SOL Documentation

Release 0.9

Victor Heorhiadi

Oct 20, 2019

Contents

SOL is a library that lets you rapidly prototype *network management applications* that require constructing an **optimization**. It is designed to work well with Software Defined Networking (SDN) as it makes use of the global view of the network to compute a globally optimal (or near-optimal) solution.

CHAPTER 1

Why optimizations?

Optimization is incredibly common in the networking domain. Classic problems such as shortest path routing and maxflow can all be expressed as linear programs and solved efficiently.

Traffic engineering, middebox management, and other types of load balancing can also be expressed using optimizations.

CHAPTER 2

Key features

- Fast prototyping of optimizations for network management
- Composition of multiple optimization applications using different fairness modes
- Flexible resource computation logic
- Integrations with ONOS SDN controller
- Novel optimization capabilities, reusable across different applications,(e.g., reconfiguration minimization)

CHAPTER 3

Integrations

SOL is desgined to be modular and could potentially integrate with multiple SDN controllers. This library contains the core optimization logic. It can be used on its own to quickly prototype applications, compose_apps multiple optimizations and examine resulting solutions.

A rough view of the SOL library and integrations is as follows:

- SOL library (this project/repository)
- ONOS integration Allows use of the SOL library from the applications running on top of ONOS controller
- SOL workflows A collection of examples and workflows to give users an idea of how SOL can be used
- SOL-viz A nifty tool for interactive vizualization of SOL solutions

CHAPTER 4

Python documentation

4.1 Getting started with SOL

4.1.1 Installing SOL

Supported python versions

Both python 2.7 and 3.5 are supported, although most testing was performed with python version 3.5 only. We encourage you to also embrace the python 3.x movement.

Dependencies

SOL has multiple dependencies, most of which can be easily installed automatically using *pip* (or *conda* or similar package manager):

```
pip install -r requirements.txt
```

however TMgen and Gurobi must be **installed manually**.

While Gurobi is a commercial product, free academic licensing is available

Optional but recommended

1. Anaconda by Continuum.io It has many of the required scientific python packages and even an easily installable gurobi Python package.
2. We also recommend setting up a virtualenv (either using *conda* or virtualenv) for convenience.

Download and install SOL

The code is publicly available at https://github.com/progwriter/SOL

1. Clone it using:

```
git clone https://github.com/progwriter/SOL
```

2. Install SOL development mode:

```
pip install -e .
```

4.1.2 Understanding fundamental inputs to SOL

SOL requires the following things to correctly do its job:

- Network topology
- Information about network traffic
- The application's optimization goals and constraints

In this quickstart guide we describe the necessary inputs and show how to create a simple application. We will go over how to construct a very small network and create a simple maxflow optimization.

Feel free to follow along using the Jupyter notebook that accomapnies this written guide.

Topology

The `sol.Topology` class represents the network as a graph.

Each node (vertex) in the graph denotes a switch/router and is identified by an integer ID. The topology also stores data about network functions and resources. That is, each node will have attributes indicating whether it has a middlebox attached to it, what functions it performs (e.g., 'firewall') and any resources/capacities associated with this node.

Each edge, naturally, represents a link between any two given nodes (a tuple of ints) and is also allowed to have resources/capacities asscociated with it.

Lets look at some examples. SOL includes some primitive topology generators:

```
>>> from sol.topology.generators import chain_topology
>>> t = chain_topology(5)
>>> list(t.nodes()) # topology nodes
[0, 1, 2, 3, 4]
```

Lets add a middlebox to node 3, and make it a firewall:

```
>>> t.set_mbox(3)
>>> t.add_service_type(3, 'firewall')
>>> t.set_resource(3, 'cpu', 3000)
>>> list(t.nodes(data=True))
[(0, {'resources': {}, 'services': 'switch'}),
 (1, {'resources': {}, 'services': 'switch'}),
 (2, {'resources': {}, 'services': 'switch'}),
 (3,
  {'hasMbox': 'True',
   'resources': {'cpu': 3000.0},
   'services': 'firewall;switch'}),
 (4, {'resources': {}, 'services': 'switch'})]
```

