cortexpy Documentation

Release 0.46.5

Warren Kretzschmar and Kiran Garimella

Mar 15, 2020

Contents

CHAPTER 1

Overview of Cortexpy

tests	
package	
docs	

Cortexpy is a Python package for sequence analysis using linked and colored De Bruijn graphs such as the ones created by Cortex and Mccortex. This project aims to mirror many of the features contained in CortexJDK.

Cortexpy also comes with a command-line tool for basic inspection and manipulation of Cortex graphs with and without links.

1.1 Audience

The audience of cortexpy is researchers working with colored De Bruijn graphs and link information in Cortex and Mccortex format.

1.2 Free software

Cortexpy is free software; you can redistribute it and/or modify it under the terms of the Apache License version 2.0. Contributions are welcome. Please join us on GitHub.

1.3 Installation

```
pip install cortexpy
```

1.4 Documentation

For more information, please see cortexpy documentation.

1.5 Citing cortexpy

If you use cortexpy in your work, please consider citing:

> Akhter, Shirin, Warren W. Kretzschmar, Veronika Nordal, Nicolas Delhomme, Nathaniel R. Street, Ove Nilsson, Olof Emanuelsson, and Jens F. Sundström. "Integrative analysis of three RNA sequencing methods identifies mutually exclusive exons of MADS-box isoforms during early bud development in *Picea abies*." *Frontiers in Plant Science* 9 (2018). https://doi.org/10.3389/fpls.2018.01625

1.6 Bugs

This code is maintained by Warren Kretzschmar <winni@warrenwk.com>. For bugs, please raise a GitHub issue.

1.7 Development

1. Install conda.
2. Download mccortex for testing:

   ```
   conda env create -f environment.yml -n my-dev-environment
   ```

3. Activate development environment:

   ```
   conda activate my-dev-environment
   ```

4. Install remaining development tools:

   ```
   pip3 install -r requirements.txt
   ```

All remaining commands in the development section need to be run in an activated conda dev environment.

1.7.1 Tests

```
make test
```

1.7.2 Deploy new cortexpy version to pypi

Requires access credentials for pypi.

```
make deploy
```

1.7.3 Building the docs

The documentation is automatically built by read-the-docs on push to master. To build the documentation manually:

```
# install sphinx dependencies
pip install -r docs/requirements.txt

make docs
```

CHAPTER 2

Tutorial

The cortexpy package consists of a python API and a command-line tool for working with Cortex graphs. Below, we start by looking at how to use the python API to perform an example workflow.

2.1 Using the python API to filter Cortex graphs

2.1.1 Building Cortex files

Let's start by by creating two Cortex files to work with. At present, cortexpy does not provide a way to easily create a Cortex file, so we will instead use Mccortex. Mccortex can be compiled from source or installed using bioconda.

Let's start by creating two FASTA files from which to create two Cortex files:

```
echo -e '>1\nAAAAA' > file1.fasta
echo -e '>1\nCCCCC' > file2.fasta
```

We now have two FASTA files each containing a single sequence. We can now build a Cortex graph from each file. We choose to use a kmer-size of 5:

```
mccortex 5 build --sort -k 5 --sample file1 -1 file1.fasta file1.ctx
mccortex 5 build --sort -k 5 --sample file2 -1 file2.fasta file2.ctx
```

We now have two cortex files: `file1.ctx` and `file2.ctx`. As the Cortex format represents colored De Bruijn graphs, we could have stored the information from the two FASTA files in a single graph as two separate colors. However, we are creating two files in order to demonstrate the cortexpy API later on.

We can check what kmers are stored in each graph using the **`cortexpy`** command-line tool:

```
> cortexpy view graph file1.ctx
AAAAA 1 ........

> cortexpy view graph file2.ctx
CCCCC 1 ........
```

This output tells us that each graph consists of a single kmer with coverage 1.

2.1.2 Inspecting Cortex graphs in Python

Cortexpy offers many ways to inspect Cortex files. Much of that functionality is available through the `RandomAccess` class. Let us start by loading a Cortex file inside python:

```
>>> from cortexpy.graph.parser.random_access import RandomAccess
>>> # make sure to open the cortex graph in binary mode
>>> ra = RandomAccess(open('file1.ctx', 'rb'))
```

We can now interrogate the `ra` object. Let's see what the header size of the Cortex file is:

```
>>> ra.header.kmer_size
5
```

Let's check if the kmer `AAAAA` exists in the graph and retrieve it:

```
>>> 'AAAAA' in ra
True
>>> ra['AAAAA']
Kmer(_kmer_data=KmerData(_data=b'\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00
↪', kmer_size=5, num_colors=1, _kmer='AAAAA', _coverage=None, _edges=None), num_
↪colors=1, kmer_size=5, _revcomp=None)
```

We can see that the returned kmer object contains information on the kmer size (5) and the number of colors stored in the kmer (1).

