graphkit Documentation

Release 1.0

Yahoo Vision and Machine Learning Team: Huy Nguyen, Arel Cor

Mar 05, 2020

Contents

It's a DAG all the way down!

CHAPTER 1

Lightweight computation graphs for Python

GraphKit is a lightweight Python module for creating and running ordered graphs of computations, where the nodes of the graph correspond to computational operations, and the edges correspond to output –> input dependencies between those operations. Such graphs are useful in computer vision, machine learning, and many other domains.

1.1 Operations

At a high level, an operation is a node in a computation graph. GraphKit uses an `operation` class to represent these computations.

1.1.1 The `operation` class

The `operation` class specifies an operation in a computation graph, including its input data dependencies as well as the output data it provides. It provides a lightweight wrapper around an arbitrary function to make these specifications.

There are many ways to instantiate an `operation`, and we'll get into more detail on these later. First off, though, here's the specification for the `operation` class:

1.1.2 Operations are just functions

At the heart of each `operation` is just a function, any arbitrary function. Indeed, you can instantiate an `operation` with a function and then call it just like the original function, e.g.:

```
>>> from operator import add
>>> from graphkit import operation

>>> add_op = operation(name='add_op', needs=['a', 'b'], provides=['a_plus_b'])(add)

>>> add_op(3, 4) == add(3, 4)
True
```

1.1.3 Specifying graph structure: `provides` and `needs`

Of course, each `operation` is more than just a function. It is a node in a computation graph, depending on other nodes in the graph for input data and supplying output data that may be used by other nodes in the graph (or as a graph output). This graph structure is specified via the `provides` and `needs` arguments to the `operation` constructor. Specifically:

- `provides`: this argument names the outputs (i.e. the returned values) of a given `operation`. If multiple outputs are specified by `provides`, then the return value of the function comprising the `operation` must return an iterable.
- `needs`: this argument names data that is needed as input by a given `operation`. Each piece of data named in needs may either be provided by another `operation` in the same graph (i.e. specified in the `provides` argument of that `operation`), or it may be specified as a named input to a graph computation (more on graph computations *here*).

When many operations are composed into a computation graph (see *Graph Composition* for more on that), Graphkit matches up the values in their `needs` and `provides` to form the edges of that graph.

Let's look again at the operations from the script in *Quick start*, for example:

```
>>> from operator import mul, sub
>>> from graphkit import compose, operation

>>> # Computes |a|^p.
>>> def abspow(a, p):
...   c = abs(a) ** p
...   return c

>>> # Compose the mul, sub, and abspow operations into a computation graph.
>>> graphop = compose(name="graphop")(
...    operation(name="mul1", needs=["a", "b"], provides=["ab"])(mul),
...    operation(name="sub1", needs=["a", "ab"], provides=["a_minus_ab"])(sub),
...    operation(name="abspow1", needs=["a_minus_ab"], provides=["abs_a_minus_ab_cubed
↪"], params={"p": 3})(abspow)
... )
```

The `needs` and `provides` arguments to the operations in this script define a computation graph that looks like this (where the oval are operations, squares/houses are data):

1.1.4 Constant operation parameters: `params`

Sometimes an `operation` will have a customizable parameter you want to hold constant across all runs of a computation graph. Usually, this will be a keyword argument of the underlying function. The `params` argument to the `operation` constructor provides a mechanism for setting such parameters.

`params` should be a dictionary whose keys correspond to keyword parameter names from the function underlying an `operation` and whose values are passed as constant arguments to those keyword parameters in all computations utilizing the `operation`.

1.1.5 Instantiating operations

There are several ways to instantiate an `operation`, each of which might be more suitable for different scenarios.

