Assignment 4

Sahand Sabour - 山姆 - 2022380024

Task 1

Q1

A trivial approach to decrease the network traffic would be computing possible calculations within before sending the results from the Mapper to the Reducer. The paper introduces **Combiner** functions that can be written as an intermediate process between a map and a reduce task. For instance, for the WordCount example, an initial count of the words (i.e., sum up the number of occurrences for the same word) could be calculated before the keys and values are sent to the reduce function, which reduces the network traffic as less key-value sets would need to be sent to the Reducer.

Q2

The paper introduces the concept of **stragglers** to represent tasks that take an unusually long time to complete one of the last few steps of either mapping or reducing tasks. The master creates backups of the remaining unfinished tasks to alleviate this issue. Accordingly, the task can be completed by its primary execution or the created backup, whichever is faster. This process has shown to significantly reduce the execution time without significantly increasing the computational resources usage. This is referred to as creating **backup tasks**. Hence, a possible approach to address the issue of a mapper or reducer task being too slow would be for the master to create backups of such tasks that could run in parallel and the results from the execution that completes faster could be used.

Task 2

The process of this task is relatively similar to the Wordcount example, as we can think of each starting node as a word and count each edge as one of its occurrences. Therefore, we would read the input file and split it into different lines for the mapper. Accordingly, since the format of each line is ***a u v w*** and we only care about what the starting node u is, we split each line by the whitespace and extract ***u*** from the resulting array. Then, we record this as one occurrence of ***u*** in our data. In the reducer, for the same starting node u, we count how many times we have seen the pair ***(u, 1)*** in the mapper and consider the

sum of these values as the out-degree for ***u***. Since performance did not matter in this assignment, we did not implement a combiner function.

```
// Mapper
public static class OutDegreeMapper extends Mapper<Object, Text, Text,
IntWritable> {

  private final static IntWritable one = new IntWritable(1); //value
  private Text node = new Text(); //key

  public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString(), "\\n"); //split
by lines
    while (itr.hasMoreTokens()) {
      node.set(itr.nextToken().split(" ")[1]); // get u from (a, u, v, w)
      context.write(node, one); // write the (key, value) pair to context
    }
  }
}

// Reducer
public static class IntSumReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
  private IntWritable result = new IntWritable(); // out-degree

  public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;
    // count the number of occurrences
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum); // set sum as value for the node
    context.write(key, result);
  }
}
```

Task 3

Similar to the previous task, we first calculate the outdegree of each vertice (node). That is, the output file of Task 2 would be used as the starting point of this task. Accordingly, in the mapper function, we would replace keys and values in each pair (i.e., (key, value) to (value, key) as we would like to sort the pairs based on the number of occurrences.

```
// Mapper - sorter
public static class OutDegreeSortMapper extends Mapper<Object, Text,
IntWritable, Text> {

  private Text node = new Text(); // key
  private IntWritable outdegree = new IntWritable(); // value

  public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString(), "\\n"); //
split by lines
    while (itr.hasMoreTokens()) {
      String[] line = itr.nextToken().split("\\t");
      node.set(line[0]); // get u from (u, out_degree)
      outdegree.set(Integer.parseInt(line[1])); // get out_degree from (u,
out_degree)
      context.write(outdegree, node); // write the (value, key) to reverse the
pair
    }
  }
}
```

```
public static class OutDegreeSortComparator extends WritableComparator {
  public OutDegreeSortComparator() {
    super(IntWritable.class, true);
  }

  public int compare(WritableComparable wc1, WritableComparable wc2) {
    IntWritable key1 = (IntWritable) wc1;
    IntWritable key2 = (IntWritable) wc2;
    return -1 * key1.compareTo(key2);
  }
}
```

Then, as shown above, we leverage an intermediary function to sort the pairs based on

the keys representing the out-degrees of different nodes. Lastly, since we have a sorted list of (value, key) pairs, in the reduce function, we only need to reverse the pair once again and only save the top-n results, where n=2 for case1 and n=20 for case2.

```
// Reducer - sorter
public static class OutDegreeSortReducer
  extends Reducer<IntWritable, Text, Text, IntWritable> {
  int count = 0; // Number of nodes already saved

  public void reduce(IntWritable key, Iterable<Text> values,
                     Context context) throws IOException, InterruptedException
{
    // the number of top-n biggest nodes
    int print_len =
Integer.parseInt(context.getConfiguration().get("print_len"));
    for (Text val : values) {
      if (count >= print_len)
        break;
      context.write(val, key);
      count++;
    }
  }
}
```

Guide

First, direct to the project directory:

```
cd hw5_src
```

To run Task 2, as provided in the assignment guideline, simply run the following command while in the project directory:

```
./run_od.sh
```

Similarly, to run Task 3, simply run the following command while in the project directory:

```
./run_od_sorted.sh
```