Now let's put it all together and search both graphs that we created while *Building Cortex files* for our kmer of interest, `AAAAA`:

Listing 1: search.py

```
from cortexpy.graph.parser.random_access import RandomAccess

for graph in ['file1.ctx', 'file2.ctx']:
    # make sure to open the cortex graph in binary mode
    with open(graph, 'rb') as fh:
        ra = RandomAccess(fh)

        # let's see if our favorite kmer is in the graph
        if 'AAAAA' in ra:
            print(f'AAAAA exists in {graph}!')
```

This is what we see if we run this code from the command line:

```
> python3 search.py
AAAAA exists in file1.ctx!
```

CHAPTER 3

On link-informed graph traversal

The all-simple-paths algorithm as implemented in `nx.all_simple_paths()` inside the `networkx` (abbreviated below as `nx`) package version 2.2 uses a depth-first traversal scheme to find all possible paths from a start node to one or more end nodes[1].

For example, let nodes `A-F` represent unitigs in a De Bruijn graph created from sequencing reads of transcripts:

An all-simple-paths traversal starting at `A` will return the paths `ABDE`, `ABDF`, `ACDE`, and `ACDF`. However, what if the sequenced reads that were used to create this graph only originated from two paths: `ABDE` and `ACDF`? Can some of these sequencing reads be used to restrict the paths returned by the all-simple-paths algorithm?

Mccortex provides a data structure called "links" for annotating De Bruijn graphs in Cortex format. In the example above, links can be used to store information on a read that covers both the `A` –> `B` and `D` –> `E` junctions. Cortexpy can use these links to performed a "link-informed" (that is, a restricted) all-simple-paths traversal.

3.1 Link-informed graph traversal in `cortexpy`

3.1.1 Cortexpy uses `networkx` algorithms

Cortexpy represents Cortex graphs as `nx.DiGraph` objects[2]. This allows the easy application of `networkx` algorithms to Cortex graphs. `cortexpy` achieves link-informed traversal by wrapping a Cortex graph in a *`LinkedGraphTraverser`* object, which modifies the behavior of the *`__getitem__()`* method. To understand why this works, let us first take a look at the all-simple-paths algorithm:

```
def _all_simple_paths_graph(G, source, targets, cutoff):
    """From networkx.algorithms.simple_paths"""
    visited = collections.OrderedDict.fromkeys([source])
    stack = [iter(G[source])]
    while stack:
        children = stack[-1]
```

(continues on next page)

[1] Cortexpy currently uses a copy of `nx.all_simple_paths()` named `_all_simple_paths_graph()`.
[2] The implementation is not perfect and could use some improvement.

(continued from previous page)

```
7             child = next(children, None)
8             if child is None:
9                 stack.pop()
10                visited.popitem()
11            elif len(visited) < cutoff:
12                if child in visited:
13                    continue
14                if child in targets:
15                    yield list(visited) + [child]
16                visited[child] = None
17                if targets - set(visited.keys()):  # expand stack until find all targets
18                    stack.append(iter(G[child]))
19                else:
20                    visited.popitem()  # maybe other ways to child
21            else:  # len(visited) == cutoff:
22                for target in (targets & (set(children) | {child})) - set(visited.
   ↪keys()):
23                    yield list(visited) + [target]
24                stack.pop()
25                visited.popitem()
```

The key line here is the highlighted line 18. This is the line that appends an iterator of a node's successors to the stack of nodes to visit. The algorithm asks the graph object `G` for the successor nodes of `child` by calling the *`__getitem__()`* method of `G`:

```
G[child]
```

This means that we can restrict the paths returned by `_all_simple_paths_graph()` by restricting the successor nodes returned by `G`.

3.1.2 `LinkedGraphTraverser` restricts simple paths using links

The *`__getitem__()`* method of *`LinkedGraphTraverser`* restricts the returned successors using the following rules:

1. If no link information exists for the query node, then return all successors.
2. Otherwise, return only successors that are consistent with the links encountered on the path from start to query node.
3. If the query node is annotated with links, pick up all links.
4. For each successor node, only retain links that are consistent with the path taken from the start to this successor node.
5. For each successor node, drop links that are no longer relevant to the successor node (i.e. links that have expired)

CHAPTER 4

API reference

4.1 Random access of Cortex graphs

This module contains classes for inspecting Cortex graphs with random access to their kmers.

class `cortexpy.graph.parser.random_access.`**`RandomAccess`**(*graph_handle*, *kmer_cache_size=None*)

Provide fast k-mer access to Cortex graph in log(n) time (n = number of kmers in graph)

`__getitem__`(*lexlo_string*)

Return kmer associated with kmer string

No check is performed to make sure that the input string is a lexicographically-lowest kmer string. Use *`get_kmer_for_string()`* in order to convert a kmer string to its lexlo form before retrieving it from the cortex object.

