CS5300 Project 2: Fast Convergence PageRank in MapReduce

Giri Kuncoro (gk256), Yihui Fu (yf263), Shibo Zang(sz428)

1. Overall Structure

In this project, we implemented the following versions of computing PageRank values:

- Simple Computation of PageRank.
- Jacobi Blocked Computation of PageRank.
- **[Extra credit]** Gauss-Seidel Computation of PageRank
- **[Extra credit]** Random Block Partition

2. Preprocessing

2.1 Filtering Edges

The netID used to compute rejectMin and rejectLimit is gk256 with the reversed value being 0.652. The rejectMin is 0.5868 and the rejectLimit is 0.5968. In total, 7,524,304 out of 7,600,595 edges are selected in the graph, so we removed approximately 1% of the total edges. The pre-filtered edges file is stored in our S3 bucket: https://s3-us-west-2.amazonaws.com/edu-cornell-cs-cs5300s16-gk256/processed_edges.txt. To download the file, type `aws s3 cp s3://edu-cornell-cs-cs5300s16-gk256/processed_edges.txt <target_file_name>` in the terminal. It's the same format as the given edges.txt, i.e, `src-node-number dest-node-number floating number`.

The Python code used for filtering is in `preprocess/process_edges.py`. To run it, `cd` to the directory where process_edges.py is, put `edges.txt` in the same directory, then run `python process_edges.py <input-file> <output-file>`.

2.2 Converting Edges to Nodes

Additionally, we processed the edge file into a node-centric one so that it can be an appropriate input for mapper. It's also stored in our S3 bucket: https://s3-us-west-2.amazonaws.com/edu-cornell-cs-cs5300s16-gk256/nodepairs.txt. The format of a node entry is `nodeID-blockID pagerank-value neighborNodeID-blockID [...]`. To download the file, type `aws s3 cp s3://edu-cornell-cs-cs5300s16-gk256/nodepairs.txt <target_file_name>` in the terminal. Although some edges were rejected in the filtering, all the nodes in these edges were still retained.

The python code for this is in `preprocess/process_nodes.py`. To run it, `cd` to the directory where process_nodes.py is, put `blocks.txt` and the filtered edge file in the same directory, then run `python process_nodes.py <edge-file> <block-file> <output-file>`. You may find `blocks.txt` in `data/`.

2.3 Random Block Partition

To compute a bad partition for the graph, we assigned a node to a block based on the value of `nodeID % totalBlocks` where totalBlcoks is 68. This ensures that blocks are uniformly sized. The processed file is store in the S3 bucket as well:https://s3-us-west-2.amazonaws.com/edu-cornell-cs-cs5300s16-gk256/nodepairsRandom.txt. To download the file, type `aws s3 cp s3://edu-cornell-cs-cs5300s16-gk256/nodepairsRandom.txt <target_file_name>` in the terminal.

The python code for this is in `preprocess/process_nodes_random.py`. To run it, `cd` to the directory where process_nodes_random.py is, put the filtered edge file in the same directory, then run `python process_nodes_random.py <edge-file> <block-file> <output-file>`.

3. Other Important Information Regarding Compilation

3.1 Helper Classes

`src/proj2/main/node/Node.java` :

- Includes meta data about a node to make extracting information easier: nodeID-blockID, nodeID, blockID, currentPageRank, degree, emittedPageRank, and an array of destination nodes
- Please include this file when compling code Blocked or Gauss-Seidel Computation

`src/proj2/main/util/Constants.java` :

- Defines constant values used in this project, including damping factor, total node number, total block number, pass number for simple computation, convergence threhold, and enumerations for Hadoop counters
- Please include this file when compling code for every version of computation

3.2 External Jar Dependencies

Official jar files downloaded from Hadoop distribution

- hadoop/share/hadoop/common/hadoop-common-2.7.2.jar
- hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.7.2.jar

3.3 Project Jar File

In case the submitted source code cannot be compiled, we exported a jar file for the project and included it in the submission. To run a certain version, for example, type `hadoop jar <jar-file> proj2.main.simple.SimplePageRank <input-path> <output path>` in the terminal. You may add the number of MapReduce pass you want to run as the third argument after `<output path>` .

How to run:
Required files: `nodepairs.txt` (for METIS block partition) and `nodepairsRandom.txt` (for Random block partition) downloaded from our S3 bucket, see section 2 for instruction.

SimplePageRank
```
$ hadoop jar <jar-file> proj2.main.simple.SimplePageRank <input-path> <output-path> <number-of-passes (Optional)>
```

For example if I put the `nodepairs.txt` file on hdfs input, I can type:
`hadoop jar pagerank.jar proj2.main.simple.SimplePageRank input output 3` If the `<number-of-passes>` argument is not specified, 5 MapReduce passes will be run as default.

