DISCO - Stochastic Network Calculator

Alpha-Version, April 2014

Dipl.-Math. Michael Alexander Beck

June 16, 2014

How does it work?

The DISCO - Stochastic Network Calculator (SNC) is based on the theory of $(\sigma(\theta), \rho(\theta))$-calculus, which can be found in [**?**] and more detailed in [**?**]. As a typical network calculus the $(\sigma(\theta), \rho(\theta))$-calculus allows two operations, which are crucial for the SNC: First the posibility to compute output bounds. This means if a node is a dynamic S-Server and it serves a flow A, then the output can be $(\sigma(\theta), \rho(\theta))$-bounded, if the S-Server and the flow can be $(\sigma(\theta), \rho(\theta))$-bounded. This description of the output allows to interpret it as an input flow to another S-Server and iterate this procedure. The second important operation is subtracting a flow A from a S-Server, to calculate the leftover service S_l, which is available for another flow B.

These two operations allow to analyse feed-forward networks, as long as there are $(\sigma(\theta), \rho(\theta))$-bounds on the initial arrivals and initial nodes. The network calculator does this by performing the following steps in the *simple analysis*:

1. For each node is checked, wether their arrivals are $(\sigma(\theta), \rho(\theta))$-bounded. If for all of their arrivals such a bound is known, the node is pushed into a stack.

2. Until a $(\sigma(\theta), \rho(\theta))$-bound for the flow of interest *at* the node of interest and a $(\sigma(\theta), \rho(\theta))$-bound at the node of interest *for* the flow of interst is known the following steps are performed:

   a) The first node in the stack is popped and:

   b) An output bound for the flow with highest priority is calculated.

   c) If this leads to the case that for all of a node's arrivals $(\sigma(\theta), \rho(\theta))$-bounds are known, then this node is pushed into the stack.

   d) A leftover service is calculated.

   e) If there is another flow to serve the node is pushed again.

   f) If the stack runs out of nodes before the flow and node of interest is found, the network has not been feed-forward and no bound can be given by this analysis.

3. From the $(\sigma(\theta), \rho(\theta))$-bounds for the flow of interest and node of interest the needed backlog-/delay- or output-bound can be calculated.

If a bound is found it is always given in the compact *arrival-description*:

- Output bounds are given directly by its $(\sigma(\theta), \rho(\theta))$ description.
- Backlog bounds are given by $(\sigma(\theta, x), 0)$ such that:

$$\mathbb{P}(q(n) > x) \leq e^{\theta\sigma(\theta,x)}$$

- Delay bounds are given by $(\sigma(\theta)\rho(\theta))$ such that:

$$\mathbb{P}(d(n) > N) \leq e^{\theta\rho(\theta)N+\theta\sigma(\theta)}$$

The found bound is then given to an optimizer. So far there exists only the *simple optimizer*, which is just a discretization of the search space and calculating the found bound for every value of the discretized search space. From all of these results the best one is choosen.

What to consider, when writing your own arrivals, service, analyses or optimization?

So far the SNC only knows about constant rate servers, constant rate arrivals, arrivals with i.i.d. exponentially distributed increments and the above mentioned analysis and optimization methods. You might however want to implement your own kind of any of these. Here I describe, where changes are needed (in the core of the program, as well as in the GUI) to do this.

Writing your own arrival

If you want to consider something different to exponentially distributed increments you first need a corresponding bound on this flow, i.e. if the flow is denoted by A you want to find $\sigma(\theta)$ and $\rho(\theta)$ such that:

$$\mathbb{E}(e^{\theta A(m,n)}) \leq e^{\theta\rho(\theta)(m-n)+\theta\sigma(\theta)}$$

The found functions σ and ρ are represented by classes implementing the `functionIF`-interface. These classes are best placed in `unikl.disco.mgf`. Normally σ and ρ are dependent on further parameters (e.g. the rate λ of an exponential distribution). This further parameters should be initialized in the constructor. The `getValue`-method must return for a given `theta` the value $\sigma(\theta)$ or $\rho(\theta)$ respectively.

After defining the classes corresponding to σ and ρ you can then use them to construct an `Arrival` with them and use these normally in the core of the program.

If you want to add your new kind of arrivals to the GUI a few more changes are needed. Go into the class `FlowEditor` in `unikl.disco.mgf.GUI` and do the following:

- After the `final String EXPONENTIAL`... add your arrival type.
- Also add your arrival type in the String-array: `arrivalBox`.
- A few lines later you need to define your own `JPanel`, with fields to set the parameters of your flow (compare the other cards for examples). Don't forget to also add this new card by `topCardContainer.add(card3, YOUR − ARRIVAL)`.
- All left to do now is adding the case, that your kind of arrival was choosen into the `ActionListener` of the OK-Button. For this go to the comment right before the first appearance of the command `dispose()`. Construct a new case, which initializes your rho- and sigma-function, by using the input from the card you have defined in the previous step.

Now your new type of arrivals can also be constructed from the GUI.

Writing your own service

Writing your own kind of service is very similar to writing your own kind of arrivals. Again you need first a bound on your service, i.e. if S is your dynamic S-server, you need σ and ρ sucht that:

$$\mathbb{E}(e^{-\theta S(m,n)}) \leq e^{\theta\rho(\theta)(n-m)+\theta\sigma(\theta)}$$

To implement σ and ρ see the first steps of the previous section. You can use them directly by constructing a `Service` with them. Again, if you want to use your new service also in the GUI same changes are needed. Go herefor in the class `VertexEditor` in `unikl.disco.mgf.GUI` and do the following:

- Add your service type after `final String CONSTANT`...
- Also add your service type in the String-array: `serviceBox`.
- A few lines later you need to define your own `JPanel`, with field to set the parameters of your service. Don't forget to also add this new card by `topCardContainer.add(card2, YOUR − SERVICE)`.
- As above you need to modify the `ActionListener` of the OK-Button. For this go right before the first appearance of the command `dispose()`. Construct a new case, which initializes your rho- and sigma-function, by using the input from the card you have defined in the previous step.

Now your new type of service can also be constructed from the GUI.

Writing your own analysis

Your own kind of analysis should be placed in `unikl.disco.mgf.network` and implement the `Analyzer` interface. Furthermore, it is recommended to extend the `AbstractAnalysis`-class to get started with your own analyzer. The core of your analysis lies in the method `analyze`(). Make sure, that the output of `analyze`() is the searched bound in the above described *arrival-description*. The next thing you need to do is modify the `SNC`-class in `unikl.disco.mgf`:

- Go to the `enum` called `AnalysisType` and add another element, representing your type of analysis, together with a string-representation.
- Add the new analysis to the switch-case in the `AnalysisFactory`

Writing your own optimization

Your own kind of optimization should be placed in `unikl.disco.mgf.optimization` and implement the `Optimizer` interface. A good way to start your implementation is to extend the `AbstractOptimizer`-class. The core of your analysis lies in the `minimize`(...) method, the methods `Bound`(...) and `ReverseBound`(...) are a leftover and considered deprecated. The `minimize`(...) method needs a given granularity for θ and appearing Hölder parameters. Further parameters are given to the your optimizer directly through the constructor in the `OptimizationFactory`. There, among others, the bound-to-be-optimized is provided as input. Note, that every input to optimizers is wrapped in the `Optimizable` interface, which serves as a uniform abstraction on which all optimization methods can be built. This yields less coupled and more modular code because the different optimizers are able to minimize any function which implements the `Optimizable` interface.

Last but not least, your new optimization method has to be added to the `OptimizationType` enum as well as to the switch case in the `OptimizationFactory`.

References