`__iter__`()

Iterate over kmer strings in graph in order stored in graph

`get_kmer_for_string`(*string*)

Will compute the revcomp of kmer string before getting a kmer

`items`()

Iterate over kmer strings and kmers in graph in order stored in graph

`values`()

Iterate over kmers in cortex graph

4.2 Cortex graph headers

This module contains classes for parsing and representing a Cortex file header

class `cortexpy.graph.parser.header.`**`Header`**(*version=6*, *kmer_size=1*, *kmer_container_size=None*, *num_colors=None*, *mean_read_lengths=None*, *total_sequences=None*, *sample_names=None*, *error_rates=None*, *color_info_blocks=NOTHING*)

Cortex header object

This object allows access to header information contained in a cortex file

classmethod from_stream(*stream*)

Extract a cortex header from a file handle

4.3 Cortex kmers

This module provides classes and functions for working with Cortex kmers.

class `cortexpy.graph.parser.kmer.`**`Kmer`**(*kmer_data*, *num_colors*, *kmer_size*, *revcomp=None*)

Represents a Cortex kmer

This class wraps a kmer data object with attributes and methods for inspecting and manipulating the underlying kmer data object.

`increment_color_coverage`(*color*)

Increment the coverage of a color by one

class `cortexpy.graph.parser.kmer.`**`StringKmerConverter`**(*kmer_size*)

Converts kmer strings to various binary representations

`to_uints`(*kmer_string*)

Converts kmer_string to big-endian uint64 array

`cortexpy.graph.parser.kmer.`**`connect_kmers`**(*first*, *second*, *color*, *identical_kmer_check=True*)

Connect two kmers

`cortexpy.graph.parser.kmer.`**`disconnect_kmers`**(*first*, *second*, *colors*)

Disconnect two kmers

`cortexpy.graph.parser.kmer.`**`find_all_neighbors`**(*first*, *second*)

Return kmers and letters to get from first kmer to second

4.4 Link-informed graph traversal

This module provides classes for parsing Mccortex link files and for traversing graphs while using links.

class `cortexpy.links.`**`LinkOrientation`**

An enumeration.

class `cortexpy.links.`**`LinkWalker`**(*links*, *junctions*)

Manages the loading and walking of links for kmers

`choose_branch`(*base*)

Choose a branch and advance all links. Keep only links consistent with branch.

`load_kmer`(*kmer*)

Load the link group for a kmer in the orientation of the kmer.

next_junction_bases()
Returns the the bases of the branches that can be chosen.

class cortexpy.links.**LinkedGraphTraverser**(*graph*, *walkers=NOTHING*)
Adapter for linked walkers to be able to work with `nx.all_simple_paths()`

class cortexpy.links.**UnitigLinkWalker**(*link_walker*, *unitigs*, *kmer_size*, *current_unitig*)
Traverses a unitig graph with links

choose(*successor*)
Register the choice of a successor and advance

link_successors()
Only returns unitigs based on link information

successors()
Returns nodes from links or all available junctions if no link info exists

class cortexpy.links.**LinkedGraphTraverser**(*graph*, *walkers=NOTHING*)
Adapter for linked walkers to be able to work with `nx.all_simple_paths()`

__getitem__(*item*)
Get the children of `item` according to the walker object associated with `item`

Warning: This scheme only works with depth-first search.

4.5 Representing Cortex graphs as `nx.Graph` objects

This module contains classes for representing Cortex graphs as objects that are compatible with `networkx` algorithms.

todo: Simplify the Graph implementations

class cortexpy.graph.cortex.**ConsistentCortexDiGraph**(*kmer_mapping=NOTHING*, *graph=NOTHING*)
Graph that stores kmer strings that are consistent with each other

class cortexpy.graph.cortex.**CortexDiGraph**(*kmer_mapping=NOTHING*, *graph=NOTHING*)
Stores cortex k-mers and conforms to parts of the interface of `nx.MultiDiGraph`

add_edge(*first*, *second*, ***, *key*)
Note: edges can only be added to existing nodes

nbunch_iter(*nbunch=None*)
Return an iterator over nodes contained in nbunch that are also in the graph.