Decorator specification

If you are defining your computation graph and the functions that comprise it all in the same script, the decorator specification of `operation` instances might be particularly useful, as it allows you to assign computation graph structure to functions as they are defined. Here's an example:

```
>>> from graphkit import operation, compose

>>> @operation(name='foo_op', needs=['a', 'b', 'c'], provides='foo')
... def foo(a, b, c):
...   return c * (a + b)

>>> graphop = compose(name='foo_graph')(foo)
```

Functional specification

If the functions underlying your computation graph operations are defined elsewhere than the script in which your graph itself is defined (e.g. they are defined in another module, or they are system functions), you can use the functional specification of `operation` instances:

```
>>> from operator import add, mul
>>> from graphkit import operation, compose

>>> add_op = operation(name='add_op', needs=['a', 'b'], provides='sum')(add)
>>> mul_op = operation(name='mul_op', needs=['c', 'sum'], provides='product')(mul)

>>> graphop = compose(name='add_mul_graph')(add_op, mul_op)
```

The functional specification is also useful if you want to create multiple `operation` instances from the same function, perhaps with different parameter values, e.g.:

```
>>> from graphkit import operation, compose

>>> def mypow(a, p=2):
...    return a ** p

>>> pow_op1 = operation(name='pow_op1', needs=['a'], provides='a_squared')(mypow)
>>> pow_op2 = operation(name='pow_op2', needs=['a'], params={'p': 3}, provides='a_
↪cubed')(mypow)

>>> graphop = compose(name='two_pows_graph')(pow_op1, pow_op2)
```

A slightly different approach can be used here to accomplish the same effect by creating an operation "factory":

```
from graphkit import operation, compose

def mypow(a, p=2):
   return a ** p

pow_op_factory = operation(mypow)

pow_op1 = pow_op_factory(name='pow_op1', needs=['a'], provides='a_squared')
pow_op2 = pow_op_factory(name='pow_op2', needs=['a'], params={'p': 3}, provides='a_
↪cubed')

graphop = compose(name='two_pows_graph')(pow_op1, pow_op2)
```

1.1.6 Modifiers on `operation` inputs and outputs

Certain modifiers are available to apply to input or output values in `needs` and `provides`, for example to designate an optional input. These modifiers are available in the `graphkit.modifiers` module:

1.2 Graph Composition

GraphKit's `compose` class handles the work of tying together `operation` instances into a runnable computation graph.

1.2.1 The `compose` class

For now, here's the specification of `compose`. We'll get into how to use it in a second.

1.2.2 Simple composition of operations

The simplest use case for `compose` is assembling a collection of individual operations into a runnable computation graph. The example script from *Quick start* illustrates this well:

```
>>> from operator import mul, sub
>>> from graphkit import compose, operation

>>> # Computes |a|^p.
>>> def abspow(a, p):
...     c = abs(a) ** p
...     return c

>>> # Compose the mul, sub, and abspow operations into a computation graph.
>>> graphop = compose(name="graphop")(
...     operation(name="mul1", needs=["a", "b"], provides=["ab"])(mul),
...     operation(name="sub1", needs=["a", "ab"], provides=["a_minus_ab"])(sub),
...     operation(name="abspow1", needs=["a_minus_ab"], provides=["abs_a_minus_ab_cubed
↪"], params={"p": 3})(abspow)
... )
```

The call here to `compose()` yields a runnable computation graph that looks like this (where the circles are operations, squares are data, and octagons are parameters):

1.2.3 Running a computation graph

The graph composed in the example above in *Simple composition of operations* can be run by simply calling it with a dictionary argument whose keys correspond to the names of inputs to the graph and whose values are the corresponding input values. For example, if `graph` is as defined above, we can run it like this:

```
# Run the graph and request all of the outputs.
>>> out = graphop({'a': 2, 'b': 5})
>>> out
{'a': 2, 'b': 5, 'ab': 10, 'a_minus_ab': -8, 'abs_a_minus_ab_cubed': 512}
```

Producing a subset of outputs

By default, calling a graph-operation on a set of inputs will yield all of that graph's outputs. You can use the `outputs` parameter to request only a subset. For example, if `graphop` is as above:

```
# Run the graph-operation and request a subset of the outputs.
>>> out = graphop({'a': 2, 'b': 5}, outputs=["a_minus_ab"])
>>> out
{'a_minus_ab': -8}
```

When using `outputs` to request only a subset of a graph's outputs, GraphKit executes only the `operation` nodes in the graph that are on a path from the inputs to the requested outputs. For example, the `abspow1` operation will not be executed here.

