CSCI4180: Tutorial 6

Assignment 2 Review (Part 1)

Huancheng Puyang

Department of Computer Science & Engineering
The Chinese University of Hong Kong

2022.10.26

Outline

- Problem
  - Single-source shortest path length by parallel Dijkstra’s algorithm
- Three main modules
  - PDPreProcess.java
  - PDNodeWritable.java
  - ParallelDijkstra.java
- Implementation hints
  - Emit a node
  - Stop condition
  - Chain MapReduce jobs

Outline

- Problem
  - Single-source shortest path length by parallel Dijkstra's algorithm
- Three main modules
  - PDPreProcess.java
  - PDNodeWritable.java
  - ParallelDijkstra.java
- Implementation hints
  - Emit a node
  - Stop condition
  - Chain MapReduce jobs

Problem

- Single-source shortest path
  - Find shortest paths from a source node to all other nodes in the graph
  - Formalization
    - Given a graph $G = (V, E)$ and a source node $v_s \in V$
    - Find the shortest path length (distance) from v_s to every other reachable node in V
  - Breadth-first-search (BFS): smallest hop counts (unweighted)
  - Dijkstra’s algorithm: lowest weights (weighted)

Problem (Cont.)

- Parallel Dijkstra's algorithm
  - Refer to Slides 35-49 of Lecture 5 for details
  - Similar workflow as parallel BFS, yet with some differences
    - Consider general weights (weighted) instead of unweighted.
    - Consider stop iteration: how to know that all distances have been found?
      - Maintain a counter inside MapReduce program for some statistic, e.g., the number of nodes to be processed after each iteration (detailed later)

Problem (Cont.)

- Dataset: Twitter social graph
  - URL: https://anlab-kaist.github.io/traces/WWW2010
  - Original per-line structure, e.g., user1 user2
    - It means the relationship that user1 follows user2
  - Modification for part 2 of assignment 2
    - User ➔ node; relationship ➔ edge
      - Each user is represented as a node
      - Each node takes a unique positive integer as the **node ID**
      - E.g., for user1 follows user2, an edge is created from node1 to node2 in the graph
    - Original dataset is unweighted, we randomly assign a **positive integer** between 1 and 50 inclusively as the weight for each edge, e.g., user1 user2 ➔ node1 node2 30
    - Original dataset is tens of GB in total, we build a smaller one by sampling

Problem (Cont.)

- Dataset: Twitter social graph (cont.)
  - We have uploaded the dataset to Blackboard
    - Each part has a small case and a large case
      - Small case: example given by specification (tens of bytes)
      - Large case: sampled from Twitter dataset (tens of MB)
  - You can use the dataset for debug and test

Outline

- Problem
  - Single-source shortest path length by parallel Dijkstra’s algorithm
- Three main modules
  - PDPreProcess.java
  - PDNodeWritable.java
  - ParallelDijkstra.java
- Implementation hints
  - Emit a node
  - Stop condition
  - Chain MapReduce jobs

Three Main Modules

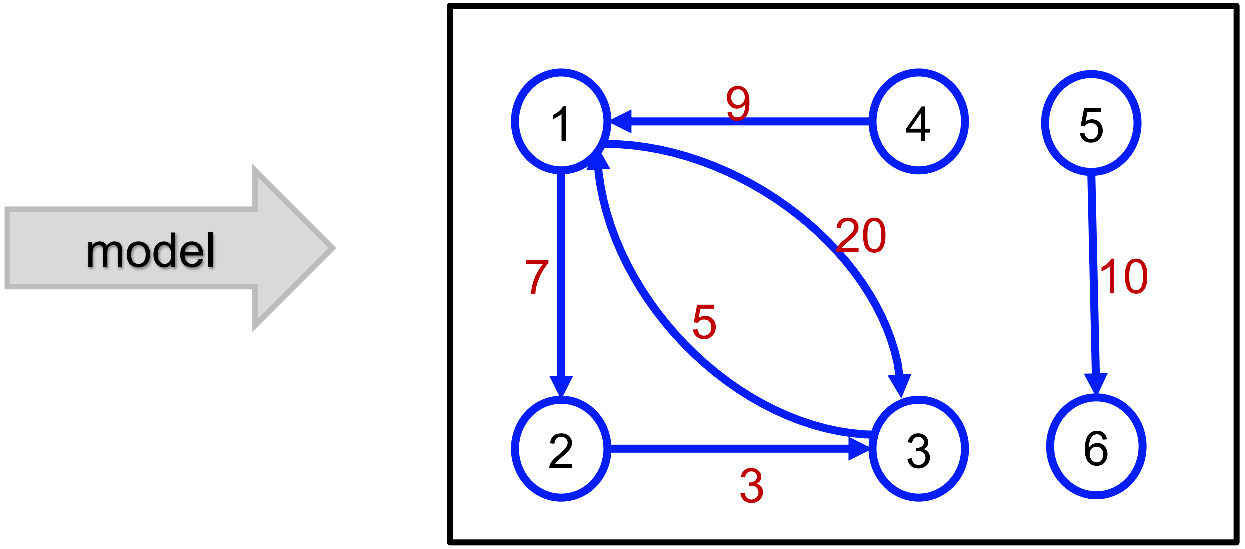
- PDPreProcess.java for parsing input
  - Example for parsing input
    - Model following lines to ***G = (V, E)***
    - 1173 1173 10
      - Add node 1173 to *V*
      - No edge is added to *E*
    - 1173 6267522 14
      - Add node 1173 to *V*
      - Add node 6267522 to *V*
      - Add edge from 1173 to 6267522 to *E*, whose weight is 14

Three Main Modules (Cont.)