This code has been copied from `networkx`.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

Parameters **nbunch** (*single node, container, or all nodes (default= all nodes)*) – The view will only report edges incident to these nodes.

Returns **niter** – An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

Return type iterator

Raises `NetworkXError` – If nbunch is not a node or or sequence of nodes. If a node in nbunch is not hashable.

See also:

`Graph.__iter__()`

Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use "if nbunch in self:", even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a `NetworkXError` is raised. Also, if any object in nbunch is not hashable, a `NetworkXError` is raised.

This method was copied from Networkx version 2.1 and then modified

class `cortexpy.graph.cortex.`**`CortexGraphMapping`**(*ra_parser*, *exclusion_set=NOTHING*, *new_kmers=NOTHING*, *n_duplicates=0*)

Create a dict-like kmer mapping from a RandomAccess parser (ra_parser)

The exclusion set tracks kmers deleted from the ra_parser. The new_kmers track kmers that have been added to the mapping. Kmers that exist in both new_kmers and ra_parser are considered overwritten. The kmers in new_kmers have precedence.

`connect_kmers`(*first*, *second*, *color*, *identical_kmer_check=True*)

Connect two kmers

`disconnect_kmers`(*first*, *second*, *colors*)

Disconnect two kmers

`cortexpy.graph.cortex.`**`build_cortex_graph`**(**, *sample_names*, *kmer_size*, *num_colors*, *colors*, *kmer_generator=None*, *kmer_mapping=None*)

Colored de Bruijn graph constructor

`cortexpy.graph.cortex.`**`get_canonical_edge`**(*first*, *second*)

Get canonical edge.

Canonical edges are between lexlo kmers and are ordered lexicographically

Return canonical edge, if the first and second nodes were lexlo

4.6 Interacting with graphs

This module contains classes and functions for inspecting, manipulating, and traversing graphs

class `cortexpy.graph.interactor.`**`SeedKmerStringIterator`**(*seed_kmer_strings*, *unseen_lexlo_kmer_strings*, *seen_lexlo_kmer_strings=NOTHING*)

Iterates seeds and their lexlo representations that exist in the supplied all_kmers:

```
>>> list(SeedKmerStringIterator.from_all_kmer_strings_and_seeds(['AAC'], ['GTT']))
[('GTT', 'AAC')]
```

Kmers that are not in the seed list are return after that:

```
>>> list(SeedKmerStringIterator.from_all_kmer_strings_and_seeds(['AAA', 'AAC'], [
↪'GTT']))
[('GTT', 'AAC'), ('AAA', 'AAA')]
```

Seeds that do not exist in the all_kmers are not returned.

```
>>> list(SeedKmerStringIterator.from_all_kmer_strings_and_seeds([], ['CCC']))
[]
```

Returned kmers from all_kmers list are returned in order.

```
>>> list(SeedKmerStringIterator.from_all_kmer_strings_and_seeds(['AAA', 'AAG',
↪'AAC'], []))
[('AAA', 'AAA'), ('AAG', 'AAG'), ('AAC', 'AAC')]
```

`cortexpy.graph.interactor.`**`edge_nodes_of`**(*graph*)

Find all edge nodes of a graph

Second return value is direction of edge.

`cortexpy.graph.interactor.`**`make_copy_of_color_for_kmer_graph`**(*graph*, *color*, *include_self_refs=False*)

Makes a copy of graph, but only copies over links with key=color. Only copies over nodes that are linked by a link with key=color.

4.7 Utility functions

This module contains utility functions that are used inside cortexpy. These functions may also be useful outside of cortexpy.

`cortexpy.utils.`**`kmerize_contig`**(*contig*, *kmer_size*)

Return generator of kmers in contig

The returned kmers are not lexicographically lowest.

```
>>> list(kmerize_contig('ATTT', 3))
['ATT', 'TTT']
```

`cortexpy.utils.`**`kmerize_fasta`**(*fasta*, *kmer_size*)

Return generator to all kmers in fasta

`cortexpy.utils.`**`lexlo`**

Return lexicographically lowest version of a kmer string and its reverse complement

The reverse complement of a kmer string is generated and the lexicographically-lowest kmer string is returned.

```
>>> lexlo('AAA')
'AAA'
```

```
>>> lexlo('TTT')
'AAA'
```

CHAPTER 5

License

Cortexpy is distributed under the Apache Lincense version 2.0:

```
                              Apache License
                        Version 2.0, January 2004
                     http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
```

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index