Short-circuiting a graph computation

You can short-circuit a graph computation, making certain inputs unnecessary, by providing a value in the graph that is further downstream in the graph than those inputs. For example, in the graph-operation we've been working with, you could provide the value of `a_minus_ab` to make the inputs `a` and `b` unnecessary:

```
# Run the graph-operation and request a subset of the outputs.
>>> out = graphop({'a_minus_ab': -8})
>>> out
{'a_minus_ab': -8, 'abs_a_minus_ab_cubed': 512}
```

When you do this, any `operation` nodes that are not on a path from the downstream input to the requested outputs (i.e. predecessors of the downstream input) are not computed. For example, the `mul1` and `sub1` operations are not executed here.

This can be useful if you have a graph-operation that accepts alternative forms of the same input. For example, if your graph-operation requires a `PIL.Image` as input, you could allow your graph to be run in an API server by adding an earlier `operation` that accepts as input a string of raw image data and converts that data into the needed `PIL.Image`. Then, you can either provide the raw image data string as input, or you can provide the `PIL.Image` if you have it and skip providing the image data string.

1.2.4 Adding on to an existing computation graph

Sometimes you will have an existing computation graph to which you want to add operations. This is simple, since `compose` can compose whole graphs along with individual `operation` instances. For example, if we have `graph` as above, we can add another operation to it to create a new graph:

```
>>> # Add another subtraction operation to the graph.
>>> bigger_graph = compose(name="bigger_graph")(
...     graphop,
...     operation(name="sub2", needs=["a_minus_ab", "c"], provides="a_minus_ab_minus_c
↪")(sub)
... )

>>> # Run the graph and print the output.
>>> sol = bigger_graph({'a': 2, 'b': 5, 'c': 5}, outputs=["a_minus_ab_minus_c"])
>>> sol
{'a_minus_ab_minus_c': -13}
```

This yields a graph which looks like this (see *Plotting*):

1.2.5 More complicated composition: merging computation graphs

Sometimes you will have two computation graphs—perhaps ones that share operations—you want to combine into one. In the simple case, where the graphs don't share operations or where you don't care whether a duplicated operation is run multiple (redundant) times, you can just do something like this:

```
combined_graph = compose(name="combined_graph")(graph1, graph2)
```

However, if you want to combine graphs that share operations and don't want to pay the price of running redundant computations, you can set the `merge` parameter of `compose()` to `True`. This will consolidate redundant `operation` nodes (based on `name`) into a single node. For example, let's say we have `graphop`, as in the examples above, along with this graph:

```
>>> # This graph shares the "mul1" operation with graph.
>>> another_graph = compose(name="another_graph")(
...     operation(name="mul1", needs=["a", "b"], provides=["ab"])(mul),
...     operation(name="mul2", needs=["c", "ab"], provides=["cab"])(mul)
... )
```

We can merge `graphop` and `another_graph` like so, avoiding a redundant `mul1` operation:

```
>>> merged_graph = compose(name="merged_graph", merge=True)(graphop, another_graph)
>>> print(merged_graph)
NetworkOperation(name='merged_graph',
                 needs=[optional('a'), optional('b'), optional('c')],
                 provides=['ab', 'a_minus_ab', 'abs_a_minus_ab_cubed', 'cab'])
```

This `merged_graph` will look like this:

As always, we can run computations with this graph by simply calling it:

```
>>> merged_graph({'a': 2, 'b': 5, 'c': 5}, outputs=["cab"])
{'cab': 50}
```

1.2.6 Errors

If an operation fails, its exception gets annotated with the folllowing properties as a debug aid:

```
>>> def scream(*args):
...     raise ValueError("Wrong!")
```

```
>>> try:
...     compose("errgraph")(
...         operation(name="screamer", needs=['a'], provides=["foo"])(scream)
...     )({'a': None})
... except ValueError as ex:
...     print(ex.execution_node)
...     print(ex.execution_plan)
FunctionalOperation(name='screamer', needs=['a'], provides=['foo'])
ExecutionPlan(inputs=('a',), outputs=(), steps:
  +--FunctionalOperation(name='screamer', needs=['a'], provides=['foo']))
```

Of course from the `ExecutionPlan` you can explore its `dag` property or the `net` that compiled it.

