Ander Barrio and Odysseas Kyparissis

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Semantic Data Management
Distributed Graph Processing Lab

Exercise 1: Getting familiar with GraphX's Pregel API

Firstly, the initialization of the vertices and edges of the graph takes place. Next, the configuration of Spark for parallel graph processing, and distribution of data across the cluster is enabled. A Graph object is then created by inserting vertex and edge data, as well as a GraphOps object for enabling graph-specific operations. Finally, the identification of the vertex with the maximum value is completed. Supersteps for finding the maximum value of all vertices are explained below:

Superstep 0: On the first iteration, the vertex program is invoked on all vertices and the pre-defined message is passed, which is 2147483647. **VProg** includes a statement for the initial superstep, that if the message is equal to 2147483647 then the vertex's value is returned. So, each vertex will have its own initial value and be in active state (Node 1: **9**, Node 2: **1**, Node 3: **6**, Node 4: **8**). Once **Vprog** is completed, **sendMsg** is called. Then adjacent node values are compared and if the value of the destination vertex is smaller that the value of the source vertex, the source vertex value is propagated to the destination vertex, by sending the appropriate message. This is enabled because of the triplet views available in the Pregel framework. Thus, the following vertices are compared (following *source_vertex_id* : *destination_vertex_id* format) **1:2, 2:3, 2:4, 3:1, 3:4**. For comparison **1:2**, **sendMsg** propagates the source vertex value to the destination, since 9 is bigger than 1. For comparisons **2:3, 2:4, 3:1** and **3:4 sendMsg** does not send anything to the destination vertices. Then the execution of superstep 1 starts.

Superstep 1: Here, Node 2 receives the message of superstep 0, which contains the value 9 from Node 1 and it remains active while its value is changed from 1 to 9. Once no other message was sent from superstep 0, sendMsg is invoked. The values of the vertices is the following: (Node 1: **9**, Node 2: **9**, Node 3: **6**, Node 4: **8**). In this case, the only comparisons which are being completed are **2:3** and **2:4** once all the other nodes are inactive because they did not receive any message from superstep 0. For both comparisons **2:3** and **2:4**, **sendMsg** propagates the source vertex value to the destination ones, since 9 is bigger than 6 and 8 respectively. Lastly superstep 2 is executed following the same logic.

Superstep 2: In this step, Node 3 and Node 4 receive the messages of superstep 1, containing the value 9 from Node 2. For that reason, their values are changed to 9 and they become active. To continue with, **sendMsg** is called again to complete the following comparisons: **3:1** and **3:4** (due to the fact that only Node 3 and Node 4 are active, but Node 4 do not have any outgoing Nodes connected). For both comparisons, source and destination nodes have the same value (9), and consequently **sendMsg** function does not send anything to any vertex. The computation of the max value of the graph terminates here.

Exercise 2: Computing shortest paths using Pregel

The task at hand is to program the calculation of the shortest routes from a starting point in the given graph. The logic of the implemented functions is the following one:

- **VProg**: The function takes three arguments: vertexID, vertexValue, and message, and returns an integer. During the superstep 0, the vertex program is invoked on all vertices and the pre-defined message is passed, which is 2147483647. If the message is equal to the pre-defined message, the function returns the current vertexValue. Otherwise, it returns the minimum value between the current vertexValue and the message. In other words, the function updates the state of each active vertex in the graph based on the minimum of its current state and the received message.

- **sendMsg**: This code implements a message-passing algorithm for graph processing, where each vertex can send messages to its neighbors in each iteration. The apply method of the sendMsg class takes an EdgeTriplet object as an argument, extracts the source vertex attribute and edge attribute from it, and determines whether the source vertex has still the pre-defined message (2147483647) or it has received useful information from its neighbors in previous supersteps. If so, the function creates a Tuple2 object with the destination vertex ID and the sum of the source vertex attribute and edge attribute, and returns it as a singleton iterator. Otherwise, the function returns an empty iterator. The code also includes a conversion from a Java iterator to a Scala iterator. To sum up, the sendMsg class defines the behavior of each vertex during the message-passing algorithm, which is to send a message to its neighbors containing the sum of the source vertex attribute and edge attribute if useful information has been received.

- **merge**: This merge function is used to combine the messages received by a vertex in each superstep of the algorithm. In each superstep, a vertex can receive multiple messages coming from its neighboring vertices, and these messages need to be combined in a meaningful way to update the vertex's state or compute new values. In this case, the merge function specifies that the value of a vertex should be updated with the minimum value of the set of values received from its neighbors. This is a common operation in graph algorithms and is used, specifically in finding the shortest path between two vertices in a graph. Overall, the merge function defines how messages should be combined during the message-passing algorithm to update the state of each vertex in the graph.

Excercise 3: Extending shortest path's computation

The previous method for calculating the shortest paths had a limitation. While we could determine the minimum cost to reach any other node from node A, we could not determine the actual path. Therefore, in this exercise, we are required to enhance the previous implementation to include the shortest path. Therefore, the new logic of the implemented functions is the following one:

- **MyVertexClass**: This is a Java class called MyVertexClass which extends the Tuple2 class. The Tuple2 class is a built-in class in Java which stores a pair of values. MyVertexClass has three constructors, one with no arguments which sets the first value to 0 and the second value to a new empty ArrayList of Strings, another with one argument (an integer) which sets the first value to the specified integer and the second value to an empty ArrayList of Strings, and a third with two arguments which sets the first value to the specified integer and the second

value to the specified List of Strings. The class has several methods, including getters for the two values, methods to create new instances of the class with updated values, a toString method which returns a String representation of the class, an equals method to check if two instances of the class are equal, and a hashCode method which returns a hash code for the class. Overall, this class is used to store pairs of integer and list of string values, and provides methods to create new instances with updated values and to check if two instances are equal. Finally, by using the functions of this class appropriately, the inclusion of the minimum path from a source to a destination vertex can be easily added to the solution of the previous exercise.

