Lab 5
Parallel and Distributed Algorithms

Prim's algorithm for Minimum Spanning Tree

Authors: Adam Brzeski, Karol Draszawka
Date: 5.05.2016

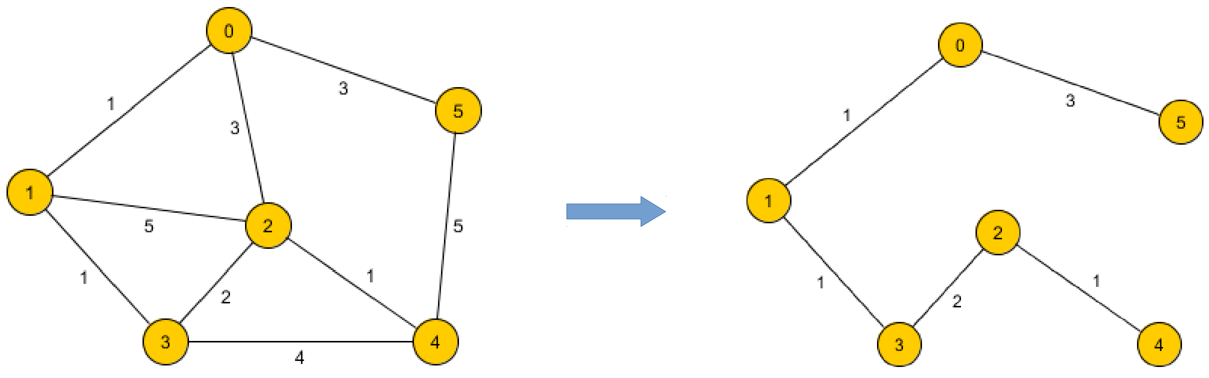
1 Aims of the laboratory

The aim is to practice distributed programming skills by implementing distributed version of Prim's algorithm for finding Minimum Spanning Tree (MST). The implementation will utilize the provided java framework for simulating distributed system, which was already used in the previous classes.

The code for this lab is in Lab05.zip file. Extracted folder contain a project for Netbeans IDE. The source can be however easily imported to other IDEs (Idea, Eclipse).

2 Reminder – Minimum Spanning Tree

A minimum spanning tree in a connected, undirected graph is a tree that connects all of the vertices of the graph and the total weight of the used edges is minimal. An example of MST found in a graph is presented below:



input graph G

minimum spanning tree

(total weight = 8)

3 Distributed Prim's algorithm

Prim's algorithm for finding MST is an iterative, greedy algorithm. For implementing the distributed version of the algorithm, we assume the graph is represented in a form of an adjacency matrix A. However, the adjacency matrix is distributed over nodes, meaning that each node has access only to a range of columns of the matrix. An example of the partition of the A matrix is presented below:

$$A=\begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

Adjacency matrix of the graph G

$$A=\begin{bmatrix} 0 \\ 1 \\ 3 \\ \infty \\ \infty \\ 3 \end{bmatrix}\begin{bmatrix} 1 & 3 \\ 0 & 5 \\ 5 & 0 \\ 1 & 2 \\ \infty & 1 \\ \infty & \infty \end{bmatrix}\begin{bmatrix} \infty \\ 1 \\ 2 \\ 0 \\ 4 \\ \infty \end{bmatrix}\begin{bmatrix} \infty & 3 \\ \infty & \infty \\ 1 & \infty \\ 4 & \infty \\ 0 & 5 \\ 5 & 0 \end{bmatrix}$$

Partition of the adjacency matrix for 4 processes
(the matrix parts are not equal)

The A matrix columns assigned to a process represent edge weights of a set of vertices. In each step of the Prim's algorithm, each node is responsible for evaluating its local vertices and finding the one with the lowest weight connecting it to any of the already visited vertices. The detailed description of the distributed Prim's algorithm can found at: http://parallelcomp.uw.hu/ch10lev1sec2.html

The algorithm includes the following steps:

```
1.  Init visted array with an arbitrary vertice (e.g. vertice 0)
2.  Init local d[] array (representing current distances of local vertices to any of the visited vertices)
3.  While (not all vertices visited)
4.      Determine the local vertice with the lowest distance to any visited vertice
5.      Reduce – compare the vertices found by the nodes and choose the one with lowest weight (performed by master node)
6.      Broadcast – send the chosen vertice to all nodes (performed by master node)
7.      Update visited and d[] arrays (the chosen vertice is now considered visited, so it affects the distances of the remaining vertices)
```

4 Student's task

The task is evaluated with two test named testParallelPrim() and testScalability() provided in the testAll() method of labs.Lab05 class. In order to pass the tests, it is required to implement the following method of the algorithms.shared.GraphAlgorithms class:

```
public static void findMSTPrim(Node node, boolean printResult)
```

The implementation should write the total weight of the MST to mstTotalWeight variable in order to pass the tests.

The testAll() method contains also 2 additional tests:

- testSerialPrim() - tests the serial implementation of the Prim's algorithms, which is already provided in method findMSTPrimSerial(). Can be used as a reference of the Prim's algorithm
- testEdgesCommunication() - illustrates sending the edges between the nodes using reduce and broadcast operations

5 Cheat Sheat

`nNodes = node.getNumberOfAllNodes()`	The number of nodes
`node.A`	Part of the adjacency matrix assigned to the node
`node.A.getNCols()`	The number of vertices assigned to the node
`node.A.getNRows()`	Total number of vertices in the graph
`(myId*nVertices)/nNodes`	Index of the first assigned vertice in the array of all vertices

`Edge`	Class describing an edge
`Edge.serialize()`	Serialize the edge for sending
`Edge.deserialize(data)`	Deserialize edge from received data

`BasicCommunication.reduce(node, data, operator)`	Reduces the distributed data parts to a single part by applying given operator. The result is returned only in the node 0
`BasicCommunication.broadcast(node, data)`	Broadcast the data from the node 0 to all other data. The return value contains the broadcasted data