Jacobi BlockedPageRank
```
$ hadoop jar <jar-file> proj2.main.blocked.BlockedPageRank <input-path> <output-path> <number-of-passes (Optional)>
```

For example: `hadoop jar pagerank.jar proj2.main.blocked.BlockedPageRank input output 3` to run 3 MapReduce passes. If the `<number-of-passes>` argument is not specified, MapReduce will run until convergence. If user specifies more than required passes to convergence, it will stop when it converges.

Gauss-Seidel BlockedPageRank
```
$ hadoop jar <jar-file> proj2.main.gauss.GaussPageRank <input-path> <output-path> <number-of-passes (Optional)>
```

For example: `hadoop jar pagerank.jar proj2.main.gauss.GaussPageRank input output 3` to run 3 MapReduce passes. If the `<number-of-passes>` argument is not specified, MapReduce will run until convergence. If user specifies more than required passes to convergence, it will stop when it converges.

Random Block Partition

This can be run on both Jacobi and Gauss-Seidel Blocked PageRank, just simply change the input to `nodepairsRandom.txt` and run the same command for each algorithm you want to run.

4. Simple Computation of PageRank

`src/proj2/main/simple` package:

- SimplePageRank.java
  - Sets the input and output path accordingly
  - Runs 5 MapReduce passes as default
  - Initializes a MapReduce job for each MapReduce pass
  - Computes per iteration average residual using a Hadoop Counter

- SimpleMapper.java
  - Emits < srcNodeID-blockID, node entry > for computing residual and recovering node structure
  - Emits < destNodeID-blockID, outgoingPageRank> for every outgoing edge

- SimpleReducer.java
  - Updates the PageRank value for a node based on the PageRank values of its immediate neighbors, using the formula `(1-d)/N + d * sum (<PRt(u)/degree(u)>)` where d = 0.85 and N = 685230
  - Emits < nodeID-blockID, updated PageRank and reconstructed adjancency list >
  - Adds the residual for this node $|(PR^{t}(u) - PR^{t+1}(u))| / PR^{t+1}(u)$ to a residual counter

The average residual errors in each MapReduce pass are as the following table. After 5 MapReduce passes, it's still far from convergence.

Iteration	Average Error
0	2.338036138444408
1	0.32297669467638457
2	0.19207763229971833
3	0.09406491742335128
4	0.06281398244773434

5. Jacobi Blocked Computation of PageRank

`src/proj2/main/blocked` package:

- BlockedPageRank.java
  - Sets the input and output path accordingly
  - Runs MapReduce passes until reaching convergence where the average relative residual error is below 0.001
  - Computes average relative residual and average number of iterations per block for each reduce task by using two Hadoop Counters
  - Reports the PageRank value for two lowest-numbered notes in each block after the entire computation converged

- BlockedMapper.java
  - Emits < src-blockID, PR node entry > to compute residual in reducer and reconstruct node structure
  - Emits < dest-blockID, BE srcNodeID-blockID destNodeID-blockID > for every destionation node in the same block as the source node
  - Emits < dest-blockID, BC srcNodeID-blockID emittedPageRank destNodeID-blockID > for every destionation node not in the same block as the source node

- BlockedReducer.java
  - Parse the input values and store in different HashMaps accordingly

  - Performs in-block iterations until the in-block residual for the last iteration reaches convergence threshold 0.001
  - Emits < nodeID-blockID, updated PageRank value and reconstructed adjancency list >
  - Add the entire block reducer residual to a residual counter
  - The relative residual error reflects the difference of a node's PageRank value before and after internal block iterations: |(PR^{start}(v) - PR^{end}(v))| / PR^{end}(v)
  - Add the number of iterations in this block to an iteration counter

The average error and average in-block iteration number per block for each MapReduce pass of this version is as the following table. It shows that Jacobi Computation can reach convergence in 6 MapReduce passes while the Simple Computation would take longer.

