pldl Documentation

Release latest

Wai-Shing Luk

Apr 25, 2024

CONTENTS

1 Note **3**

2 Contents **5**
- 2.1 Contributing . . . 5
- 2.2 License . . . 9
- 2.3 Contributors . . . 9
- 2.4 Changelog . . . 9
- 2.5 pldl . . . 10

3 Indices and tables **17**

Python Module Index **19**

Index **21**

Add a short description here!

CHAPTER

ONE

NOTE

This is the main page of your project's Sphinx documentation. It is formatted in Markdown. Add additional pages by creating md-files in `docs` or rst-files (formatted in reStructuredText) and adding links to them in the `Contents` section below.

Please check Sphinx and MyST for more information about how to document your project and how to configure your preferences.

CHAPTER

TWO

CONTENTS

2.1 Contributing

Welcome to `primal-dual-approx-py` contributor's guide.

This document focuses on getting any potential contributor familiarized with the development processes, but other kinds of contributions are also appreciated.

If you are new to using git or have never collaborated in a project previously, please have a look at contribution-guide.org. Other resources are also listed in the excellent guide created by FreeCodeCamp[1].

Please notice, all users and contributors are expected to be **open, considerate, reasonable, and respectful**. When in doubt, Python Software Foundation's Code of Conduct is a good reference in terms of behavior guidelines.

2.1.1 Issue Reports

If you experience bugs or general issues with `primal-dual-approx-py`, please have a look on the issue tracker. If you don't see anything useful there, please feel free to fire an issue report.

Tip: Please don't forget to include the closed issues in your search. Sometimes a solution was already reported, and the problem is considered **solved**.

New issue reports should include information about your programming environment (e.g., operating system, Python version) and steps to reproduce the problem. Please try also to simplify the reproduction steps to a very minimal example that still illustrates the problem you are facing. By removing other factors, you help us to identify the root cause of the issue.

2.1.2 Documentation Improvements

You can help improve `primal-dual-approx-py` docs by making them more readable and coherent, or by adding missing information and correcting mistakes.

`primal-dual-approx-py` documentation uses Sphinx as its main documentation compiler. This means that the docs are kept in the same repository as the project code, and that any documentation update is done in the same way was a code contribution.

When working on documentation changes in your local machine, you can compile them using tox :

[1] Even though, these resources focus on open source projects and communities, the general ideas behind collaborating with other developers to collectively create software are general and can be applied to all sorts of environments, including private companies and proprietary code bases.

```
tox -e docs
```

and use Python's built-in web server for a preview in your web browser (`http://localhost:8000`):

```
python3 -m http.server --directory 'docs/_build/html'
```

2.1.3 Code Contributions

Submit an issue

Before you work on any non-trivial code contribution it's best to first create a report in the issue tracker to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

Create an environment

Before you start coding, we recommend creating an isolated virtual environment to avoid any problems with your installed Python packages. This can easily be done via either virtualenv:

```
virtualenv <PATH TO VENV>
source <PATH TO VENV>/bin/activate
```

or Miniconda:

```
conda create -n primal-dual-approx-py python=3 six virtualenv pytest pytest-cov
conda activate primal-dual-approx-py
```

Clone the repository

1. Create an user account on GitHub if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on GitHub.
3. Clone this copy to your local disk:

   ```
   git clone git@github.com:YourLogin/primal-dual-approx-py.git
   cd primal-dual-approx-py
   ```

4. You should run:

   ```
   pip install -U pip setuptools -e .
   ```

   to be able to import the package under development in the Python REPL.

5. Install pre-commit:

   ```
   pip install pre-commit
   pre-commit install
   ```

   `primal-dual-approx-py` comes with a lot of hooks configured to automatically help the developer to check the code being written.

Implement your changes

1. Create a branch to hold your changes:

   ```
   git checkout -b my-feature
   ```

   and start making changes. Never work on the main branch!

2. Start your work on this branch. Don't forget to add docstrings to new functions, modules and classes, especially if they are part of public APIs.

3. Add yourself to the list of contributors in `AUTHORS.rst`.

4. When you're done editing, do:

   ```
   git add <MODIFIED FILES>
   git commit
   ```

   to record your changes in git.

