ(u) )
    m ← arg min_{v∈N∪{u}} ℓ_v
    Emit ⟨v; m⟩ ∀v ∈ N
end procedure
```

1. Initialization, to transform data into targeting format.
2. Set lists of comparable class in order to sort custom-made key.
3. Perform large star and small star operations.
4. Terminate when convergence
5. Termination, gather nodes of each connected components.
6. check if duplicated.
7. translate cluster type to text type.

5.3.2 Implementation

Below, it is shown a demonstration of usage of the ConnectedComponents class that allows us to run the connected component algorithm on graph. This code simply creates a

ConnectedComponents object, taking as input the graph and the output folder. Invoking the run method, the algorithm will produce an hdfs file for each Reducer task, invoked by MapReduce framework, into the output folder. These files will contain the clusters found in the input graph.

```
1  package app;
2  import pad.ConnectedComponents;
3  public class App
4  {
5      public static void main( String[] args ) throws
           Exception
6      {
7              // Create ConnectedComponents object giving an
                   adjacency/cluster
8              // list rappresenting the graph and the output folder as input
9              ConnectedComponents cc = new
                   ConnectedComponents( new Path("graph.txt")
                   , new Path("out") );
10
11             // Run all the Jobs necessary to compute the result
12             if ( !cc.run() )
13                     System.exit( 1 );
14             System.exit( 0 );
15     }
16 }
```

5.3.2.1 Input

The input file can contain:

- the adjacency list of the graph, i.e. multiple lines in the following format:

  $< NodeID >< TAB >< Neighborhood >$

  $< NodeID >$: is a unique integer ID corresponding to an unique node of the graph.

  $< Neighborhood >$: is a comma separated list of increasing unique IDs corresponding to the nodes of the graph that are linked to $< NodeID >$.

- the clique list of the graph, i.e. multiple lines in the following format:

  $< NodeID1 >< SPACE >< NodeID2 >< SPACE > ... < SPACE >< NodeIDN >$

  indicating that all the nodes, $< NodeID1 > ... < NodeIDN >$, are linked to each other.

5.3.2.2 Output

The program will create a folder where the clusters found are stored in a star list format.
In particular the output files produced by the Reducer tasks are formatted by the
$SequenceFileOutputFormat < pad.ClusterWritable, org.apache.hadoop.io.NullWritable >$,

where $pad.ClusterWritable$ represents an array of integers and is serialized writing on the output file its size and then its elements. Therefore, each output file has the following format:

$< Cluster1 >< Cluster2 > ... < ClusterK >$

where each Cluster is a sequence of increasing numbers which they compound a connected component of the input graph.

5.3.2.3 Design

Here it is shown the algorithm of our implementation of Alternating Algorithm.

Algorithm 4 Our implementation of the Alternating Algorithm

INPUT: G = (V, E) represented with an adjacency/cluster list.
INPUT: A unique and positive label ℓ_v for every node v ∈ V.

```
1: Initialization_Phase
2: repeat
3:     Large-Star
4:     Small-Star
5: until Convergence
6: Termination_Phase
7: Check_Phase
```

Where:

- Initialization_Phase → Transform the adjacency/clique list in a list of edges $< NodeID >< TAB >< NeighborID >$
- Termination_Phase → Transform the list of edges into sets of nodes (cluster files, i.e. star list)
- Check_Phase → Verify that no clusters is malformed

When we are talking about huge graph, usually they are represented with an adjacency list or a clique list. In fact, both of these format can be represented in a more compacted way than the sparse matrix format.

Since Algorithm 1 expects as input a set of key-values pairs $< u; v >$, for what we have said above, we have decided to introduce an Initialization Phase that transforms the adjacency/cliques list into a edges list $< NodeID >< TAB >< NeighbourID >$. In this phase, we also capture the nodes and clique number of the input.

As you can see from Algorithm 4, after the Initialization Phase, takes place the cycle suggested by Algorithm 1 where the Large-Star and Small-Star operations are called sequentially until the graph does not reach a convergent configuration.

When we finally reach this point, we still have the edges list $< NodeID >< TAB >< NeighbourID >$. So we need to identify the clusters and print them in a usable format. This operation is performed during the Termination Phase. The counting of nodes and clusters numbers is performed again in order to compare these values with the initial ones.

Finally, thanks to the Check Phase, we verify that no clusters is malformed, i.e. every node belonging to a cluster cannot be present in another one, but must be unique.

1. Initialization

Thanks to this format, the nodes number is easily calculated counting the input file rows. We use InitializationMapperAdjacent.java as Mapper and zero Reducer. In the Mapper, we read a line and we split it by the <TAB> character. Then, for each adjacent node, we produce the pair <NodeID, AdjacentNodeID> only if NodeID > AdjacentNodeID. In fact, whether the graph is directed or not, with this input format, we assume that every link is an undirected link. Mainly the reason is that, in the following operations, we only need the edge direction from the larger to smaller node.

if a node has no neighbours, it is emitted the pair <NodeID, -1>. In this way, when in the following phases we recognize this information, we just bring it forward, until the Termination Phase where we emit a cluster formed only by that NodeID.

input:

“‘ [src]<tab>[dst] “‘

output:

“‘ <[node], [adjacent node]> while NodeID > AdjacentNodeID <[node], -1> while node alone “‘

code:

```
public void map( LongWritable , Text value , Context context )
   throws IOException , InterruptedException
        {
                // Read line.
                String line = value.toString();

                // Increment the number of nodes, since the input file presents a
                    new node in each line.
                context.getCounter( UtilCounters.
                   NUM_INITIAL_NODES ).increment( 1 );

                // Split the line on the tab character.
                String[] userID_neighbourhood = line.split( "
                   \t" );

                // Extract the nodeID.
                nodeID.set( Integer.parseInt(
                   userID_neighbourhood[0] ) );

                // If the node is alone.
                if ( userID_neighbourhood.length == 1 )
                {
                        // NeighbourID is set to minus one, to indicate that the
                            node is alone.
                        context.write( nodeID , MINUS_ONE );
                        return;
                }

                // Split by "," to find the list of neighbours of nodeID.
                String[] neighbours = userID_neighbourhood
                   [1].split( "," );
```

