UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Semantic Data Management

Lab 2 – Distributed Graph Processing

Agustina Martinez
agustina.martinez@estudiantat.upc.edu

Emmanuel Werr
emmanuel.werr@estudiantat.upc.edu

Facultat d'Informàtica de Barcelona
Master of Data Science

April 10, 2023

1 Getting familiar with GraphX's Pregel API

Superstep 0
Within each iteration of the graph, any vertex that has learned a higher value from the received messages sends it to all its neighbors. In particular, the Integer.MAX_VALUE function allows us to automatically assign to any variable the maximum possible integer without needing to remember the exact number.
Initially V1 sends a message to V2 acquiring the new value of the message equal to 9, since this value of the vertex is greater than the destination vertex.
Vertex V2 is unchanged because the value of the vertex is lower than the rest of the vertices. (V2: 1 < V3: 6) and (V2: 1 < V4: 8)
Vertex V3 does not send any messages because its value smaller than the others vertices. (V3: 6 < V1: 9) and (V3: 6 < V4: 8)
Vertex V4 does not send any messages because it does not have any neighbors to send messages to.

Superstep 1
In this step only vertex 2 sends the message =9 that it received from vertex 1, since its value is smaller.
Regarding the rest of the vertices:
Vertex 1 does not generate message, because now the value of vertex 2 is equal, so it has no change.
Vertex 2 sends the message=9 to vertices 3 and 4. (V2: 9 > V3: 6) and (V2: 9 > V4: 8)
Vertex 3 does not send message, because its value is less than the rest of the connected vertices (V3: 6 < V1: 9) and (V3: 6 < V4: 8).
Vertex V4 does not send any messages because it does not have any neighbors to send messages to.

Superstep 2
In this last step, the vertices that receive messages are those corresponding to vertex 3 and 4 from vertex 2, since the value is greater than the current value of these vertices.
Vertex 1 does not send a message, because its value is equal to vertex 2 to which it is connected.
Vertex 2 does not send a message, since its value is now equal to the values of vertices 3 and 4.
Vertex 3 does not send message because its value is equal to the value of the vertices to which it is connected V1 and V4.

Superstep 3
In this last phase, each vertex has stopped. There are no new messages, so 9 is the maximum value in the graph.

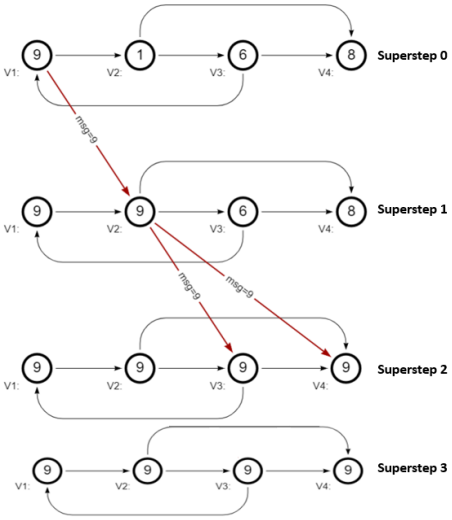

Figure 1: visualization of superstep actions during exercise 1 program run

2 Computing shortest paths using Pregel

In the second exercise, we are given the task of implementing the calculation of the shortest paths from a source vertex to a destination vertex in the graph provided, based on the modifications made in exercise 1.

For this purpose we have changed the Vprog function to keep the minimum value between existing and incoming message, in order to obtain the shortest path between vertices. It was also necessary to modify the sendMsg function in order to send messages containing the sum of all previous weights of the path. And finally regarding pregel, to have the results of the output we have added the sortBy function to show the required results as they are sorted.

```
Minimum cost to get from A to A is 0
Minimum cost to get from A to B is 4
Minimum cost to get from A to C is 2
Minimum cost to get from A to D is 9
Minimum cost to get from A to E is 5
Minimum cost to get from A to F is 20
```

Figure 2: output of exercise 2 program run

3 Extending shortest path's computation

To obtain each shortest path along with the total cost, we can use almost the same code as for exercise 2, but we need to modify some of the data structures used within the program and update them accordingly throughout.