   Please make sure to see the validation messages from pre-commit and fix any eventual issues. This should automatically use flake8/black to check/fix the code style in a way that is compatible with the project.

   Important: Don't forget to add unit tests and documentation in case your contribution adds an additional feature and is not just a bugfix.

   Moreover, writing a descriptive commit message is highly recommended. In case of doubt, you can check the commit history with:

   ```
   git log --graph --decorate --pretty=oneline --abbrev-commit --all
   ```

   to look for recurring communication patterns.

   ---

5. Please check that your changes don't break any unit tests with:

   ```
   tox
   ```

   (after having installed tox with `pip install tox` or `pipx`).

   You can also use tox to run several other pre-configured tasks in the repository. Try `tox -av` to see a list of the available checks.

Submit your contribution

1. If everything works fine, push your local branch to the remote server with:

   ```
   git push -u origin my-feature
   ```

2. Go to the web page of your fork and click "Create pull request" to send your changes for review.

Troubleshooting

The following tips can be used when facing problems to build or test the package:

1. Make sure to fetch all the tags from the upstream repository. The command `git describe --abbrev=0 --tags` should return the version you are expecting. If you are trying to run CI scripts in a fork repository, make sure to push all the tags. You can also try to remove all the egg files or the complete egg folder, i.e., `.eggs`, as well as the `*.egg-info` folders in the `src` folder or potentially in the root of your project.

2. Sometimes tox misses out when new dependencies are added, especially to `setup.cfg` and `docs/requirements.txt`. If you find any problems with missing dependencies when running a command with tox, try to recreate the `tox` environment using the `-r` flag. For example, instead of:

   ```
   tox -e docs
   ```

   Try running:

   ```
   tox -r -e docs
   ```

3. Make sure to have a reliable tox installation that uses the correct Python version (e.g., 3.7+). When in doubt you can run:

   ```
   tox --version
   # OR
   which tox
   ```

   If you have trouble and are seeing weird errors upon running tox, you can also try to create a dedicated virtual environment with a tox binary freshly installed. For example:

   ```
   virtualenv .venv
   source .venv/bin/activate
   .venv/bin/pip install tox
   .venv/bin/tox -e all
   ```

4. Pytest can drop you in an interactive session in the case an error occurs. In order to do that you need to pass a `--pdb` option (for example by running `tox -- -k <NAME OF THE FALLING TEST> --pdb`). You can also setup breakpoints manually instead of using the `--pdb` option.

2.1.4 Maintainer tasks

Releases

If you are part of the group of maintainers and have correct user permissions on PyPI, the following steps can be used to release a new version for `primal-dual-approx-py`:

1. Make sure all unit tests are successful.
2. Tag the current commit on the main branch with a release tag, e.g., `v1.2.3`.
3. Push the new tag to the upstream repository, e.g., `git push upstream v1.2.3`
4. Clean up the `dist` and `build` folders with `tox -e clean` (or `rm -rf dist build`) to avoid confusion with old builds and Sphinx docs.
5. Run `tox -e build` and check that the files in `dist` have the correct version (no `.dirty` or git hash) according to the git tag. Also check the sizes of the distributions, if they are too big (e.g., > 500KB), unwanted clutter may have been accidentally included.

6. Run `tox -e publish -- --repository pypi` and check that everything was uploaded to PyPI correctly.

2.2 License

2.3 Contributors

- Wai-Shing Luk <luk036@gmail.com>

2.4 Changelog

2.4.1 Version 0.1

- Feature A added
- FIX: nasty bug #1729 fixed
- add your changes here!

2.5 pldl

2.5.1 pldl package

Submodules

pldl.cover module

pldl.cover.**min_cycle_cover**(*ugraph: Graph*, *weight: MutableMapping*, *coverset: Set | None = None*) → Tuple[Set, int | float]

The *min_cycle_cover* function performs minimum cycle cover using a primal-dual approximation algorithm (without post-processing).

Parameters

- **ugraph** (*nx.Graph*) – The *ugraph* parameter is a *nx.Graph* object representing the input graph. It contains the nodes and edges of the graph
- **weight** (*MutableMapping*) – The *weight* parameter is a dictionary that assigns a weight to each node in the graph. The weights are used to determine the minimum cycle cover
- **coverset** (*Optional[Set]*) – The *coverset* parameter is an optional set that contains the nodes that are already covered by previous cycles. It is used to keep track of the nodes that have already been included in the minimum cycle cover. If no *coverset* is provided, it is initialized as an empty set

Returns

The function *min_cycle_cover* returns a tuple containing a set and either an integer or a float. The set represents the minimum cycle cover, and the integer or float represents the weight of the minimum cycle cover.

