Charles Zheng CME 323 HW 1

```
1.
   Checkpoint on slide 11:
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
   Checkpoint on slide 55:
(SparkCamp, 4)
(Spark, 3)
(spark,1)
(SparkSQL, 1)
(../spark/bin/spark-submit,1)
   Code for slide 60:
val rdd1 = sc.textFile("README.md").filter(_ contains "Spark")
val rdd2 = sc.textFile("spark/docs/contributing-to-spark.md").filter(_ contains "Spar
val wc1 = rdd1.flatMap(l => l.split(" ")).map(w => (w, 1)).reduceByKey(_ + _)
val wc2 = rdd2.flatMap(1 => 1.split(" ")).map(w => (w, 1)).reduceByKey(_+ _)
val joined = wc1.join(wc2)
   Checkpoint on slide 60:
(Spark, (3,2))
2.
   The mapper emits one key-value output pair for every input pair of
vertices: the output key is the sorted vertices and the output value indicates
the direction of the edge. The mapper takes directed edge \langle a, b \rangle: if a < b, it
emits \langle (a,b), 1 \rangle and if a > b, it emits \langle (b,a), 2 \rangle.
   The reducer sees all the values v_1, \ldots, v_n for a given key (c, d). If \{1, 2\} \subseteq
\{v_1,\ldots,v_n\} then it emits (c,d); otherwise it emits nothing.
   No combiner is used; combiners are not likely to help in this problem.
Algorithm 1 Map
  function Map(\langle a, b \rangle)
     if a < b then
         Emit \langle (a,b), 1 \rangle
```

else

end if end function

Emit $\langle (b,a), 2 \rangle$

Algorithm 2 Reduce

```
function Reduce(Key (c,d), Values \{v_1,\ldots,v_n\})

if n<2 then return

end if

for i=2,\ldots,n do

if v_i\neq v_{i-1} then

Emit \langle c,d\rangle

return

end if

end for

end function
```

3.

The combiner is the same as the reducer. Let N be the total number of words.

Algorithm 3 Map

```
function Map(String s)
for Word w in s do
Emit \langle w, 1 \rangle
end for
end function
```

Algorithm 4 Reduce/Combine

```
function Reduce(Key w, Values \{v_1, \ldots, v_n\})
s \leftarrow 0
for i = 1, \ldots, n do
s \leftarrow s + v_i
end for
Emit \langle w, s \rangle
end function
```

Without combiners– The shuffle size is N, and the reduce takes $N \pm O(B)$ operations.

With combiners—After the combine step, there are at most k key-value output pairs, since ther are at most k distinct keys. The shuffle size is kB, and the reduce takes $kB \pm O(B)$ operations.

4.

Let me briefly state the naive solution, which does not parallelize effectively. Run one map-reduce to count the number of elements N. Next, map each input pair $\langle i, a_i \rangle$ to N-i+1 output pairs $\langle i, a_i \rangle$, $\langle i+1, a_{i+1} \rangle, \ldots, \langle N, a_N \rangle$. Reduce by summing all values for a given key. With combiners, the shuffle size is N, and the number of reduce operations is NB where B is the number of mappers. The problem with this solution is that each mapper requires O(N) storage to hold the output keys—but this is on the same order as the size of the entire data.

A better idea is to use divide-and-conquer. The idea is to divide the keyset into B equally-sized partitions: $P_1 = \{1, \ldots, n_1\}, P_2 = \{n_1+1, \ldots, n_2\}, \ldots, P_B = \{n_{B-1}+1, \ldots N\}$. Accordingly define

$$\phi(i) = b$$
 such that $i \in P_b$.

Then define the partial sums p_1, \ldots, p_B by

$$p_b = \sum_{i \in P_b} a_i,$$

and define quantities

$$u_b = \sum_{c < b} p_b.$$

For i = 1, ..., n define the within-partition prefix sums as

$$t_i = \sum_{j \in P_{\phi(i)}: j \le i} a_j$$

Now observe that

$$s_i = t_i + u_{\phi(i)}$$

Therefore the procedure is as follows

- 1. (Map/Reduce 1) Count the number of keys N
- 2. (Map 2) Input: the original key-value pairs. Partition the keys sequentially into B workers
- 3. (Reduce 2) Each worker b = 1, ..., B computes p_b and sends it to the driver
- 4. The driver computes $u_b = \sum_{c < b} p_c$ and sends u_b to each worker b for $b = 1, \ldots, B$.

5. (Reduce 3) Input: the output of step 2. Each worker b = 1, ..., B computes $s_i = u_b + t_i$ and emits $\langle i, s_i \rangle$ for each $i \in P_b$.

To formalize this procedure in the map/reduce framework we have to designate the partition number b as a key throughout steps 2-5. In steps 2 and 5, we have (i, a_i) as values. Note that step 5 uses the same input as step 3.