```
25
26                      // Emit the pair <nodeID, neighbourID> for each neighbours.
27                      for (String neighbour : neighbours) {
28                              neighbourID.set(Integer.parseInt(
                                    neighbour));
29                              // only if nodeID > neighbourID
30                              if (nodeID.get() > neighbourID.get())
31                                      context.write(nodeID,
                                            neighbourID);
32                      }
33              }
```

2. Set lists of comparable class in order to sort custom-made key.

During the implementation of the algorithm, we heavily use a technique that allows the programmer to control the order in which the values show up during the Reducer phase, i.e. the comparable interface is designed to do so. In fact in the Hadoop's implementation of MapReduce, this feature is not provided by the framework. So we need several custom-made class in order to exploit this feature.

First of all, we need a composite key that contains the necessary information to perform the sorting and we have to know how compare those composite keys during the sorting phase. NodesPairWritable.java class is a writable and comparable data structure used to wrap two nodes into a key:

- NodeID: is the natural key used for joining purposes;
- NeighbourID: is the value for which we want to sort;

and contains the compareTo method that use both nodes during the sorting phase.

Second, we need a custom partitioner: NodePartitioner.java. Thanks to this class, the composite key will be sent to a Reducer only considering the NodeID, i.e. a given NodeID will be sent to only one Reducer.

Finally, we have to ensure that when the Reducer read the map output records from local disk, those records are combined by NodeID. The NodeGroupingComparator.java class specify how to preform this grouping process.

The most important usage of this technique is during the Small-Star and Large-Star operations. In fact, as we can see from Algorithms 2 and 3 respectively at line 6 and 9, we need to obtain the minimum neighbour. Thanks to the secondary sort technique, the Reducer will receive for a given NodeID all its neighbours sorting in ascending order. So, it is sufficient to compare this NodeID with its first neighbour in order to identify the minimum label. Then, we use it during the Termination Phase, since we want to store the set of nodes that compose a cluster in ascending order.

Related classes:

(Not every details are shown here)

```
1  **NodesPairWritable.java**
2
3  package GCC;
4  public class NodesPairWritable implements WritableComparable<
       NodesPairWritable>
5  {
```

```
        public Integer NodeID = -1;
        public Integer NeighbourID = -1; // default to -1 if node alone.

        // Deserializes the array. Read the data out in the order it is written.
        public void readFields( DataInput in ) throws
           IOException{...}

        // Serializes this array. Write the data out in the order it is read.
        public void write( DataOutput out ) throws
           IOException{...}

        // Convert the object into a string.
        public String toString(){...}

    // Compare this object with other one of its kind.
    // First compare NodeID. if same, compare NeighbourID.
    // Return: 0 if identical, -1 if this smaller, 1 if this greater
        public int compareTo( NodesPairWritable other ){...}

        // calculate the hash code using NodeID and NeighbourID
    public int hashCode(){...}

        // Check if two objects are equals.
    // Return: True if equal, False if not equal.
    public boolean equals( Object other ){...}
}
```

```
**NodePartitioner.java**

```java
package GCC;

public class NodePartitioner extends Partitioner<
   NodesPairWritable, IntWritable>
{
        // Choose the Reducer identifier to which send the record using only the
           NodeID information
    public int getPartition( NodesPairWritable pair,
       IntWritable _, int numPartitions ){
        return pair.NodeID % numPartitions;
    }
}
```

NodeGroupingComparator.java

```java
```
```

```java
package GCC;

public class NodeGroupingComparator extends
    WritableComparator
{
        // Initializes a new instance of the NodeGroupingComparator class.
        protected NodeGroupingComparator(){
        super( NodesPairWritable.class, true );
    }

        // Compare two keys read from the mapper output records, only looking to the
            first component, i.e NodeID.
    // Return: 0 if identical, -1 if this smaller, 1 if this greater
        public int compare( WritableComparable key1,
            WritableComparable key2 ){...}
}
```

3. Large-star Small Star

Since the structure of two operations is very similar, we have decided to use the following java class files: StarDriver.java, StarMapper.java, StarCombiner.java, StarReducer.java and change their behaviour using an external variable, i.e. the Mapper and Reducer extract this variable during the setup and than they act accordingly depending on its value. Then, we just follow the exactly behaviour described in Algorithms 2 and 3, as you can see from the code we wrote for the Mapper and Reducer.

After a Large-Star or Small-Star operation, it is possible that some links are replicated.

For this reason, we added a Combiner phase (StarCombiner.java) that is responsible of reducing the amount of data written to local storage, merged, sorted and sent over the network.

The Hadoop framework does not ensure how many times a Combiner is called in action, but we know that, if it happens, is after the data sorting:

- after in-memory sort and before writing data to disk;
- after the merging and sorting phases.

Therefore, knowing that the data is sorted, we can exploit the secondary sort and delete the duplicates simply storing the last NeighbourId emitted and skipping all the following neighbours with the same identifier.

Related classes:

(Not every details are shown here)

```
**StarDriver.java**

```java
package GCC;
// Driver of the Job responsible for executing the Small-Star or Large-Star operation on the
    input graph.
public class StarDriver extends Configured implements Tool
{
        // The StarDriver can be of type Large-StarDriver or Small-StarDriver
        public enum StarDriverType { LARGE, SMALL };
```
```

```
10
11          public StarDriver( StarDriverType type, Path input,
                Path output, int iteration, boolean verbose )
12          {
13          this.type = type;
14                  this.title = type.equals( StarDriverType.
                        SMALL ) ? "Small-Star" + iteration : "
                        Large-Star" + iteration;
15                  ...
16          }
17
18          public int run( String[] args ) throws Exception
19          {
20                  //set different job accordingly.
21          conf.set( "type", this.type.toString() );
22                  Job job = new Job( conf, this.title );
23                  ...
24
25                  // Set up the private variable looking to the counter value
26                  this.numChanges = job.getCounters().
                        findCounter( UtilCounters.NUM_CHANGES ).
                        getValue();
27          }
28
29          // Return the number of changes occurred during the operation Small-Star or
                Large-Star.
30          public long getNumChanges(){
31          return this.numChanges;
32      }
33  }
34
35  ```
36
37  **StarMapper.java**
38
39  input:
40
41  ```
42  <[node], [adjacent node]>
43  ```
44
45  output:
46
47  ```
48  <pair{[node], [adjacent node]}, [node]>
49  <pair{[node], [adjacent node]}, [adjacent node]>
50  # the value depends on the type of star operation and the
        comparation of NodeID and NeighbourID
51  ```
```

code:

```java
package GCC;

public class StarMapper extends Mapper<IntWritable, IntWritable, NodesPairWritable, IntWritable>
{
        // Extract type variable from the context configuration.
        // Based on this value, this Mapper will behave as a Small-Star Mapper or Large-Star Mapper
        public void setup( Context context ){
        smallStar = context.getConfiguration().get( "type" ).equals( "SMALL" );
    }

        // If Large-Star, emits the pairs <u,v> and <v,u>.
        // If Small-Star, emits the pair <max(u,v), min(u,v)>.
        public void map( IntWritable nodeID, IntWritable neighbourID, Context context ) throws IOException, InterruptedException
        {
                // if the node is alone, emit it like is it in order to keep that information
                if ( neighbourID.get() == -1 ){ context.write( pair, neighbourID ); }


                // If we are running Small-Star, we emit only when the neighbourID is smaller than nodeID
                if ( smallStar ){...}
                // If we are running Large-Star, we always emit: <NodeID; NeighbourID> and <NeighbourID; NodeID>
                else {...}
        }
}
```

StarReducer.java

input:

```
<pair{[node], [adjacent node]}, list{[node]}>
<pair{[node], [adjacent node]}, list{[adjacent node]}>
# the value depends on the type of star operation
```

output:

```
<[node], [minNode]>
<[adjacent node], [minNode]>
# the value depends on the type of star operation and the comparation of NeighbourID and the NodeID
```

code:

```java
package GCC;

public class StarReducer extends Reducer<NodesPairWritable, IntWritable, IntWritable, IntWritable>
{
        // same setup as StarMapper.java
        public void setup( Context context )
        {
                smallStar = context.getConfiguration().get( "type" ).equals( "SMALL" );
        }

        /* The neighbours are sorted, thanks to the NodesPairWritable.java, we know that the minimum node is either the NodeID or the first neighbour. We call it MinNodeID.
        For each neighbour, we produce the pairs <NeighbourID, MinNodeID> and <MinNodeID, NeighbourID> :
        Small-Star: -   always;
        Large-Star: -   only when NeighbourID is greater than NodeID */
        public void reduce( NodesPairWritable pair, Iterable<IntWritable> neighbourhood, Context context )
        throws IOException, InterruptedException
        {
                // This means that the nodeID is isolated, so we emit it unchanged
                if ( pair.NeighbourID == -1 ){...}

                // We know that the first element contains the neighbour node with the minimum label. We just need to compare it with the node id.
                minNodeID.set( Math.min( pair.NodeID, pair.NeighbourID ) );

                // If we are running Small-Star, we need to connect this node to the minimum neighbours
                if ( smallStar && ( pair.NodeID != minNodeID.get() ) ){...}

```

```
                    // same impossible value
                    int lastNodeSeen = -2;
                    for ( IntWritable neighbour : neighbourhood )
                    {
                            // Skip the duplicate nodes.
                            if ( neighbour.get() == lastNodeSeen
                                )
                                    continue;

                            // If we are running Small-Star, we always emit the
                                neighbours except when it is the minNodeID
                            // If we are running Large-Star, we emit only when the
                                neighbourID is greater than nodeID
                            boolean cond = ( smallStar ? (
                                neighbour.get() != minNodeID.get()
                                 ) : ( neighbour.get() > pair.
                                NodeID ) );
                            if ( cond ){...}

                            // Store the last neighbourId that we have processed.
                            lastNodeSeen = neighbour.get();
                    }

                    // If the NodeID has not the minimum label means that the
                        produced pairs will be different,
                    // so we increment the number of changes by the number of
                        produced pairs
                    if ( pair.NodeID != minNodeID.get() ){...}
            }
}
```

```
**StarCombiner.java**

input:

<[node], [minNode]>
<[adjacent node], [minNode]>
# the value depends on the type of star operation and the
   comparation of NeighbourID and the NodeID

output:

# same, but not duplicated.
```

```
code:

```java
package GCC;

/** Combiner task of the \see StarDriver Job. */
public class StarCombiner extends Reducer<NodesPairWritable,
    IntWritable, NodesPairWritable, IntWritable>
{

        // It reduces the number of duplicates that are emit by the StarMapper.
        public void reduce( NodesPairWritable pair, Iterable<
            IntWritable> neighbourhood, Context context )
            throws IOException, InterruptedException
        {
                // set a impossible value to check if seen.
                int lastNodeSeen = -2;
                for ( IntWritable neighbour : neighbourhood )
                {
                        // 1. Skip the duplicate nodes.
                        // 2. Emit the pair
                        // 3. Store the last neighbourId that we have
                            processed.
                }
        }
}
```
```

4. Convergence

The number of modified edges can be counted in this kind of situation that whether the minimum neighbour is the same node as the node we are analyzing.

If it's not the same, the number of modified edges is equal to the number of emitted edges.

In conclusion, the Convergence condition can be presented in an equation as following:

$$LargeStar.NumChanges + SmallStar.NumChanges = 0$$

This means that only when both the operations: Large-Star and Small-Star do not modify the graph, then we can conclude that the graph has reached its convergence point.

5. Termination phase

In order to realize this phase, we used three java class files: TerminationDriver.java, TerminationMapper.java, and TerminationReducer.java. The first one is used to configure the Termination Job that has the other two class files respectively as Mapper and Reducer. This Job expects a pairs list of integers nodes as input, and produces an output formatted as following:

"‘ Cluster_1 Cluster_2 ... Cluster_K Each cluster is a list of incresing NodeID, forming a connected component of the input graph. "‘

For each pair <u, v>, this Mapper emits the pair <u, v> if ‘u < ‘v, otherwise <v, u>. In this way, the Reducer will receive a cluster for each key. In fact, if a connected component has reached its convergence point, all nodes belonging to this sub-graph are connected to the minimum node.

Therefore, in the Reducer phase, we receive the minimum NodeID of a cluster and all the nodes belonging to it (sorted in ascending order thanks to the comparable technique). What remains to be done is just adding these nodes to a ClusterWritable object and emitting it, so producing a star list. As we said, during this phase, the counting of nodes and clusters number is performed again in order to compare these values with the initial ones.

Related classes

(Not every details are shown here)

```
**StarDriver.java**

```java
package GCC;
// Driver of the Job responsible for executing the Small-Star or Large-Star operation on the input graph.
public class StarDriver extends Configured implements Tool
{
        // The StarDriver can be of type Large-StarDriver or Small-StarDriver
        public enum StarDriverType { LARGE, SMALL };

        public StarDriver( StarDriverType type, Path input, Path output, int iteration, boolean verbose )
        {
        this.type = type;
                this.title = type.equals( StarDriverType.SMALL ) ? "Small-Star" + iteration : "Large-Star" + iteration;
                ...
        }

        public int run( String[] args ) throws Exception
        {
                //set different job accordingly.
        conf.set( "type", this.type.toString() );
                Job job = new Job( conf, this.title );
                ...

                // Set up the private variable looking to the counter value
                this.numChanges = job.getCounters().findCounter( UtilCounters.NUM_CHANGES ).getValue();
        }

        // Return the number of changes occurred during the operation Small-Star or Large-Star.
        public long getNumChanges(){
        return this.numChanges;
    }
}

```
```

```
```

StarMapper.java

input:

```
<[node], [adjacent node]>
```

output:

```
<pair{[node], [adjacent node]}, [node]>
<pair{[node], [adjacent node]}, [adjacent node]>
# the value depends on the type of star operation and the comparation of NodeID and NeighbourID
```

code:

```java
package GCC;

public class StarMapper extends Mapper<IntWritable, IntWritable, NodesPairWritable, IntWritable>
{
        // Extract type variable from the context configuration.
        // Based on this value, this Mapper will behave as a Small-Star Mapper or Large-Star Mapper
        public void setup( Context context ){
        smallStar = context.getConfiguration().get( "type" ).equals( "SMALL" );
    }

        // If Large-Star, emits the pairs <u,v> and <v,u>.
        // If Small-Star, emits the pair <max(u,v), min(u,v)>.
        public void map( IntWritable nodeID, IntWritable neighbourID, Context context ) throws IOException, InterruptedException
        {
                // if the node is alone, emit it like is it in order to keep that information
                if ( neighbourID.get() == -1 ){ context.write( pair, neighbourID ); }