Examples

```
>>> ugraph = nx.Graph()
>>> ugraph.add_edges_from([(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])
>>> weight = {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
>>> soln = set()
>>> min_cycle_cover(ugraph, weight, soln)
({0, 1, 2}, 3)
```

pldl.cover.**min_hyper_vertex_cover**(*hyprgraph*, *weight: MutableMapping*, *coverset: Set | None = None*) → Tuple[Set, int | float]

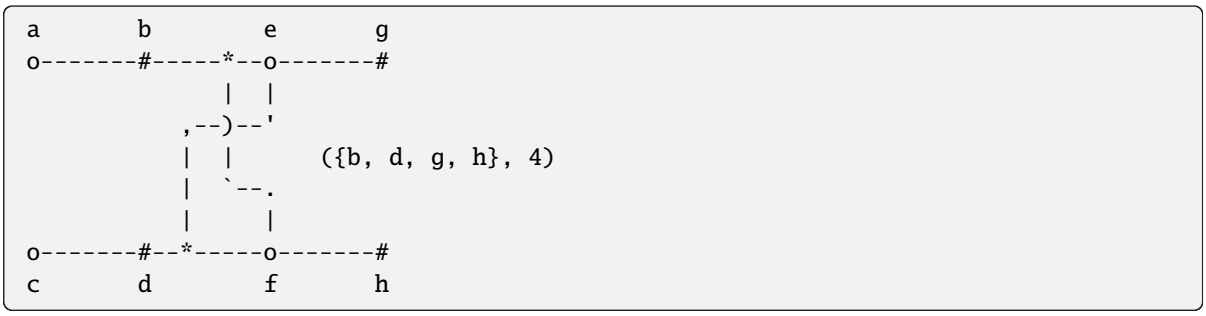
The *min_hyper_vertex_cover* function performs minimum weighted vertex cover using a primal-dual approximation algorithm (without post-processing).

Parameters

- **hyprgraph** – The *hyprgraph* parameter represents a hypergraph, which is a generalization of a graph where an edge can connect more than two vertices. It is likely represented as a data structure that contains information about the vertices and edges of the hypergraph
- **weight** (*MutableMapping*) – The *weight* parameter is a mutable mapping that assigns a weight to each vertex in the hypergraph. It is used to determine the minimum weighted vertex cover
- **coverset** (*Optional[Set]*) – The *coverset* parameter is an optional set that represents the current vertex cover. It contains the vertices that have been selected as part of the cover. If no *coverset* is provided, it defaults to an empty set

Returns

The function *min_hyper_vertex_cover* returns a tuple containing two elements. The first element is a set representing the minimum weighted vertex cover, and the second element is either an integer or a float representing the weight of the vertex cover.



pldl.cover.**min_odd_cycle_cover**(*ugraph: Graph*, *weight: MutableMapping*, *coverset: Set | None = None*) → Tuple[Set, int | float]

The *min_odd_cycle_cover* function performs minimum odd cycle cover using a primal-dual approximation algorithm (without post-processing).

Parameters

- **ugraph** (*nx.Graph*) – The *ugraph* parameter is a *nx.Graph* object representing the input graph. It is used to define the graph structure and find cycles in the graph
- **weight** (*MutableMapping*) – The *weight* parameter is a dictionary that assigns a weight to each node in the graph
- **coverset** (*Optional[Set]*) – The *coverset* parameter is an optional set that represents the initial set of vertices that are covered by the minimum odd cycle cover. This set can be empty if no vertices are initially covered

Returns

The function *min_odd_cycle_cover* returns a tuple containing a set and either an integer or a float. The set represents the minimum odd cycle cover, and the integer or float represents the weight of the cover.