Algorithm 5 Step 2: Map 2

```
Parameters n_1, \ldots, n_{B-1} determined in Step 1, and n_B = N.

function MAP(\langle i, a \rangle from original inputs)

for b \in 1, \ldots, B do

if n_b > i then

Emit \langle b, (i, a) \rangle

return

end if

end for

end function
```

Algorithm 6 Step 3: Reduce 2

```
function Reduce(Key b, values (i,a) from step 2)
p \leftarrow 0
for (i,a) in values do
p \leftarrow p + a
end for
Emit \langle b, p \rangle
end function
```

Algorithm 7 Step 5: Reduce 3

```
Parameters u_1, \dots u_B computed by driver in step 3.

function Reduce(Key b, values (i, a) from step 2)

Sort values (i, a) by i
s \leftarrow u_b

for Value (i, a) in sorted list do
s \leftarrow s + a
Emit \langle i, s \rangle
end for

end function
```

The cost of the computation is dominated by step 2, when the data is partitioned: this requires a shuffle size of N. This is followed by a reduce in step 3 requiring O(N/B) operations and O(1) space. The driver has to complete O(B) operations in step 4. Finally, each worker has to complete O(N/B) operations in step 5, requiring O(1) memory. The overall number of Map/Reduce iterations is 3, including the initial count.

5.

a. Let A be the adjacency matrix. Then the i, j entry of A^2 is nonzero if and only if there is a path of length 2 from vertex i to j. Therefore the procedure is to evaluate A^2 , which costs $O(n^{2.376})$ operations, and check the n(n-1)/2 off-diagonals, which costs $O(n^2)$ operations.

b. Let $b = \lfloor n/r \rfloor$, and write $A = [A_1, \ldots, A_b]^T$ where A_1, \ldots, A_{b-1} are $r \times r$ and A_b is $r \times (n - (b-1)r)$. Also write $B = [B_1, \ldots, B_b]$ where B_1, \ldots, B_{b-1} are $r \times r$ and $B_b is r \times (n - (b-1)r)$. Then C = AB is composed of blocks $C_{i,j} = A_i^T B_j$. The cost to compute $C_{i,j}$ is the cost to multiply A_i^T and B_j , which is bounded by $O(r^{2.376})$. Meanwhile, there are b^2 such blocks, where $b^2 = O((n/r)^2)$. Therefore the cost is $O((n/r)^2 r^{2.376})$.

Let $m = n^q$, where $q \ge 1$. Our goal is to provide an algorithm with runtime $O(m^{.55}n^{1.45}) = O(n^{1.45+0.55q})$.

In the proof we will make use of the following Lemma:

Lemma 1 Let $x = (x_1, ..., x_n)$, such that $x \ge 0$, $\sum_i x_i \le m$, and $\max_i x_i \le t$. Then

$$||x||^2 \le mt$$

Proof. Since x > 0, we have $\sum_i x_i = \sum_i |x_i| = ||x||_1$ and also $\max_i x_i = \max_i |x_i| = ||x||_{\infty}$. Now apply Holder's inequality:

$$||x||^2 \le ||x||_1 ||x||_\infty \le mt$$

Algorithm.

- 1. Compute the degree of each vertex, and compute q. Let p=1.45-0.45q.
- 2. Order the vertices in descending order by degree. Let $1, \ldots, n_h$ denote all the vertices with degree larger than n^p .
- 3. Let A be the adjacency matrix of the graph, and write $A = [A_1 A_2]$ where A_1 is $n \times n_h$ and A_2 is $n \times (n n_h)$.
- 4. Compute $A_1A_1^T$ using the method in part b.
- 5. Compute $A_2A_2^T$ as the sum

$$A_2 A_2^T = \sum_{v=n_h+1}^{n} A_v A_v^T$$

where A_v is the vth column of A. Let S denote a $n \times n$ matrix intialized to 0. For each for each $v = n_h + 1, \ldots, n$, $A_v A_v^T$ is a matrix of zeros and ones, with exactly d_v^2 ones. Therefore, compute $A_2 A_v^T$ by iterating over all $v = n_h + 1, \ldots, n$ and incrementing by 1 the d_v^2 entries of S which correspond to nonzero entries of $A_v A_v^T$.

6. Compute $A^2 = A_1 A_1^T + A_2 A_2^T$. Check the off-diagonals of A^2 : G is shallow if and only if all of the off-diagonals of A^2 are nonzero.

Proof. Steps 1-3 and step 6 have costs of at most $O(n^2)$ which is less than $O(n^{1.45+0.55q})$ for any $q \ge 1$. Hence it remains to show that both of steps 5 and 5 are $O(n^{1.45+0.55q})$.

In step 4, we multiply a $n \times n_h$ matrix by an $n_h \times n$ matrix. Since the number of edges is twice the sum of the degrees, we have $n_h < n^{q-p}$. Then the result in part b implies that the cost of step 4 is $O(n^{2+0.376q-0.376p})$.

In step 5, the cost of computing $A_2A_2^T$ is $\sum_{v=n_h+1}^n d_v^2$. By construction, $\max_{v>n_h} d_v \leq n^p$. Hence applying the lemma we have $\sum_{v=n_h+1}^n d_v^2 \leq 2mn^p$, so the cost of step 5 is $O(n^{p+q})$.

To complete the proof, see that the costs of steps 4 and 5 are both $O(n^{1.45+0.55q})$ once we susbstitute p=1.45-0.45q.