Topologies can also be created by loading existing data from disk. GraphML and GML formats are supported. Note that to store resources GML format should be used, as it supports nested attributes for nodes and edges.

```
>>> from sol import Topology
>>> t.write_graph('mytopo.gml')
>>> t2 = Topology('mynewtopo')
>>> t2.load_graph('mytopo.gml')
>>> list(t2.nodes(data=True)) # all the data is preserved
[(0, {'resources': {}, 'services': 'switch'}),
 (1, {'resources': {}, 'services': 'switch'}),
 (2, {'resources': {}, 'services': 'switch'}),
 (3,
  {'hasMbox': 'True',
   'resources': {'cpu': 3000.0},
   'services': 'firewall;switch'}),
 (4, {'resources': {}, 'services': 'switch'})]
```

See full topology API in *Topology* section. For now, let us simply define link capacities in the network to be a 100 units (imagine it's Mb/s)

```
>>> for link in t.links():
>>>     t.set_resource(link, 'bandwidth', 100)
```

Traffic Classes

Traffic classes contain information about the type of traffic being routed through the network. The optimization later will determine how to best route this traffic, but to do so it needs to know entrance and exit points for traffic and its volume. Therefore, at a minimum, each traffic class must contain a source node, a destination node and volume of traffic (i.e., number of flows). For example:

```
>>> from sol import make_tc
>>> make_tc(0, 4, 1000) # a traffic class from node 0 to node 4 with 1000 flows
TrafficClass(tcid=0,name=,src=0,dst=4)
```

You can construct traffic classes directly, however you will need to keep track of traffic class ids (they must be sequential) and provide volumes as numpy arrays:

```
>>> from sol import TrafficClass
>>> import numpy
>>> tc = TrafficClass(tcid=1, name='myclass', src=0, dst=4, vol_flows=numpy.
↪array([1000]))
```

Detailed explanation for this is given in *Traffic Classes* section of the User's Guide.

Paths (per traffic class)

Each traffic class is assigned a set of valid paths. Generating and filtering paths using *predicates* is how policies are enforced. Usually, this is a one time, offline step. Any sufficiently complex application will implement its own predicate generate paths and store them for future use. In this simple guide, we will just generate paths on-the-fly using one of SOL's helper functions, since there are no policy requirements on which paths the traffic must take in the maxflow problem.

```
>>> from sol.path.generate import generate_paths_tc
>>> pptc = generate_paths_tc(t, [tc]) # get our earlier topology and put the traffic␣
↪class in a list
>>> pptc
<sol.path.paths.PPTC at 0x10dc00f98>
```

Let us treat *pptc* as an opaque object for now. You will need it to construct the application; We will detail the need for `sol.PPTC` class and its capabilities in the *Paths* section of the User's Guide.

Applications

Once the paths per traffic class have been configured, we can proceed to create a basic optimization. Let's start with a very simple maxflow problem.

```
from sol import AppBuilder
from sol.opt.funcs import CostFuncFactory
from sol.utils.const import Objective

builder = AppBuilder()
# Create a cost function where each flow consumes 1 Mb/s regardless of traffic class
cost_func = CostFuncFactory.from_number(1)
app = builder.name('maxflowapp')\
    .pptc(pptc)\
    .objective(Objective.MAX_FLOW)\
    .add_resource('bandwidth', cost_func, 'links')\
    .build()
```

The application builder allows us to set the *pptc* of the application, use a pre-defined maxflow objective function, as well as set the routing cost of traffic. In this example, each flow consumes a unit of bandwidth. SOL provides a convenitent way of specifying that using the `sol.opt.funcs.CostFuncFactory`

4.1.3 Optimization

With a single app

The optimization is constucted using the topology and the application:

```
from sol import from_app, NetworkConfig, NetworkCaps

caps = NetworkCaps(t) # Create network caps from the topology
caps.add_cap('bandwidth', cap=.5) # We can use 50% of link capacities
nconfig = NetworkConfig(networkcaps=caps)
opt = from_app(t, app, nconfig) # create the optimization
opt.solve() # solve the optimization
```

With multiple apps

Refer to the *Composition of multiple applications* part of the User's Guide.

4.1.4 Examining the solutions

The two main ways of examining the solution are:

1. Looking at the value of the objective function
2. Extracting the paths responsible for carrying traffic.

1. To see the objective function value, simply run:

```
>>> opt.get_solved_objective(app)
0.5
```

As expected, we can route 50% of the traffic, due to the link caps.