                // If we are running Small-Star, we emit only when the neighbourID is smaller than nodeID
```

```
                    if ( smallStar ){...}
                    // If we are running Large-Star, we always emit: <NodeID; NeighbourID> and <NeighbourID; NodeID>
                    else {...}
            }
}
```

StarReducer.java

input:

```
<pair{[node], [adjacent node]}, list{[node]}>
<pair{[node], [adjacent node]}, list{[adjacent node]}>
# the value depends on the type of star operation
```

output:

```
<[node], [minNode]>
<[adjacent node], [minNode]>
# the value depends on the type of star operation and the comparation of NeighbourID and the NodeID
```

code:

```java
package GCC;

public class StarReducer extends Reducer<NodesPairWritable, IntWritable, IntWritable, IntWritable>
{
        // same setup as StarMapper.java
        public void setup( Context context )
        {
                smallStar = context.getConfiguration().get( "type" ).equals( "SMALL" );
        }

        /* The neighbours are sorted, thanks to the NodesPairWritable.java, we know that the minimum node is either the NodeID or the first neighbour. We call it MinNodeID.
        For each neighbour, we produce the pairs <NeighbourID, MinNodeID> and <MinNodeID, NeighbourID> :
        Small-Star: -   always;
```

```
117             Large-Star: -   only when NeighbourID is greater than
                    NodeID */
118             public void reduce( NodesPairWritable pair, Iterable<
                    IntWritable> neighbourhood, Context context )
                    throws IOException, InterruptedException
119             {
120                     // This means that the nodeID is isolated, so we emit it unchanged
121                     if ( pair.NeighbourID == -1 ){...}
122
123                     // We know that the first element contains the neighbour node with
                            the minimum label. We just need to compare it with the node id.
124                     minNodeID.set( Math.min( pair.NodeID, pair.
                            NeighbourID ) );
125
126                     // If we are running Small-Star, we need to connect this node to
                            the minimum neighbours
127                     if ( smallStar && ( pair.NodeID != minNodeID.
                            get() ) ){...}
128
129                     // same impossible value
130                     int lastNodeSeen = -2;
131                     for ( IntWritable neighbour : neighbourhood )
132                     {
133                             // Skip the duplicate nodes.
134                             if ( neighbour.get() == lastNodeSeen
                                    )
135                                     continue;
136
137                             // If we are running Small-Star, we always emit the
                                    neighbours except when it is the minNodeID
138                             // If we are running Large-Star, we emit only when the
                                    neighbourID is greater than nodeID
139                             boolean cond = ( smallStar ? (
                                    neighbour.get() != minNodeID.get()
                                     ) : ( neighbour.get() > pair.
                                    NodeID ) );
140                             if ( cond ){...}
141
142                             // Store the last neighbourId that we have processed.
143                             lastNodeSeen = neighbour.get();
144                     }
145
146                     // If the NodeID has not the minimum label means that the
                            produced pairs will be different,
147                     // so we increment the number of changes by the number of
                            produced pairs
148                     if ( pair.NodeID != minNodeID.get() ){...}
149             }
150     }
151     ```
```

```
**StarCombiner.java**

input:

```
<[node], [minNode]>
<[adjacent node], [minNode]>
# the value depends on the type of star operation and the
    comparation of NeighbourID and the NodeID
```

output:

```
# same, but not duplicated.
```

code:

```java
package GCC;

/** Combiner task of the \see StarDriver Job. */
public class StarCombiner extends Reducer<NodesPairWritable,
    IntWritable, NodesPairWritable, IntWritable>
{

        // It reduces the number of duplicates that are emit by the StarMapper.
        public void reduce( NodesPairWritable pair, Iterable<
            IntWritable> neighbourhood, Context context )
            throws IOException, InterruptedException
        {
                // set a impossible value to check if seen.
                int lastNodeSeen = -2;
                for ( IntWritable neighbour : neighbourhood )
                {
                        // 1. Skip the duplicate nodes.
                        // 2. Emit the pair
                        // 3. Store the last neighbourId that we have
                            processed.
                }
        }
}
```

6. Check phase

In the Termination Phase, we have made sure to not store duplicate nodes within a cluster, exploiting the secondary sorting property. As we said, a cluster is not legal if one of its node is present in another cluster. So in order to verify if all the resulting clusters are well-formed,
```

it is sufficient to check that all nodes are unique. If a node is found twice, for what said above, it is surely present in two different clusters.

As usual, we need a class to configure the Job: CheckDriver.java. The Mapper (CheckMapper.java) emits all nodes of the clusters. The Reducer (CheckReducer.java) increments an error counter if receives a given node identifier more that one time.

To be precise, we should check if all the nodes of input graph are present in the final result. During the steps of the algorithm, a node is never added. Besides, as we said, we ensure that there are no duplicated nodes.

Therefore, in order to verify the property, it is necessary to check that the nodes number of the input graph is equal to the one of the final results. These two counting are calculated respectively in the Initialization and Termination Phases. We have chosen to simply report this two values to the final user that will judge if to keep or not the obtained result.

Related classes:

(Not every details are shown here)

```
**CheckDriver.java**
```java
package GCC;

public class CheckDriver extends Configured implements Tool
{
        public CheckDriver( Path input, boolean verbose ){
                ...
        }

        public int run( String[] args ) throws Exception
        {
                ...

                // Set up the private variable looking to the counter value
                this.testOk = ( job.getCounters().findCounter
                   ( UtilCounters.NUM_ERRORS ).getValue() ==
                   0 );

                // Delete the output folder ( we did not write on it )
                FileSystem.get( conf ).delete( input.suffix("
                   _check"), true  );

                return 0;
        }

        // Return False if the checking found malformed, True otherwise.
        public boolean isTestOk(){
                return this.testOk;
        }
}
```
```

```
**CheckMapper.java**

input:

```
<[cluster], NULL>
```

output:

```
<[node], NULL>
```

code:

```java
package GCC;

public class CheckMapper extends Mapper<ClusterWritable,
    NullWritable, IntWritable, NullWritable>
{
        // For each cluster, it emits all its nodes.
        public void map( ClusterWritable cluster,
            NullWritable _, Context context ) throws
            IOException, InterruptedException
        {
                for ( Integer node : cluster ){...}
        }
}
```
```

7. Translate phase

Because the output is in the format of sequence file, which is binary, so we need another step to translate the output into something human can read.

TranslatorDriver.java makes it possible to translate the result files in text and look which nodes form the connected components.

Related class:

