SE205 - TD6 Énoncé

General Instructions

- You can download all source files from:
  `https://se205.wp.mines-telecom.fr/TD6/`

1 Simple Hello World (20 minutes)

Aims: *Compile and run an application with Akka actors.*

Download the source code for today's TD and decompress the archive. Then, inspect the source code of the example in the sub-directory `src/td6_1`.

- How many actors are involved in this simple Hello World program?
- Draw a diagram representing their interactions.
- Compile the source code with the `ant` tool. Use the target `compile` and `jar`, e.g., `ant compile`.
- Run the compiled program, again, using the `ant` tool and the target `run-hello`.
- Modify the `Greeter` actor so that it prints its own path as well as the path of the sending actor whenever it receives a `GREET` message.
- Modify the `HelloWorld` actor in order to change the name of the `Greeter` actor (you may choose any name you like).

2 Asynchronous Bellman-Ford (70 minutes)

Aims: *Implement a non-trivial algorithm using actors.*

The goal of this exercise is to implement the Bellman-Ford algorithm using asynchronous message passing. The algorithm computes the length of the shortest path from a given root node to each other node in a directed graph. The length of a path is the sum of the weight associated with each edge on the path. For the example below, for instance, the algorithm would yield a shortest path of length 7 from node N1 to N4. While the shortest path from N1 to N3 evaluates to 8.

$$\begin{tikzcd}
\text{N1} \arrow[r, "5"] \arrow[rrr, bend right, "7"'] & \text{N2} \arrow[r, "3"] & \text{N3} \arrow[r, "2"] & \text{N4}
\end{tikzcd}$$

A description of the sequential algorithm and an interactive website demonstrating its operation are available at the following website:

```
https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html
```

In an asynchronous implementation of the algorithm the nodes and edges of the graph are simply actors exchanging messages. The messages themselves represent the lengths of paths originating at the root node. These lengths are iteratively refined as shorter paths are discovered. We will assume that three kinds of actors exist in the system: (1) `Graph`, (2) `Node`, and (3) `Edge`.

The `Graph` actor represents the entire graph (including edges and nodes) and controls the computation. In our case it will read the definition of the weighted graph from a file, where each line defines an edge as follows:

```
[source node name] =[edge weight]> [destination node name]
```

For instance, the line "`a =5> c`" defines an edge from a node named "`a`" to another node named "`c`" with weight 5. Nodes are defined implicitly, i.e., each node name appearing in the definition of an edge implicitly defines the node. An example file for the graph shown in the figure above is included in the downloaded source code package (`data/simple.graph`). The root node is given by the source node of the first edge. Each node tracks the shortest path from the root node to itself using a member variable (`shortestPathWeight`).

The asynchronous Bellman-Ford algorithm then consists of exchanging messages between `Node` and `Edge` actors representing the graph. The computation starts when the `Graph` actor send a message 0 to the root node, i.e., the shortest path from the root node to itself obviously has length 0. In response to this message, the root node updates its `shortestPathWeight` variable and then send a message to all out-going edges informing them that a shorter path was discovered. The `Edge` actors in response to this message send a message to their respective destination nodes by adding their own weight to the weight of the in-coming message. The destination nodes in turn update their `shortestPathWeight` variables if the path is indeed shorter and send messages to their out-going edges. It is easy to see that this processing will stabilize at some point and gives the correct length of the shortest path at each node.

For the example graph from above, the `Graph` actor would send a message 0 to the `Node` actor N1, which updates its `shortestPathWeight` variable and sends messages containing 0 to the actors representing its out-going edges N1 $\rightarrow$ N2 and N1 $\rightarrow$ N4. The `Edge` actors send messages 5 $(0+5)$ and 7 $(0+7)$ to the node actors N2 and N4. Both update their `shortestPathWeight` variables. N4 does not have any out-going edges and thus is done. N2, however, will send a message 5 to the actor representing its out-going edge, which in turn will send a message 5 $(5+3)$ to `Node` actor N3. N3 will update its `shortestPathWeight` variable and send a message 10 $(8+2)$, via the `Edge` actor N3 $\rightarrow$ N4, to N4. N4 already knows a shorter path and thus is done. No further messages are sent and the algorithm ends.

A basic code skeleton of the asynchronous Bellman-Ford algorithm is available with the source that you downloaded, in sub-directory `src/td6_2`. The code skeleton does not compile or run in its current form. You need to complete the source code by following the steps below:

- The provided code skeleton consists of four Java source files in the sub-directory `src/td.2`. Carefully read the source code and try to understand its operation.

- Given the provided code and the description of the algorithm from above, draw a diagram representing the messages exchanges between the various actors in the system.
- Try to compile and run the given source code using `ant` and the target `run-bf`. This will fail, since some methods are currently not implemented.
- Implement the `preps` method of the `Edge` class. Try to re-run/compile the program and check which functions are missing.

  Hint: See the lecture slides for an example.
- Implement the message handling method of the `Edge` class. Your code should only handle messages containing integer values and signal all other kinds of unexpected messages as indicted in the lecture. Once the actor receives a message containing an integer value, add the edge's own weight to the in-coming value and send the result to the destination node.

  Hint: See the lecture slides to get started. You can use the `instanceof` operator to check the type of the in-coming message.
- Implement the message handling method of the `Node` class. Your code should only handle messages containing integer values and signal all other kinds of unexpected messages. Once the actor receives a message containing an integer value, compare the shortest path known for the node (`shortestPathWeight`) with the in-coming value. If the length of newly discovered path is shorter, update the node's member variable and send a message to its out-going edges.

  Hint: Use `getContext().actorSelection()` in order to find all edges originating at the current node. You need to construct a string pattern containing a part (`substring`) of the `Node` actor's name (`getSelf().path().name()`). See the `Graph` class to see how `Edge` and `Node` actor names are constructed.
- Your code should now compile and run. Verify by running `ant` using the target `run-bf`.
- Now reread the source code and determine how the `Graph` actor learns the lengths of the shortest paths from each node.
- Finally, try to determine how and when the actor application terminates? Is there a way for the `Graph` actor to determine that the shortest path lengths of the various nodes have stabilized?