- PDPreProcess.java for parsing input
  - Example for parsing input
    - Input example from specification:

```
1 2 7
1 3 20
2 3 3
3 1 5
4 1 9
5 6 10
```

Input



Three Main Modules (Cont.)

- PDPreProcess.java for parsing input
  - **Input** format → **adjacency list** format
  - A separate Mapper/Reducer to do the transformation
  - For the input example from specification:

```
1  2  7
1  3  20
2  3  3
3  1  5
4  1  9
5  6  10
```

Input

parse

```
1  (2  7), (3  20)
2  (3  3)
3  (1  5)
4  (1  9)
5  (6  10)
```

Adjacency List

Three Main Modules (Cont.)

- PDNodeWritable.java for node structure
  - Take a node v_k as an example, where v_s is the source node
    - Stores the node ID v_k to identify which node the structure belongs to
    - Stores the distance (shortest path length) from v_s to v_k
    - Keeps track of the neighbors of v_k by adjacency list
    - Keeps track of the ID of previous node leading to the minimum distance from v_s to v_k
  - Hints
    - Carefully design your structure
      - Proper constructor
      - Proper type for node ID, distance and adjacency list
    - Feel free to design your own format for right functionality

Three Main Modules (Cont.)

- ParallelDijkstra.java for main module
  - Mapper
    - For each node v_k
      - Receive node structure
        - Distance from v_s to v_k and adjacency list of v_k including distances from v_k to its neighbors
      - Calculate the distances from v_s to v_k's neighbors through v_k
      - Emit the node structure of v_k itself
      - Emit the distances from v_s to v_k's neighbors through v_k
      - Consider how to design your implementation

Three Main Modules (Cont.)

- ParallelDijkstra.java for main module
  - Reducer
    - For each node v_k
      - Receive the node structure of v_k and the distances from v_s to v_k through the predecessors
    - Find the smallest value and the corresponding previous node
      - Update the distance of v_k in node structure using the smallest value
      - Record the previous node in node structure
      - Think about how to fix the conflicting information
    - **NOTE**: Emitted node can be used by the Mappers in the next iteration

Outline

- Problem
  - Single-source shortest path length by parallel Dijkstra’s algorithm
- Three main modules
  - PDPreProcess.java
  - PDNodeWritable.java
  - ParallelDijkstra.java
- Implementation hints
  - Emit a node
  - Stop condition
  - Chain MapReduce jobs

Implementation Hints

- Emit a node
  - The Java Way
    - Implements a toString() method and a fromString() method in PDNodeWritable class
      - You can convert a Node to String, and parse a Node from String
    - Emit node.toString() simply by a Text and parse a Node by fromString() from a Text
      - See the hyperlink for how to convert between Text and String
    - NOTE: functions for parsing String like String.split() could be slow
  - The Hadoop Way (Recommended)
    - Implement Writable in PDNodeWritable class
      - You need to provide implementation for methods, e.g., readFields, write, and read
    - Emit a Node as the value directly
    - Could be faster than parsing String in the Java way, worth trying if you want to go for Bonus!

Implementation Hints (Cont.)

- Stop condition
  - Command line argument: iterations
    - Indicate the number of MapReduce iterations
    - Positive integer
  - Stop condition:
    - Iterations > 0 ➔ Stop after *iterations* runs (maximum number) of MapReduce (maybe earlier if all nodes are processed)
    - You can use Counters to decide if all reachable nodes have been processed (Slide 4)
  - Count the number of iterations
    - You can use Counters in Hadoop

Implementation Hints (Cont.)

- Stop condition (cont.)
  - Basic operations about Hadoop Counter
    - Declare Counter
      - public static enum ReachCounter { COUNT };
    - Increment Counter
      - context.getCounter(ReachCounter.COUNT).increment(1);
    - Retrieve Counter Value
      - long reachCount = job.getCounters().findCounter(ParallelDijkstra.ReachCounter.COUNT).getValue();

Implementation Hints (Cont.)

- Chain MapReduce jobs
  - Each "Map + Reduce" only explores one step further from v_s, we need to repeat this process
    - Map1, Reduce1, Map2, Reduce2, Map3...
  - How to chain MapReduce jobs?

Implementation Hints (Cont.)

- Chain MapReduce jobs (cont.)
  - The Java Way
    - Drive jobs in the main() function using loops, counter and conditions
    - Different **Configuration** and **Job Object** in each iteration
    - Set job.waitForCompletion(true);
    - Jobs communicate by writing and reading intermediate files on HDFS
    - For Job_i :
      - FileInputFormat.addInputPath(***OutputPath of*** $\boldsymbol{Job_{i-1}}$)

Implementation Hints (Cont.)

- Chain MapReduce jobs (cont.)
  - The Hadoop Way
    - Use JobControl
    - Add jobs to JobControl
      - JobControl jc = new JobControl();
      - jc.addJob(job1);
      - jc.addJob(job2);
    - Add dependency (e.g. job2 depends on job1)
      - job2.addDependingJob(job1)
    - Run JobControl
      - jc.run()
    - NOTE: YARN will schedule those jobs for you wisely

Implementation Hints (Cont.)

- Remarks
  - Cleanup
    - Transform the output of the last iterator to the required format
    - Only outputs the tuple for the nodes which are reachable from the source node
    - Feel free to use another set of Mapper/[Reducer] to do this step
  - Tips
    - Test your program correctness with hand-craft test cases (loops, directed edge, etc.)
    - **NOTE**: Please start early!

Thank You
Q&A