Iteration	Average Error	Average Number of Iterations per Block
0	2.8105539255441174	17.529411764705884
1	0.037807487029411765	7.161764705882353
2	0.023995235897058823	5.882352941176471
3	0.009842906882352941	3.8970588235294117
4	0.003830367220588235	2.514705882352941
5	9.506281323529412E-4	1.338235294117647

The PageRank values for the two lowest-numbered nodes in each block are in `result/jacobi_pagerank_values.txt`

6. Gauss-Seidel Computation of PageRank

`src/proj2/main/gauss` package:

- GaussPageRank.java
  - This is the same as the BlockedPageRank.java
- GaussMapper.java
  - This is the same as the BlockedMapper.java
- GaussReducer.java
  - Overall functionality is similar to `BlockedMapper.java` (Jacobi version), including parse emitted output from Mapper, perform in-block iteration, and calculating in-block residual. However, in the `iterateBlockOnce` method, instead of updating newPageRank values in the end of every iteration, it updates values right after the newPageRank is computed. And when summing up the incoming edges, we get the pageRank values from the newest one, instead of value before the in-block iteration loop, like what Jacobi did. Thus, by this approach we would be able to compute using updated pageRank values whenever possible.
  - We attempted Topological sort in order to compute even faster using the maximum differences between inDegree and outDegree for each node to handle cycle and isolated nodes (typically naive topological sort can only compute DAG graph). However, the pageRank computation converges in 9 passes, which is not valid, thus we didn't include it in the deliverables.

The average error and average in-block iteration number per block for each MapReduce pass of this version is as the following table:

Iteration	Average Error	Average Number of Iterations per Block
0	2.8109963380735294	9.308823529411764
1	0.03863734327941176	5.220588235294118
2	0.02519361007352941	4.544117647058823
3	0.010960004794117647	3.2941176470588234
4	0.004877597617647059	2.3823529411764706
5	0.00179155175	1.6176470588235294
6	7.902894411764706E-4	1.3088235294117647

The PageRank values for the two lowest-numbered nodes in each block are in `result/gauss_pagerank_values.txt`.

Comparison Jacobi vs Gauss-Seidel

The results show that Gauss-Seidel Computation takes less number of in-block iteration to converge, particularly for the first three MapReduce passes, and need one pass more than Jacobi to converge. The first pass improves average number of iterations per block by 47%, and the second pass improves by 27%. This satisfies the claim that by using the newest pageRank values whenever possible, the in-block pageRank converges faster.

7. Random Block Partition

7.1 Convergence Performance for Jacobi Reducer using METIS partition and a random partion

Iteration	Content	Jacobi METIS	Jacobi Random
0	Average error	2.8105539255441174	2.3386830244264702
0	Average iteration/block	17.529411764705884	3.0
1	Average error	0.037807487029411765	0.32240249805882354
1	Average iteration/block	7.161764705882353	2.7205882352941178
2	Average error	0.023995235897058823	0.19121401836764707
2	Average iteration/block	5.882352941176471	2.0
3	Average error	0.009842906882352941	0.09338011023529412
3	Average iteration/block	3.8970588235294117	2.0
4	Average error	0.003830367220588235	0.0619256143382353
4	Average iteration/block	2.514705882352941	2.0
5	Average error	9.506281323529412E-4	0.03344842155882353
5	Average iteration/block	1.338235294117647	2.0

Compared to the result based on METIS, the Jacobi Computation based on a random partition converges much slower. Actually, it converges as slow as simple computation. This concludes that blockedPageRank only performs good in a good partition of block. It will have similar performance to SimplePageRank in a bad partitioned nodes, since the nodes will still be computed as node by node, in-block edges and boundary conditions are not reachable, the reducer will be getting a bunch of isolated in-blocks from the mapper.

7.2 Convergence Performance for Gauss-Seidel Reducer using METIS partition and a random partion

Iteration	Content	Gauss-Seidel METIS	Gauss-Seidel Random
0	Average error	2.8109963380735294	2.3386887671911762
0	Average iteration/block	9.308823529411764	2.514705882352941
1	Average error	0.03863734327941176	0.32242048685294117
1	Average iteration/block	5.220588235294118	2.0294117647058822
2	Average error	0.02519361007352941	0.19124016294117646
2	Average iteration/block	4.544117647058823	2.0
3	Average error	0.010960004794117647	0.09341979076470588
3	Average iteration/block	3.2941176470588234	2.0
4	Average error	0.004877597617647059	0.062080906352941174
4	Average iteration/block	2.3823529411764706	2.0
5	Average error	0.00179155175	0.0335133024117647
5	Average iteration/block	1.6176470588235294	2.0
6	Average error	7.902894411764706E-4	0.026807231970588236
6	Average iteration/block	1.3088235294117647	2.0

The random partition performance for Gauss-Seidel is the same as random performance in Jacobi, that is, it has slow performance similar to SimplePageRank. This due to similar reason: in bad partitioned block, reducer will be getting a bunch of isolated in-blocks nodes without reachable in-block edges and boundary values. Thus, no matter what BlockPageRank algorithm we are using, for either Jacobi or Gauss-Seidel, blockPageRank only performs better in a good partitioned nodes.