```
**TranslatorDriver**

```java
package GCC;

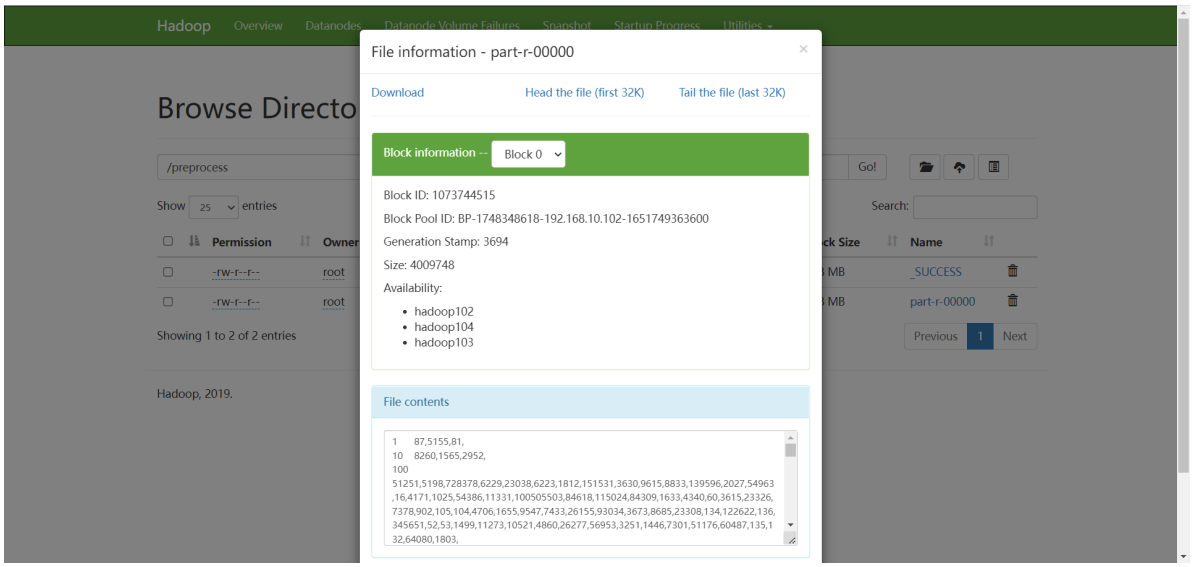
public class TranslatorDriver extends Configured implements
    Tool
{
        public TranslatorDriver( TranslationType type, Path
            input, Path output ){...}
```
```

```
9
10          public int run( String[] args ) throws Exception
11          {
12                  ...
13          // simply set the Mapper as the initial Mapper.class
14                  job.setMapperClass( Mapper.class );
15          ...
16          }
17  }
18  ```
```

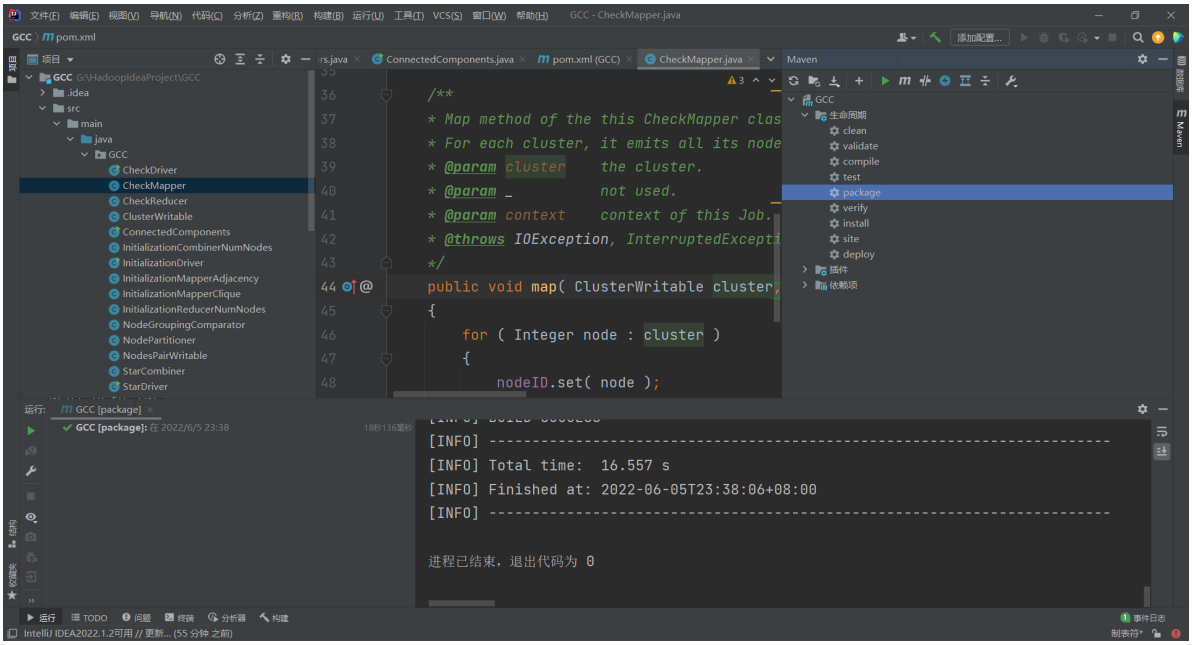
5.3.3 Results

1. preprocess the dataset

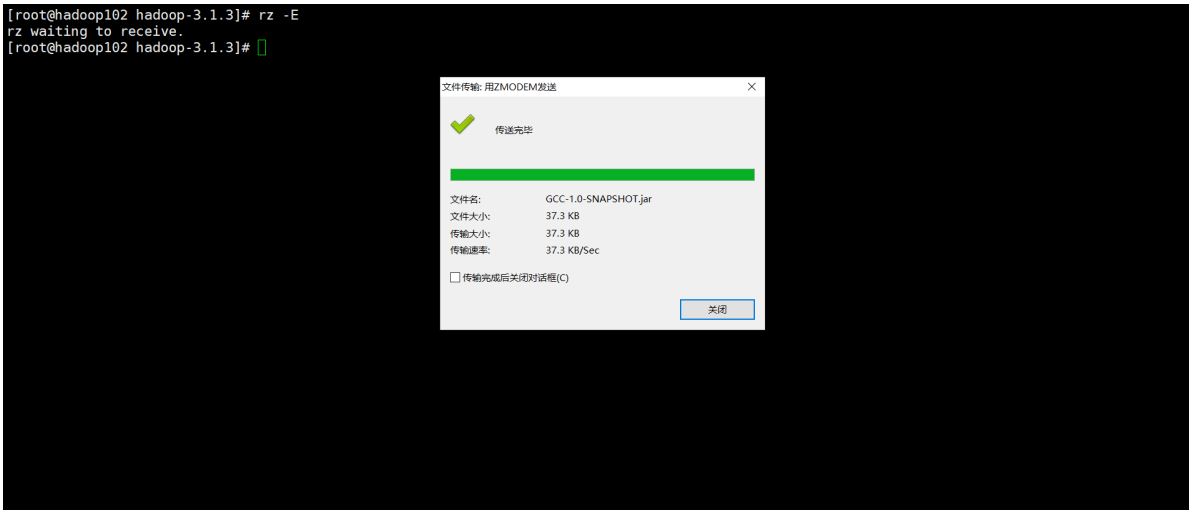
```
[root@hadoop102 hadoop-3.1.3]# hadoop jar MapReduceDemo-1.0-SNAPSHOT.jar example.org.mapreduce.lab.driver /input/PP-Decagon_ppi.csv /preprocess
2022-06-05 23:08:16,178 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 23:08:16,555 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and ex
ecute your application with ToolRunner to remedy this.
2022-06-05 23:08:16,571 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1654434
395401_0030
2022-06-05 23:08:16,647 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:08:16,747 INFO input.FileInputFormat: Total input files to process : 1
2022-06-05 23:08:16,775 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:08:16,800 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:08:16,815 INFO mapreduce.JobSubmitter: number of splits:1
2022-06-05 23:08:16,907 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:08:16,928 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654434395401_0030
2022-06-05 23:08:16,928 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 23:08:17,064 INFO conf.Configuration: resource-types.xml not found
2022-06-05 23:08:17,064 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2022-06-05 23:08:17,107 INFO impl.YarnClientImpl: Submitted application application_1654434395401_0030
2022-06-05 23:08:17,134 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654434395401_0030/
2022-06-05 23:08:17,135 INFO mapreduce.Job: Running job: job_1654434395401_0030
```



1. compile program to Jar file



2. load Jar file to VM



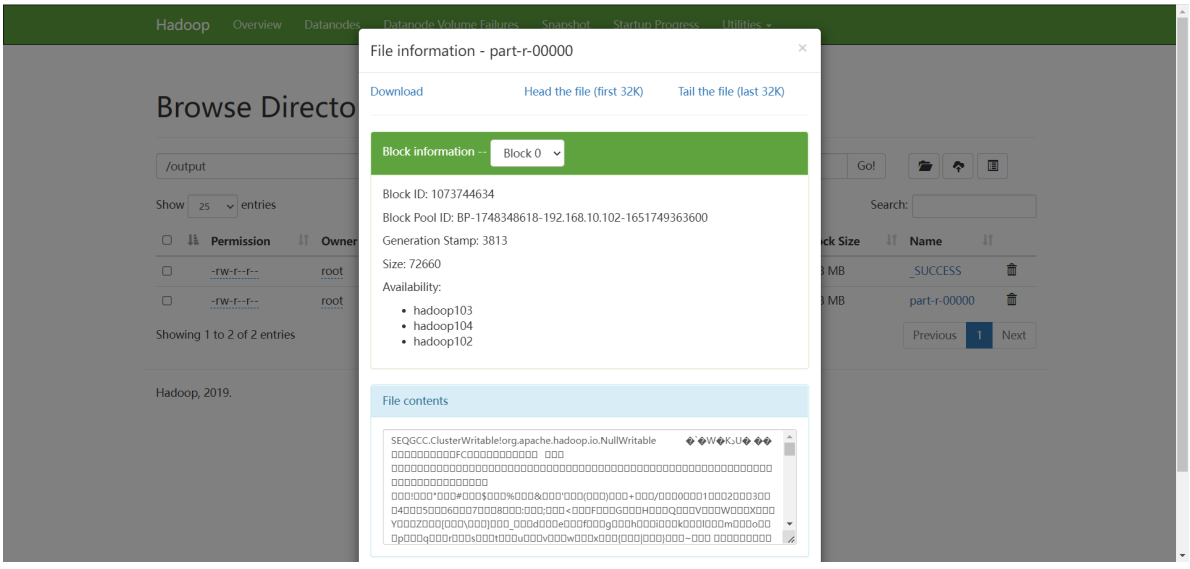
3. run program in VM

task1: ConnectedComponents.class