Examples

```
>>> ugraph = nx.Graph()
>>> ugraph.add_edges_from([(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])
>>> weight = {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
>>> soln = set()
>>> min_odd_cycle_cover(ugraph, weight, soln)
({0, 1, 2}, 3)
```

pldl.cover.**min_vertex_cover**(*ugraph: Graph*, *weight: MutableMapping*, *coverset: Set | None = None*) → Tuple[Set, int | float]

The *min_vertex_cover* function performs minimum weighted vertex cover using a primal-dual approximation algorithm (without post-processing).

Parameters

- **ugraph** (*nx.Graph*) – The parameter *ugraph* is a *nx.Graph* object, which represents the input graph. It is an undirected graph where each edge represents a connection between two vertices
- **weight** (*MutableMapping*) – The *weight* parameter is a dictionary that assigns a weight to each vertex in the graph. The weights are used to determine the minimum weighted vertex cover
- **coverset** (*Optional[Set]*) – The *coverset* parameter is an optional set that represents the current vertex cover solution. It is used to keep track of the vertices that are included in the cover. If no *coverset* is provided, an empty set is used as the initial cover

Returns

The function *min_vertex_cover* returns a tuple containing two elements. The first element is a set representing the minimum weighted vertex cover, and the second element is either an integer or a float representing the weight of the minimum vertex cover.

Examples

```
>>> ugraph = nx.Graph()
>>> ugraph.add_edges_from([(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])
>>> weight = {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
>>> soln = set()
>>> min_vertex_cover(ugraph, weight, soln)
({0, 1, 2, 3}, 4)
```

pldl.cover.**pd_cover**(*violate: Callable*, *weight: MutableMapping*, *soln: Set*) → Tuple[Set, int | float]

The function *pd_cover* implements a primal-dual approximation algorithm for covering problems.

Parameters

- **violate** (*Callable*) – The *violate* parameter is a callable function or oracle that returns a set of violate elements. It is used to generate sets of elements that violate the current solution. Each set represents a potential improvement to the solution
- **weight** (*MutableMapping*) – The *weight* parameter is a dictionary that represents the weight of each element. The keys of the dictionary are the elements, and the values are their corresponding weights
- **soln** (*Set*) – The *soln* parameter is a set that represents the current solution set. It initially contains no elements, and elements are added to it during the algorithm

Returns

a tuple containing the updated solution set and the total primal cost.

Examples

```
>>> def violate_graph() -> Generator:
...     yield [0, 1]
...     yield [0, 2]
...     yield [1, 2]
>>> weight = {0: 1, 1: 2, 2: 3}
>>> soln = set()
>>> pd_cover(violate_graph, weight, soln)
({0, 1}, 4)
```

pldl.graph_algo module

Minimum vertex cover for weighed graphs. 1. Support Lazy evalution

pldl.graph_algo.**min_maximal_independant_set**(*ugraph*, *weight: MutableMapping*, *indset: Set | None = None*, *dep: Set | None = None*) → Tuple[Set, int | float]

The *min_maximal_independant_set* function performs minimum weighted maximal independent set using primal-dual algorithm.

Parameters

- **ugraph** – ugraph is an undirected graph represented using the NetworkX library. It represents the graph structure and contains the vertices and edges of the graph
- **weight** (*MutableMapping*) – The *weight* parameter is a dictionary-like object that assigns a weight to each vertex in the graph. The keys of the dictionary represent the vertices, and the values represent their corresponding weights
- **indset** (*Optional[Set]*) – The *indset* parameter is a set that represents the current independent set. It is initially set to *None* and is updated during the execution of the *min_maximal_independent_set* function
- **dep** (*Optional[Set]*) – The *dep* parameter is a set that represents the dependent vertices in the graph. These are the vertices that are not included in the independent set and are adjacent to vertices in the independent set. The *coverset* function is used to add a vertex and its adjacent vertices to the dependent set

Returns

The function *min_maximal_independant_set* returns a tuple containing the minimum weighted maximal independent set (indset) and the total primal cost (total_prml_cost).

Examples

```
>>> import networkx as nx
>>> from pldl.graph_algo import min_maximal_independant_set
>>> ugraph = nx.Graph()
>>> ugraph.add_edges_from([(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])
>>> weight = {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
>>> indset = set()
>>> dep = set()
>>> min_maximal_independant_set(ugraph, weight, indset, dep)
({0, 3}, 2)
```

pldl.graph_algo.**min_vertex_cover_fast**(*ugraph*, *weight: MutableMapping*, *coverset: Set | None = None*) → Tuple[Set, int | float]

The *min_vertex_cover_fast* function performs minimum weighted vertex cover using a primal-dual approximation algorithm (without post-processing).

