UNIVERSITAT POLITÈCNICA DE CATALUNYA

FACULTAT D'INFORMÀTICA DE BARCELONA

Distributed Graph Processing

Semantic Data Management

Spring 2023

Authors:
Tianheng, Zhou, *email*: tianheng.zhou@estudiantat.upc.edu
Lorencio Abril, Jose Antonio, *email*: jose.antonio.lorencio@estudiantat.upc.edu
Hoschek, Maren, *email*: maren.hoschek@estudiantat.upc.edu

Professor: **Flores, Javier**

Contents

1 Solutions **1**

1.1 Getting familiar with GraphX's Pregel API 1

1.2 Computing shortest path using Pregel 2

1.3 Extending shortest path computation 2

1.4 Spark Graph Frames 3

List of Figures

1.1 Output extended shortest path algorithm 3
1.2 Output of exercise 4. 5

Listings

1.1 Initialisation of the hash map 2
1.2 recursive computation of the shortest path 3
1.3 Optimizing maxIter. 4
1.4 Function getTop10. 4

Chapter 1

Solutions

All the code of the project can be accessed in its Github repository.

1.1 Getting familiar with GraphX's Pregel API

We assume the pair (x, y, z) represents Vertex x, the current state of it is y, and its received message is z.

1. First Superstep

   - **Gather:** The initial state is (1, 9, null), (2, 1, null), (3, 6, null), (4, 8, null). Since it is the first superstep, no messages are received for all of them (message equals Integer.MAX_VALUE in the program). So no Gather process here.
   - **Apply:** The user-defined function here is max(state, message), but the function won't be applied in this Apply step as there are no messages for each node. So, the resulting state is still (1, 9, null), (2, 1, null), (3, 6, null), (4, 8, null).
   - **Scatter:** According to directed edges between vertices, and the rule of sending a message, only Vertex 1 will send message 9 to Vertex 2 because the state value of Vertex 2 is 1, which is smaller than the state value 9 of Vertex 1.

2. Second Superstep

   - **Gather:** The initial state is (1, 9, null), (2, 1, 9), (3, 6, null), (4, 8, null). Vertex 2 receives and read messages.
   - **Apply:** The function max(state, message) will be applied to Vertex 2. The resulting state is (1, 9, null), (2, 9, null), (3, 6, null), (4, 8, null).
   - **Scatter:** According to directed edges between vertices, and the rule of sending a message, Vertex 2 will send message 9 to Vertex 3 and Vertex 4 because their state value 6 and 8 are both smaller than the state value 9 of Vertex 2.

3. Third Superstep

   - **Gather:** The initial state is (1, 9, null), (2, 9, null), (3, 6, 9), (4, 8, 9). Vertex 3 and 4

receive and read message.

- **Apply:** The function max(state, message) will be applied to Vertex 3 and 4. The resulting state is (1, 9, null), (2, 9, null), (3, 9, null), (4, 9, null).
- **Scatter:** According to directed edges between vertices, and the rule of sending a message, there won't be any messages sent because the state value of all vertex are equal.

So finally, 9 is the max value in the given graph, and the program finishes.

1.2 Computing shortest path using Pregel

All nodes are initialised with two properties: the node object itself and an integer attribute which we use to store the costs of the shortest path. The value for the costs is initialised with the highest possible integer value in java, with the exception of the source node, which is initialized as zero since the cost of the shortest path for this node is always zero. Every edge of the graph has three properties, the source node, the destination node and the cost it takes to get from the source to the destination node.

To calculate the shortest path, we have implemented the methods *VProg* and *sendMsg*. The method *VProg* checks whether the vertex value has to be updated by receiving a message. If the value of the received message is less than the current vertex value, the value is updated. If the value of the received message is greater or equal, the vertex value remains the same, since the shortest path has not changed. In addition, the special case that the vertex is the source vertex is handled, since in this case the value remains always set to zero.

The *sendMsg* method determines the costs for the path that are generated by sending a message through a given edge. The costs are derived from the value of the edge and the previous costs stored in the source node. If the sum of the two values is smaller than the costs stored in the destination node, the value is propagated to this node. If this is not the case, all values remain unchanged. The detailed implementation of both methods can be viewed in our source code. When running our code for exercise 2 we obtain the output given in the lab document.

1.3 Extending shortest path computation

In order to be able to return the nodes of the shortest path, we have extended the class from exercise 3 with a hash map. In the hash map object, the predecessor node on the shortest path is stored for each node. Initially, all predecessor nodes are set to minus one, since we do not yet know which node is the predecessor node on the shortest path. The code for creating and initialising the hash map is shown in Listing 1.1.

```
static Map<Object, Object> predecessors = new HashMap<Object, Object>();
for (Tuple2<Object,Integer> vertex_var :vertices){
    Object key = vertex_var._1();
    predecessors.put(key, (Object)(-11));
}
```

