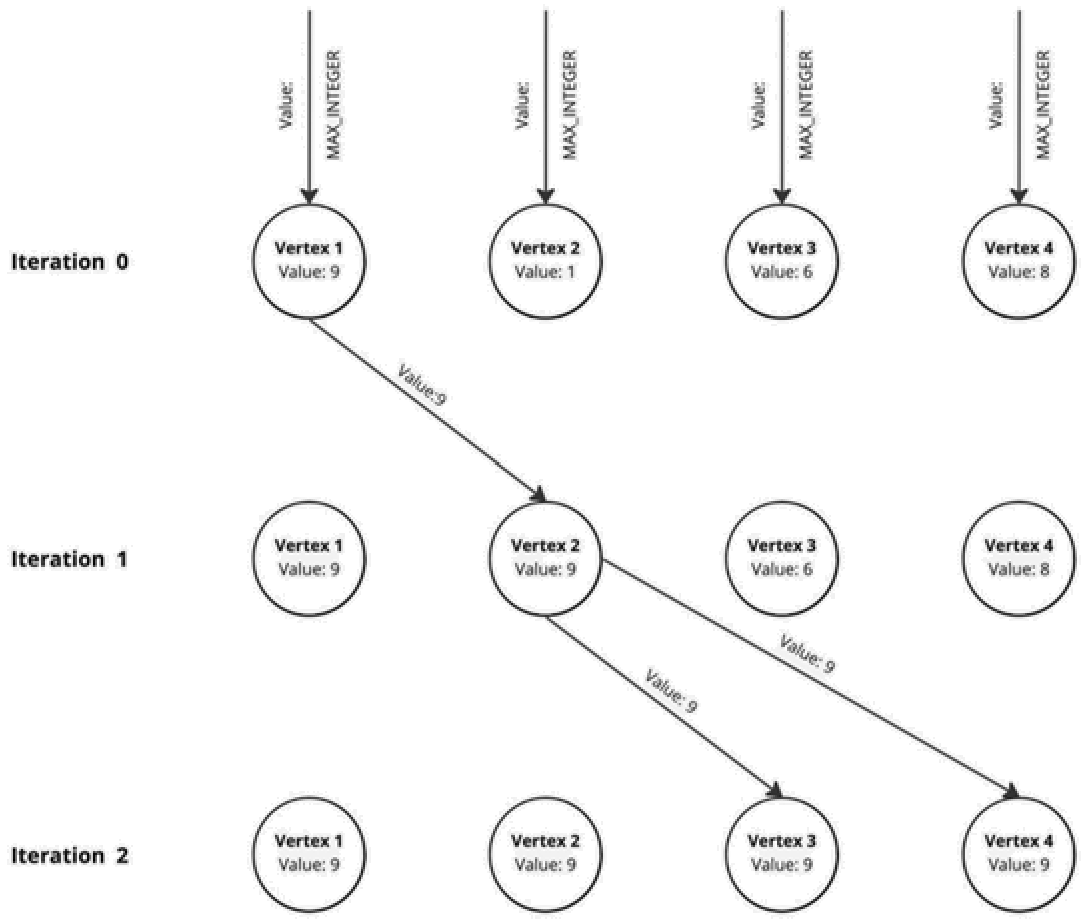
Exercise 1



All nodes receive the MAX_INTEGER value in the first iteration. By doing this, we can identify the superstep 0 and activate all nodes.

Since this is a toy example, there are very few activations. Every activated node sends a message if and only if the receiver has a lower value. This means that only vertex 1 sends a message after the first iteration, and vertex 2 updates its corresponding value to 9, and activates itself.

During the second superstep, vertex 2 sends its previously updated value 9 to its neighbours: vertices 3 and 4, which are also updated. Since they can't send a value to any other node (remember that we know in advance the values of surrounding vertices), the algorithm terminates, leading to the expected result.

Exercise 2

This exercise consisted in implementing the VProg, sendMsg and merge functions:

- **VProg**: in this function, we always return the vertexValue. This value is updated if the received message has an absolute value higher than the actual vertex value.
- **SendMsg**: when a message is sent from one node to another, this function sums the edge weight (*triplet.attr*) to the vertexValue.
- **Merge**: this chunk of code is the one dealing with the calculation of minimum values. It merges all paths and selects the one which results in the minimum cost.