2. To extract the paths

```
>>> pptc_solution = opt.get_paths()
>>> print(pptc_solution.paths(tc1))
[Path(nodes=[0 1 2 3 4], flowFraction=0.5)]
```

This is exaclty what we expected, traffic goes from node 0 to node 4 and we can manage to carry only 50% of it.

Congratulations! You are done with the tutorial. For more info head to *User's guide* for detailed instructions on how to construct more complex applications and utilize full potential of SOL.

4.2 User's guide

This guide shows how to use all of the SOL's features and explains the internals of SOL in more detail.

For a quick tutorial, please read the *Getting started with SOL* section first. In this guide, we elaborate on the functionality described in that tutorial.

Note: At times this guide will assume reader's familiarity with optimization terminology and also the numpy library

4.2.1 Network Topology

Network topologies are represented as directed graphs using the networkx library. In addition to

4.2.2 Traffic Classes

To create a traffic class, you have two options: directly calling the traffic class constuctor or relying on the `sol.make_tc()` function, which is syntactic sugar with internal state keeping track of traffic class ID numbers.

If using *make_tc*, specify only source and destination nodes, and volume in flows.

```
>>> from sol import make_tc
>>> make_tc(0, 4, 20) # a traffic class from node 0 to node 4 with 20 flows
```

If using the contructor, specify the ID (must be unique and sequential), name, src/dst nodes and volumes.

Warning: Do not mix the two methods of traffic class creation, as `sol.make_tc()` function maintains an internal traffic class ID counter, which will become out of sync if

4.2.3 Paths

Lets discuss how path generation and selection affects the outcomes of the optimization. Paths are an important component to understaning how to enforce network policies and optmization decision making.

Path generation

First and foremost we must generate paths that connect our ingress point to our egress points.

Handling Middleboxes

All of the generated paths are then passed through a path predicate, to determine if the path is valid. *This is how policies are enforced.*

Path predicates

A predicate is a function that decides whether a path is allowed or not. SOL provides some built-in predicates, but it is very easy to write your own. A function with the following signature is a valid predicate:

```
def predicate(path, topology)
    return True # or False
```

In fact, that is the implementation of the `sol.path.predicates.null_predicate()`, a predicate that considers every path to be valid.

Predicates can perform arbitrary checks from ranging from useful to silly: .. code-block:: python

Only consider a path valid if it has at least one middlebox def has_middlebox_predicate(path, topology):

return any([topology.has_mbox(n) for n in path.nodes()])

Only consider a path valid if it has an IDS def has_ids_predicate(path, topology):

return any(['ids' in topology.get_servies(n) for n in path.nodes()])

Flip a coin to decide if a path is valid def coin_flip_predicate(path, topology):

return random.random > 0.5

Only allow paths where the number of links (hops) is even def only_even_hops_predicate(path, topology):

return len(path.links()) % 2 == 0

Path selection

4.2.4 Applications & Optimizations

Supported constraints and objectives

Cost functions

Composition of multiple applications

Advanced functionality

4.3 SOL API

This describes APIs for manipulating different types of objects that SOL implements and uses, such as `sol.Topology`, `sol.TrafficClass`, etc.

4.3.1 Topology

4.3.2 TrafficClass

4.3.3 Application

4.3.4 Path objects

4.3.5 Path generation and selection

4.3.6 Optimization

4.3.7 Utils & Logging

4.3.8 Topology generators

4.3.9 Topology Provisioning

4.3.10 Exceptions

4.4 Common Questions

4.4.1 Do I have to use Gurobi? Why not optimizer *X*?

Yes. It is an actively maintained, cross-platfom, well-performing convex optimization solver. Additionally, academics can get a free license. No plans to support other solvers soon.

4.4.2 I had a network with size *X* and the code is too slow.

This is likely for one of two reasons:

- You are unintentionally performing path generation (or selection) more than once. Path generation is a slow process, and results of path generation/selection should be saved for any subsequent optimizations.
- Your data is far too granular. If you are using a topology that includes hosts or have very fine-grained traffic classes, the problem size grows needlessly. Consider consolidating traffic classes and double check topology sizes.

4.4.3 Will you add feature X?

Maybe. Depends on the feature, availability of the maintainers and the overall research direction of the project. Feel free to open an issue on Github.

4.4.4 Is SOL thread-safe/concurrency-safe?

No. No attempt has been made to make it thread-safe.

4.5 Info for developers