Listing 1.1: Initialisation of the hash map

In the *sendMsg* function, when a shorter path is found, not only the cost is propagated but also the predecessor node is set. In order to get the entire shortest path for each node at the end, the predecessor of each node is determined recursively using the hash map. The recursion is

finished when the hash map predecessor is minus one, since the start node has no predecessor and the predecessor therefore still has the initialised value of minus one. Listing 1.2 shows the code of the *get_recursive_path* method.

```
1    public static void get_recursive_path(Object vert, Map<Object, Object> p_map,
     List<String> path, Map<Long, String> labels){
2        if (p_map.get(vert)==(Object)(-11)) {
3            return;
4        } else {
5            Object parent_vertex = p_map.get(vert);
6            path.add(labels.get(parent_vertex));
7            get_recursive_path(parent_vertex, p_map, path, labels);
8        }}
```

Listing 1.2: recursive computation of the shortest path

In addition, we have implemented the method *get_path*, which returns the nodes of the shortest path as a list for outputting them as a string. The rest of the methods and program logic remains the same a in task 2. When we run our code we get the output in Figure 1.1.

```
Minimum path to get from A to A is [A] with cost 0
Minimum path to get from A to B is [A, B] with cost 4
Minimum path to get from A to C is [A, C] with cost 2
Minimum path to get from A to D is [A, C, E, D] with cost 9
Minimum path to get from A to E is [A, C, E] with cost 5
Minimum path to get from A to F is [A, C, E, D, F] with cost 20

Process finished with exit code 0
```

Figure 1.1: Output extended shortest path algorithm

1.4 Spark Graph Frames

For this last exercise, we first need to load the graph. To do this, we read the given data file containing the vertices and split each line with `'\t'` as delimiter, because the format of this files is `{id}'\t'{title}`. Then, we define the vertices_schema accordingly and close the file. The same approach is done with the edges, with the fields `src, dst,` and `relationship`. At this point, we can create the graphframe and utilize the pagerank algorithm on it.

To accomplish this final part of the exercise, we were asked to optimize the parameters of the algorithm. Let's briefly review what each parameter is, and how one can optimize them:

- *Damping factor*: the damping factor is usually interpreted as the probability that an user on a page clicks on a link and navigates to another page. Therefore, this is a semantic parameter, that cannot really be optimized, but it has to be obtained from data or assumed as a plausible value. It is usual to set it to 0.85. Notice that spark uses the $reset\_probability = 1 - damping\_factor$, so we set it to 0.15.

- *Maximum number of iterations*: this is the amount of iterations that the algorithm performs. To optimize it, we have run the algorithm increasing this parameter, until the top10 articles remains still from one execution to the following. We thought also in the possibility of increasing by bigger amounts the number of iterations, and performing later a binary search to make the search faster, but in the end this is quite an overkill to the

problem, because the graph is not very big and the convergence of this algorithm is quite fast. Thus, in the end we just kept the easy iterative approach. The code can be read in Listing 1.3.

```
public static List<Row> getTop10(GraphFrame gf, int nIter){
Boolean converged = false;
int iter = 1, jump = 10;
List<Row> left10 = null, right10 = null;
while(!converged){
    right10 = getTop10(gf, iter);
    if (left10 != null && left10.equals(right10)) {
        converged = true;
    } else {
        left10 = right10;
    }
    iter+=jump;
}
```

Listing 1.3: Optimizing maxIter.

To make the code more readable, we developed the function `getTop10`, which implements all the steps required to run the algorithm. It can be consulted in Listing 1.4. The logic is pretty simple: first, we initialize the algorithm and we run it. Then, we order the graph by the obtained pagerank values and finally we get the top 10 articles. The result is shown in 1.2, where we can see that the optimum number of iterations is 12 and the top 10 Wikipedia articles in our dataset (in which Berkeley seems to be really interesting!).

```
public static List<Row> getTop10(GraphFrame gf, int nIter){
    PageRank pagerank =  gf.pageRank().resetProbability(0.15).maxIter(nIter);
    GraphFrame rankedGraph = pagerank.run();
    Dataset<Row> ranks = rankedGraph.vertices().orderBy(functions.col("pagerank").
    desc());
    return ranks.select("id").limit(10).collectAsList();
}
```

Listing 1.4: Function getTop10.

```
The ideal number of iterations is 12 and the obtained ranking is:
+------------------+------------------+-----------------+
|                id|             title|         pagerank|
+------------------+------------------+-----------------+
|883029930693791843|University of Cal...|  3122.67599226804|
|1746517089350976281|Berkeley, California|1574.0836462829777|
|8262690695090170653|         Uc berkeley|384.43716486934414|
|7097126743572404313|Berkeley Software...| 214.3209089361296|
|8494280508059481751|Lawrence Berkeley...| 194.2048634360475|
|1735121673437871410|     George Berkeley| 193.5100549227113|
|6990487747244935452|      Busby Berkeley|112.36717757054133|
|1164897641584173425|      Berkeley Hills|106.83754165843061|
|5820259228361337957|     Xander Berkeley| 71.49157670569514|
|6033170360494767837|Berkeley County, ...| 68.28327472052656|
+------------------+------------------+-----------------+
```

Figure 1.2: Output of exercise 4.