- **VProg**: The class has an apply method which takes three arguments: a Long representing the vertex ID, and two MyVertexClass objects representing the current vertex value and the received message. The apply method first checks if the received message is equal to the INITIAL_VALUE constant. If so, it returns the current vertex value. Otherwise, it compares the integer value stored in the current vertex value with the integer value stored in the received message. If the current vertex value has a greater or equal value, it returns the received message. Otherwise, it returns the current vertex value. Overall, this class defines the behavior of the vertices in a graph during a message-passing algorithm. It updates the vertex value based on the values received from neighboring vertices and returns the updated value.

- **sendMsg**: The sendMsg class takes an EdgeTriplet object as input and returns an Iterator of Tuple2 objects containing an ID and a MyVertexClass object. The function checks if the source vertex is equal to the INITIAL_VALUE, and if so, it returns an empty iterator. Otherwise, it creates a new MyVertexClass object by adding the edge weight to the source vertex's number and appending the edge label to its list of labels, and returns an iterator containing a Tuple2 object with the destination vertex ID and the newly created MyVertexClass object.

- **merge**: This is a static class that extends the AbstractFunction2 class and implements the apply() method. It takes two objects of type MyVertexClass as input and returns an object of the same type. It compares the integer values stored in the two input objects and returns the one with the smaller value.

Exercise 4: Spark Graph Frames

This exercise aims to compute **PageRank** scores for the vertices of a given graph using **GraphFrame**, which is a Graph Processing library built on top of Apache Spark. Additionally, it is asked to optimally parameterize the PageRank algorithm, providing the values for the damping factor and the maximum number of iterations. The explanation of the code developed for this exercise is provided below:

The input graph is generated by using the information provided by the two text files: *wiki-vertices.txt* containing the articles of Wikipedia (which are the vertices of the graph) and their titles, and *wiki-edges.txt* containing the edges between the vertices. The first step in the code is to read the two text files and create two *RDDs*, one for the vertices and one for the edges. Then, two *StructTypes* are defined, one for the vertices and one for the edges, specifying the schema of the data. The schema for vertices contains two fields: *id* and *title* both of type *StringType*, which include the articles' ids and their title. Similarly, the schema for edges contains two fields: *src* and *dst* (again of type StringType), that will contain the *ids* of the connected vertices. After that, the

RDDs are being converted into *DataFrames* using the *createDataFrame* method of *SQLContext*, which is a Spark module used to work with structured data. The two *DataFrames* are then used to create a *GraphFrame* object with the apply method of *GraphFrame*. This *GraphFrame* object represents the input graph and will be used to run the *PageRank* algorithm on the graph.

The next step is to initialize a list called *timeList* which is storing the time taken for each configuration of the PageRank algorithm. The output schema of the data which is being gathered during the execution of the different configurations of PageRank, includes four fields: *damping_factor*, *reset_probability* (1 - *damping_factor*), *max_iterations*, and *time_taken_seconds*. In that way, we are able to understand which configuration of the algorithm will provide the optimal result, performance-wise.

The main loop of the code iterates over damping factor values from 0.05 to 0.95 in steps of 0.05. For each damping factor value, the code iterates over max iteration values from 5 to 20 in steps of 5. Then, PageRank algorithm is executed on the graph by using the current configuration (damping factor and max iterations). It measures the time taken for the current configuration, and stores all the information in the *timeList* list. Finally, it displays the top 10 vertices sorted by PageRank score.

To conclude, a DataFrame is created by using the *timeList* list and the output schema, and the DataFrame is written to a text file named *pageRank_log.csv*.The best configuration (based on the performance) and the results of the algorithm are presented here:

```
Configuration: damping factor = 0.45, max iterations = 5, time taken = 143.138 seconds


+------------------+--------------------+------------------+
|                id|               title|          pagerank|
+------------------+--------------------+------------------+
|8830299306937918434|University of Cal...|2230.7931323345792|
|1746517089350976281|Berkeley, California|1067.9703072109407|
|8262690695090170653|         Uc berkeley| 269.4965162383112|
|1735121673437871410|     George Berkeley|129.97812170393038|
|7097126743572404313|Berkeley Software...|128.29129066712173|
|8494280508059481751|Lawrence Berkeley...|118.18387542655265|
|6990487747244935452|      Busby Berkeley| 75.19220915551348|
|5820259228361337957|     Xander Berkeley| 53.38030894834924|
|1630393703040852365|        Berkeley, CA| 50.43130153761631|
|1899425950029509269|Berkeley County, ...| 48.98597723309792|
+------------------+--------------------+------------------+
only showing top 10 rows
```

Figure 1: Optimal Configuration and Results of PageRank

The *pageRank_log.csv* file includes the complete set of configurations tried. In our observation, we found that reducing the damping factor (which means increasing the reset probability) leads to quicker convergence of the output result with fewer maximum iterations. However, this also causes a decrease in the actual PageRank values. On the other hand, using a higher damping factor leads to a longer running time for the PageRank algorithm. As anticipated, increasing the number of iterations also increases the time it takes to complete the PageRank algorithm.