```
[root@hadoop102 hadoop-3.1.3]# rz -E
rz waiting to receive.
[root@hadoop102 hadoop-3.1.3]# hadoop jar GCC-1.0-SNAPSHOT.jar GCC.ConnectedComponents /preprocess/part-r-00000 /output
Start ConnectedComponents.
2022-06-05 23:39:41,246 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:39:41,359 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 23:39:41,690 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1654434
395401_0032
2022-06-05 23:39:41,740 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:39:41,808 INFO input.FileInputFormat: Total input files to process : 1
2022-06-05 23:39:41,830 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:39:41,857 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:39:41,872 INFO mapreduce.JobSubmitter: number of splits:1
2022-06-05 23:39:41,961 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:39:41,982 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654434395401_0032
2022-06-05 23:39:41,982 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 23:39:42,097 INFO conf.Configuration: resource-types.xml not found
2022-06-05 23:39:42,097 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2022-06-05 23:39:42,138 INFO impl.YarnClientImpl: Submitted application application_1654434395401_0032
2022-06-05 23:39:42,172 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654434395401_0032/
```

```
2022-06-05 23:42:00,341 INFO input.FileInputFormat: Total input files to process : 1
2022-06-05 23:42:00,349 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:42:00,370 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:42:00,387 INFO mapreduce.JobSubmitter: number of splits:1
2022-06-05 23:42:00,398 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:42:00,417 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654434395401_0041
2022-06-05 23:42:00,417 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 23:42:00,436 INFO impl.YarnClientImpl: Submitted application application_1654434395401_0041
2022-06-05 23:42:00,441 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654434395401_0041/
2022-06-05 23:42:15,555 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 23:42:15,572 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1654434
395401_0042
2022-06-05 23:42:15,582 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:42:15,606 INFO input.FileInputFormat: Total input files to process : 1
2022-06-05 23:42:15,620 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:42:15,641 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:42:15,658 INFO mapreduce.JobSubmitter: number of splits:1
2022-06-05 23:42:15,672 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:42:15,691 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654434395401_0042
2022-06-05 23:42:15,691 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 23:42:15,708 INFO impl.YarnClientImpl: Submitted application application_1654434395401_0042
2022-06-05 23:42:15,714 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654434395401_0042/
End ConnectedComponents.
Input file format: ADJACENCY_LIST.
Number of initial nodes: 17544.
Number of Cliques: 0.
Number of final nodes: 18055.
Number of Clusters: 30.
TestOK: true.
[root@hadoop102 hadoop-3.1.3]#
```

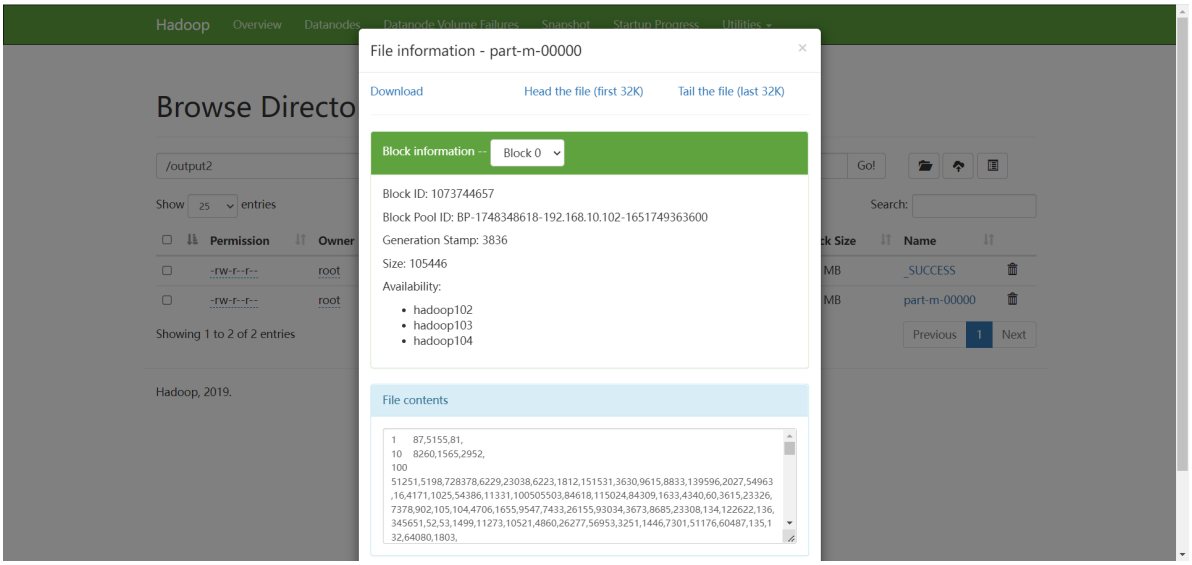
result:



this is hadoop sequence file, its binary so cannot be directly read

task2: Cluster2Text