Namely, we need to update the vertex properties to include a list of its shortest path from the origin vertex. We also need to update each of the TLAV functions in order to

perform the checks of when to send messages or not, and when to update both cost and path of its most recently found shortest path.

We also need to modify the output to include the entire shortest path for each source/destination vertices pair, and to add a line of code to order the outputs in alphabetical order of the vertex pairs. Once all these modifications have been made, the output of running the exercise 3 code yields the following output.

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

Figure 3: output of exercise 3 program run

4 Spark Graph Frames

For the last exercise, we are tasked with running the pagerank algorithm to find the top 10 most relevant wikipedia pages from provided data. We start by creating a graph of wikipedia pages from two .txt files containing the pages and their relationships. We can use some of the code provided in exercise4_warmup, but we will need to extend it.

In order to do this we must read from each file (there are many methods to do this but the preferred is by means of BufferedReader, which was also previously imported in the exercise 4 src code) and create the list of vertices (pages) and edges (page relationships). We then create RDDs from the vertices and edges, define their schemas, and instantiate the GraphFrame in order to run graph algorithms.

We have set "MAX_ITERATIONS" = 20 because we noticed pageranks approach convergence around 20 iterations so there is no need to run for longer. And we have set "DAMPING_FACTOR" = .85 because this is the standard value for damping factor. The damping factor is used to set the probability of whether a random surfer of the graph will either continue following vertex edges, or jump to a random vertex on the graph. This allows us to combat the effects of "sink holes" (vertices with no outgoing edges) on our pagerank values. Max iterations allows us to control the max number of times the pagerank algorithm runs without reaching convergence. It is important to optimize this value in order to prevent wasting compute resources. This is especially critical for applications with much larger and more complex graphs. We experimented with using different values for damping factor and they yield different pagerank values for our vertices, but the order of the top pages remains unchanged, so we decided to leave this value at standard.

After performing the modifications and putting together all parts of the code, initial run yields seemingly correct pagerank values for vertices. But we get the following warnings:

```
WARN TaskSetManager: Stage 5099 contains a task of very large size (176 KB). The maximum recommended task size is 100 KB.
WARN TaskSetManager: Stage 5241 contains a task of very large size (176 KB). The maximum recommended task size is 100 KB.
WARN TaskSetManager: Stage 5243 contains a task of very large size (176 KB). The maximum recommended task size is 100 KB.
WARN TaskSetManager: Stage 5673 contains a task of very large size (153 KB). The maximum recommended task size is 100 KB.
```

Figure 4: initial warnings of exercise 4 program run

After some research into the problem, we realized we can specify the number of slices spark will partition our data into during the creation of the JavaRDD. We proceeded to set the 'numSlices' parameter to the minimum value that caused the size of each task to be less than 100KB. This number turned out to be 15 slices. After setting the correct value, we receive no warnings and get only the pagerank algorithm output from our program.

```
+------------------+------------------+-----------------+
|                id|             title|         pagerank|
+------------------+------------------+-----------------+
|8830299306937918434|University of Cal...| 3124.234891219508|
|1746517089350976281|Berkeley, California|1572.4654902702382|
|8262690695090170653|         Uc berkeley| 384.2585598150019|
|7097126743572404313|Berkeley Software...|214.06422079273588|
|8494280508059481751|Lawrence Berkeley...| 193.7021786178606|
|1735121673437871410|     George Berkeley| 193.6699517779405|
|6990487747244935452|      Busby Berkeley|113.23640348368971|
|1164897641584173425|      Berkeley Hills| 105.9199827619886|
|5820259228361337957|     Xander Berkeley| 71.84579008610395|
|6033170360494767837|Berkeley County, ...| 68.48570379224931|
+------------------+------------------+-----------------+
only showing top 10 rows
```

Figure 5: output of exercise 4 program run