Parameters

- **ugraph** – ugraph is a NetworkX graph object representing the graph on which the minimum weighted vertex cover algorithm will be performed. It contains the nodes and edges of the graph
- **weight** (*MutableMapping*) – The *weight* parameter is a mutable mapping that represents the weight of each vertex in the graph. It is used to determine the minimum weighted vertex cover. The keys of the mapping are the vertices of the graph, and the values are the corresponding weights
- **coverset** (*Optional[Set]*) – The *coverset* parameter is an optional set that represents the current vertex cover. It is used to keep track of the vertices that are included in the cover. If no coverset is provided, a new empty set is created

Returns

The function *min_vertex_cover_fast* returns a tuple containing the vertex cover set and the total weight of the vertex cover.

Examples

```
>>> import networkx as nx
>>> from pldl.graph_algo import min_vertex_cover_fast
>>> ugraph = nx.Graph()
>>> ugraph.add_edges_from([(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])
>>> weight = {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
>>> coverset = set()
>>> min_vertex_cover_fast(ugraph, weight, coverset)
({0, 1, 2, 3}, 4)
```

pldl.netlist module

pldl.netlist_algo module

pldl.skeleton module

This is a skeleton file that can serve as a starting point for a Python console script. To run this script uncomment the following lines in the `[options.entry_points]` section in `setup.cfg`:

```
console_scripts =
     fibonacci = pldl.skeleton:run
```

Then run `pip install .` (or `pip install -e .` for editable mode) which will install the command `fibonacci` inside your current environment.

Besides console scripts, the header (i.e. until `_logger`...) of this file can also be used as template for Python modules.

Note: This skeleton file can be safely removed if not needed!

References

- https://setuptools.readthedocs.io/en/latest/userguide/entry_point.html
- https://pip.pypa.io/en/stable/reference/pip_install

pldl.skeleton.**fib**(*n*)

Fibonacci example function

Parameters
n (*int*) – integer

Returns
n-th Fibonacci number

Return type
int

pldl.skeleton.**main**(*args*)

Wrapper allowing *fib()* to be called with string arguments in a CLI fashion

Instead of returning the value from *fib()*, it prints the result to the `stdout` in a nicely formatted message.

Parameters
args (*List[str]*) – command line parameters as list of strings (for example ["--verbose", "42"]).

pldl.skeleton.**parse_args**(*args*)

Parse command line parameters

Parameters
args (*List[str]*) – command line parameters as list of strings (for example ["--help"]).

Returns
command line parameters namespace

Return type
argparse.Namespace

pldl.skeleton.**run**()

Calls *main()* passing the CLI arguments extracted from sys.argv

This function can be used as entry point to create console scripts with setuptools.

pldl.skeleton.**setup_logging**(*loglevel*)

Setup basic logging

Parameters
loglevel (*int*) – minimum loglevel for emitting messages

Module contents

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pldl, 16
pldl.cover, 10
pldl.graph_algo, 13
pldl.skeleton, 15

INDEX

F

fib() (*in module pldl.skeleton*), 15

M

main() (*in module pldl.skeleton*), 15
min_cycle_cover() (*in module pldl.cover*), 10
min_hyper_vertex_cover() (*in module pldl.cover*), 10
min_maximal_independant_set() (*in module pldl.graph_algo*), 13
min_odd_cycle_cover() (*in module pldl.cover*), 11
min_vertex_cover() (*in module pldl.cover*), 12
min_vertex_cover_fast() (*in module pldl.graph_algo*), 14
module
    pldl, 16
    pldl.cover, 10
    pldl.graph_algo, 13
    pldl.skeleton, 15

P

parse_args() (*in module pldl.skeleton*), 16
pd_cover() (*in module pldl.cover*), 12
pldl
    module, 16
pldl.cover
    module, 10
pldl.graph_algo
    module, 13
pldl.skeleton
    module, 15

R

run() (*in module pldl.skeleton*), 16

S

setup_logging() (*in module pldl.skeleton*), 16