```
[root@hadoop102 hadoop-3.1.3]# hadoop jar GCC-1.0-SNAPSHOT.jar GCC.TranslatorDriver Cluster2Text /output/part-r-00000 /output2
Start TranslatorDriver Cluster2Text.
2022-06-05 23:51:11,685 INFO client.RMProxy: Connecting to ResourceManager at hadoop103/192.168.10.103:8032
2022-06-05 23:51:12,047 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1654434
395401_0043
2022-06-05 23:51:12,112 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:51:12,201 INFO input.FileInputFormat: Total input files to process : 1
2022-06-05 23:51:12,218 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:51:12,242 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:51:12,256 INFO mapreduce.JobSubmitter: number of splits:1
2022-06-05 23:51:12,355 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2022-06-05 23:51:12,373 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1654434395401_0043
2022-06-05 23:51:12,373 INFO mapreduce.JobSubmitter: Executing with tokens: []
2022-06-05 23:51:12,501 INFO conf.Configuration: resource-types.xml not found
2022-06-05 23:51:12,501 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2022-06-05 23:51:12,540 INFO impl.YarnClientImpl: Submitted application application_1654434395401_0043
2022-06-05 23:51:12,565 INFO mapreduce.Job: The url to track the job: http://hadoop103:8088/proxy/application_1654434395401_0043/
2022-06-05 23:51:12,565 INFO mapreduce.Job: Running job: job_1654434395401_0043
```

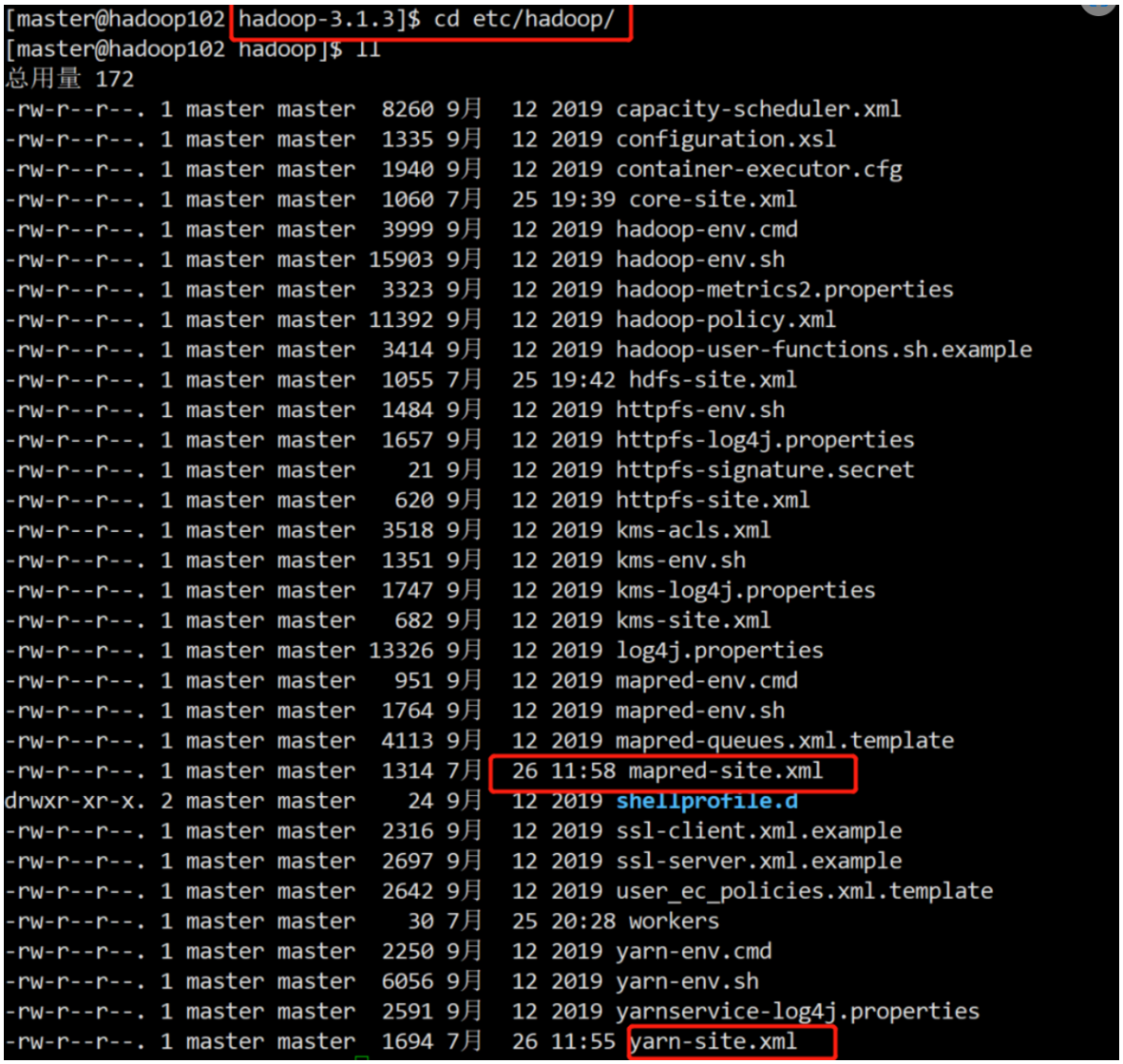


6. Difficulties and Solutions

6.1 Memory Allocation Problem

Background description: When running MapReduce: Container killed on request. Exit code is 143

Problem analysis: Insufficient memory allocation, the xml file parameters of MapReduce and yarn need to be adjusted



Change mapred-site.xml file.