1.2.7 Execution internals

1.3 API Reference

1.3.1 Module: *base*

1.3.2 Module: *functional*

1.3.3 Module: *network*

1.3.4 Module: *plot*

1.4 Changelog

1.4.1 v1.3.0 (Oct 2019): New DAG solver, better plotting & "sideffect"

Kept external API (hopefully) the same, but revamped pruning algorithm and refactored network compute/compile structure, so results may change; significantly enhanced plotting. The only new feature actually is the `sideffect`` modifier.

Network:

- FIX(#18, #26, #29, #17, #20): Revamped DAG SOLVER to fix bad pruning described in #24 & #25

  Pruning now works by breaking incoming provide-links to any given intermedediate inputs dropping operations with partial inputs or without outputs.

  The end result is that operations in the graph that do not have all inputs satisfied, they are skipped (in v1.2.4 they crashed).

  Also started annotating edges with optional/sideffects, to make proper use of the underlying `networkx` graph.

- REFACT(#21, #29): Refactored Network and introduced `ExecutionPlan` to keep compilation results (the old `steps` list, plus input/output names).

  Moved also the check for when to evict a value, from running the execution-plan, to whenbuilding it; thus, execute methods don't need outputs anymore.

- ENH(#26): "Pin* input values that may be overriten by calculated ones.

  This required the introduction of the new `PinInstruction` in the execution plan.

- FIX(#23, #22-2.4.3): Keep consistent order of `networkx.DiGraph` and *sets*, to generate deterministic solutions.

  Unfortunately, it non-determinism has not been fixed in < PY3.5, just reduced the frequency of spurious failures, caused by unstable dicts, and the use of subgraphs.

- enh: Mark outputs produced by `NetworkOperation`'s needs as `optional`. TODO: subgraph network-operations would not be fully functional until *"optional outpus"* are dealt with (see #22-2.5).

- enh: Annotate operation exceptions with `ExecutionPlan` to aid debug sessions,

- drop: methods `list_layers()`/`show layers()` not needed, `repr()` is a better replacement.

Plotting:

- ENH(#13, #26, #29): Now network remembers last plan and uses that to overlay graphs with the internals of the planing and execution:
  - execution-steps & order
  - delete & pin instructions
  - given inputs & asked outputs
  - solution values (just if they are present)
  - "optional" needs & broken links during pruning
- REFACT: Move all API doc on plotting in a single module, splitted in 2 phases, build DOT & render DOT
- FIX(#13): bring plot writing into files up-to-date from PY2; do not create plot-file if given file-extension is not supported.
- FEAT: path pydot library to support rendering in *Jupyter notebooks*.

Testing & other code:

- Increased coverage from 77% –> 90%.
- ENH(#28): use `pytest`, to facilitate TCs parametrization.
- ENH(#30): Doctest all code; enabled many assertions that were just print-outs in v1.2.4.
- FIX: `operation.__repr__()` was crashing when not all arguments had been set - a condition frequtnly met during debugging session or failed TCs (inspired by @syamajala's 309338340).
- enh: Sped up parallel/multihtread TCs by reducing delays & repetitions.

> **Tip:** You need `pytest -m slow` to run those slow tests.

Chore & Docs:

- FEAT: add changelog in `CHANGES.rst` file, containing flowcharts to compare versions `v1.2.4 <--> v1.3..0`.
- enh: updated site & documentation for all new features, comparing with v1.2.4.
- enh(#30): added "API reference' chapter.
- drop(build): `sphinx_rtd_theme` library is the default theme for Sphinx now.
- enh(build): Add `test` *pip extras*.
- sound: https://www.youtube.com/watch?v=-527VazA4IQ, https://www.youtube.com/watch?v=8J182LRi8sU&t=43s

1.4.2 v1.2.4 (Mar 7, 2018)

- Issues in pruning algorithm: #24, #25
- Blocking bug in plotting code for Python-3.x.
- Test-cases without assertions (just prints).