Exercise 3

This exercise demanded returning the cost of the shortest path and the nodes that form it. To achieve this, we modified the value stored in the nodes: instead of having Integer numbers, it is now a list of *Long* numbers.

Many possibilities were considered, for example, having a tuple of *Integer* and *List<Long>* as vertex value. We thought it the best solution since it keeps the vertex value structure as simple as possible. Since it will be sent and processed many times, this "basic unit" remains primitive. The first value of the list stores the shortest path cost in *Long* format (without conversion loss), and the rest of the elements store the node id, which is naturally defined as a long integer.

Instead of having a cost of *Integer.MAX_VALUE*, we use *Long.MAX_VALUE* as default cost values. We also use *Long.MAX_VALUE* for the initial sent message, which is a list containing only the cost. Of course, the fixed node has a value of 0 as cost. Therefore, default vertex values are lists with max *Long.MAX_VALUE* and the node id appended at the end.

Functions are modified as follows:

- **VProg**: this function always returns the vertex value. If the first position (0) of the list (i.e. cost) is greater than the one received in the message, the current node id is added at the end of the list, which is stored in the node as vertexValue.
- **SendMsg**: when a message is sent from one node to another, this function sums the edge weight (*triplet.attr*) to the shortest path cost (i.e position 0 of the List stored in vertexValue).
- **Merge**: this chunk of code is the one dealing with the calculation of minimum values. It merges all paths and selects the one which results in the minimum cost at position 0. Since we are no longer interested in only the cost, we return the whole vertexValue List.

Exercise 4

To optimise the parameters of the PageRank algorithm, we underwent an iterative process of testing different damping factors and maximum iterations. Initially, we used a high damping factor of 0.99 and 20 iterations. We then progressively reduced the damping factor while adjusting the maximum number of iterations to maintain similar algorithm results.

In this report, we present three tables that represent different iterations of the algorithm. The first table uses the initial damping factor of 0.99 and 20 iterations. The second table presents an intermediate solution with a damping factor of 0.95 and 10 iterations, yielding similar results to the initial table. The third and final solution uses a damping factor of 0.85 and a maximum of 5 iterations. We selected this final solution because it retained the top 10 pages with only slight variations in the page rank scores. Additionally, a lower damping factor generally requires fewer iterations, making this solution more efficient.

id	name	pagerank
8830299306937918434	University of California, Berkeley	3229.40
1746517089350976281	Berkeley, California	1726.32
8262690695090170653	Uc berkeley	411.06
7097126743572404313	Berkeley Software Distribution	246.71
8494280508059481751	Lawrence Berkeley National	225.13
1735121673437871410	Laboratory George Berkeley	208.22
1164897641584173425	Berkeley Hills	155.14
6990487747244935452	Busby Berkeley	122.24
6033170360494767837	Berkeley County, South Carolina	91.16
5820259228361337957	Xander Berkeley	70.44

Damping factor 0.99 and maximum iterations 20.

id	name	pagerank
8830299306937918434	University of California, Berkeley	3168.43
1746517089350976281	Berkeley, California	1693.73
8262690695090170653	Uc berkeley	403.34
7097126743572404313	Berkeley Software Distribution	239.60
8494280508059481751	Lawrence Berkeley National	222.92
1735121673437871410	Laboratory George Berkeley	200.14
1164897641584173425	Berkeley Hills	153.30
6990487747244935452	Busby Berkeley	106.74
6033170360494767837	Berkeley County, South Carolina	76.07
5820259228361337957	Xander Berkeley	64.79

Damping factor 0.95 and maximum iterations 10.

id	name	pagerank
8830299306937918434	University of California, Berkeley	3293.59
1746517089350976281	Berkeley, California	1811.95
8262690695090170653	Uc berkeley	429.15
7097126743572404313	Berkeley Software Distribution	270.09
8494280508059481751	Lawrence Berkeley National	228.16
1735121673437871410	Laboratory George Berkeley	228.08
1164897641584173425	Berkeley Hills	159.55
6990487747244935452	Busby Berkeley	148.05
6033170360494767837	Berkeley County, South Carolina	89.91
5820259228361337957	Xander Berkeley	85.33

Damping factor 0.85 and maximum iterations 5.