```
vim mapred-site.xml
```

```
<property>
    <name>mapreduce.map.memory.mb</name>
    <value>1500</value>
    <description>The Physical Memory Upper Bound of Map Task
        </description>
</property>

<property>
    <name>mapreduce.reduce.memory.mb</name>
    <value>3000</value>
    <description>The Physical Memory Upper Bound of Reduce
        Task</description>
</property>

<property>
    <name>mapreduce.map.java.opts</name>
    <value>-Xmx1200m</value>
</property>

<property>
    <name>mapreduce.reduce.java.opts</name>
    <value>-Xmx2600m</value>
</property>
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

Change yarn-site.xml file.

```
vim yarn-site.xml
```

```
<property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>22528</value>
    <discription>The Available Memory for Each Node (MB)</
        discription>
</property>

<property>
    <name>yarn.scheduler.minimum-allocation-mb</name>
    <value>1500</value>
    <discription>The Smallest Memory for each Task: 1024MB</
        discription>
</property>

<property>
    <name>yarn.scheduler.maximum-allocation-mb</name>
    <value>16384</value>
```

```
    <discription>The Largest Memory for each Task</
        discription>
</property>
```

```
stop-all.sh
start-all.sh
```

6.2 Java Heap Space is not enough

This is a kind of heap error. It is caused by a too small heap size.

We solve this problem by change the hadoop configuration,

```
cd hadoop/etc/hadoop/
vim hadoop-env.sh
```

```
export HADOOP_HEAPSIZE=2000
```

```
cd hadoop/etc/hadoop/
vim mapred-site.xml
```

```
<property>
  <name>mapred.child.java.opts</name>
  <value>-Xmx2000m</value>
</property>
```

6.3 MapReduce Implementation in LargeStar Task

The specific implementation of mapreduce for large and small stars is difficult, and the specific input and output form is very vague. We solve this problem by redefining the pairnodewritable class and implementing the compareTo interface.

```
public int compareTo(NodesPairWritable other) {
    int result = this.NodeID - other.NodeID;
    if (result == 0)
        result = this.NeighbourID - other.NeighbourID;
    return result;
}
```

6.4 Shuffle Error of MapReduce

```
Error: org.apache.hadoop.mapreduce.task.reduce.
   Shuffle$ShuffleError: error in shuffle in fetcher#1
        at org.apache.hadoop.mapreduce.task.reduce.Shuffle.
           run(Shuffle.java:134)
```

```
        at org.apache.hadoop.mapred.ReduceTask.run(ReduceTask
            .java:376)
        at org.apache.hadoop.mapred.YarnChild$2.run(YarnChild
            .java:167)
        at java.security.AccessController.doPrivileged(Native
            Method)
        at javax.security.auth.Subject.doAs(Subject.java:396)
        at org.apache.hadoop.security.UserGroupInformation.
            doAs(UserGroupInformation.java:1556)
        at org.apache.hadoop.mapred.YarnChild.main(YarnChild.
            java:162)
Caused by: java.lang.OutOfMemoryError: Java heap space
        at org.apache.hadoop.io.BoundedByteArrayOutputStream
            .<init>(BoundedByteArrayOutputStream.java:56)
        at org.apache.hadoop.io.BoundedByteArrayOutputStream
            .<init>(BoundedByteArrayOutputStream.java:46)
        at org.apache.hadoop.mapreduce.task.reduce.
            InMemoryMapOutput.<init>(InMemoryMapOutput.java
            :63)
        at org.apache.hadoop.mapreduce.task.reduce.
            MergeManagerImpl.unconditionalReserve(
            MergeManagerImpl.java:297)
        at org.apache.hadoop.mapreduce.task.reduce.
            MergeManagerImpl.reserve(MergeManagerImpl.java
            :287)
        at org.apache.hadoop.mapreduce.task.reduce.Fetcher.
            copyMapOutput(Fetcher.java:411)
        at org.apache.hadoop.mapreduce.task.reduce.Fetcher.
            copyFromHost(Fetcher.java:341)
        at org.apache.hadoop.mapreduce.task.reduce.Fetcher.
            run(Fetcher.java:165)
```

According to "Hadoop: The Definitive Guide 4th Edition" (P203-219), a shuffle process is required between the map task and the reduce task, which copies the output of the map task as the input of the reduce task.

Specifically, the input of the shuffle process is: the output file of the map task, its output receiver is: the memory buffer on the machine running the reduce task, and the shuffle process runs in parallel.

The parameter mapreduce.reduce.shuffle.input.buffer.percent controls how much of the memory on the machine running the reduce task is used as the above buffer (the default value is 0.70), and the parameter mapreduce.reduce.shuffle.parallelcopies controls the parallelism of the shuffle process (the default value is 0.70). value is 5)

Then "mapreduce.reduce.shuffle.input.buffer.percent" * "mapreduce.reduce.shuffle.parallelcopies" must be less than or equal to 1, otherwise the above error will occur.

Therefore, I set mapreduce.reduce.shuffle.input.buffer.percent to a value of 0.1 and it works fine (set to 0.2, it still throws the same error)

Another solution is to solve this problem by modifying the mapred.xml file.

```
<property>
    <name>mapreduce.reduce.input.buffer.percent</name>
    <value>0.0</value>
</property>
```

In addition, it can be found that if the default values of the two parameters are used, the product of the two is 3.5, which is much larger than 1. Why are the above errors not often thrown?

1) First, set the default values to be relatively large, mainly based on performance considerations. Setting them to relatively large can greatly speed up the speed of copying data from the map

2) Secondly, to throw the above exception, another condition needs to be met, that is, the data of the map task is ready to be copied by shuffle. In this case, the "number of threads" and " "Memory buffer usage" is a full load value, which naturally causes an insufficient memory error; and if the data of the map task is completed intermittently, there is no "thread number" and "memory buffer usage" of the shuffle process at any time. It is a full load value, so naturally it will not throw the above error

In addition, if an error still occurs after setting the above parameters, it may be that the total memory of the process running the Reduce task is insufficient, which can be adjusted by the mapred.child.java.opts parameter, such as setting mapred.child.java.opts =-Xmx2024m.

7. Conclusions

In this research projecct, we propose a new distributed graph data processing system: DGPS. DGPS is based on the distributed computing framework Hadoop, which implements parallel computing of graph algorithms. Specifically, in practice, we split the graph into several smaller graphs and put them on different machines for computation. Based on DGPS, we have implemented three graph algorithms: Small Dense Subgraphs Finding, PageRank and Graph Connected Components. We choose a large scale graph for our experiments and obtained efficient running speed and good performance no less than the single machine algorithm. These results confirm the efficiency and effectiveness of our framework. It also reveals the potential of distributed graph processing systems.

We provide the complete framework design, algorithm implementation, and experimental procedure and results. We use text descriptions, frame diagrams, pseudo-code and screen-shots to show our results and workload in a variety of ways.