1.4.3 1.2.2 (Mar 7, 2018, @huyng): Fixed versioning

Versioning now is manually specified to avoid bug where the version was not being correctly reflected on pip install deployments

1.4.4 1.2.1 (Feb 23, 2018, @huyng): Fixed multi-threading bug and faster compute through caching of *find_necessary_steps*

We've introduced a cache to avoid computing find_necessary_steps multiple times during each inference call.

This has 2 benefits:

- It reduces computation time of the compute call
- It avoids a subtle multi-threading bug in networkx when accessing the graph from a high number of threads.

1.4.5 1.2.0 (Feb 13, 2018, @huyng)

Added *set_execution_method('parallel')* for execution of graphs in parallel.

1.4.6 1.1.0 (Nov 9, 2017, @huyng)

Update setup.py

1.4.7 1.0.4 (Nov 3, 2017, @huyng): Networkx 2.0 compatibility

Minor Bug Fixes:

- Compatibility fix for networkx 2.0
- *net.times* now only stores timing info from the most recent run

1.4.8 1.0.3 (Jan 31, 2017, @huyng): Make plotting dependencies optional

- Merge pull request #6 from yahoo/plot-optional
- make plotting dependencies optional

1.4.9 1.0.2 (Sep 29, 2016, @pumpikano): Merge pull request #5 from yahoo/remove-packaging-dep

- Remove 'packaging' as dependency

1.4.10 1.0.1 (Aug 24, 2016)

1.4.11 1.0 (Aug 2, 2016, @robwhess)

First public release in PyPi & GitHub.

- Merge pull request #3 from robwhess/travis-build
- Travis build

CHAPTER 2

Quick start

Here's how to install:

```
pip install graphkit
```

OR with dependencies for plotting support (and you need to install Graphviz program separately with your OS tools):

```
pip install graphkit[plot]
```

Here's a Python script with an example GraphKit computation graph that produces multiple outputs (`a * b`, `a - a * b`, and `abs(a - a * b) ** 3`):

```
from operator import mul, sub
from graphkit import compose, operation

# Computes |a|^p.
def abspow(a, p):
    c = abs(a) ** p
    return c

# Compose the mul, sub, and abspow operations into a computation graph.
graphop = compose(name="graphop")(
    operation(name="mul1", needs=["a", "b"], provides=["ab"])(mul),
    operation(name="sub1", needs=["a", "ab"], provides=["a_minus_ab"])(sub),
    operation(name="abspow1", needs=["a_minus_ab"], provides=["abs_a_minus_ab_cubed"],␣
↪params={"p": 3})(abspow)
)

# Run the graph-operation and request all of the outputs.
out = graphop({'a': 2, 'b': 5})

# Prints "{'a': 2, 'a_minus_ab': -8, 'b': 5, 'ab': 10, 'abs_a_minus_ab_cubed': 512}".
print(out)

# Run the graph-operation and request a subset of the outputs.
```

(continues on next page)

(continued from previous page)

```
out = graphop({'a': 2, 'b': 5}, outputs=["a_minus_ab"])

# Prints "{'a_minus_ab': -8}".
print(out)
```

As you can see, any function can be used as an operation in GraphKit, even ones imported from system modules!

CHAPTER 3

Plotting

For debugging the above graph-operation you may plot the *execution plan* of the last computation it using these methods:

```
graphop.plot(show=True)                 # open a matplotlib window
graphop.plot("intro.svg")               # other supported formats: png, jpg, pdf, ...
graphop.plot()                          # without arguments return a pydot.DOT object
graphop.plot(solution=out)              # annotate graph with solution values
```

Fig. 1: The legend for all graphkit diagrams, generated by `graphkit.plot.legend()`.

Tip: The pydot.Dot instances returned by `plot()` are rendered directly in *Jupyter/IPython* notebooks as SVG images.

Note: For plots, Graphviz program must be in your PATH, and `pydot` & `matplotlib` python packages installed. You may install both when installing `graphkit` with its `plot` extras:

```
pip install graphkit[plot]
```

CHAPTER 4

License

Code licensed under the Apache License, Version 2.0 license. See LICENSE file for terms.