Domain-level Reaction Enumerator

Karthik Sarma, Casey Grun, and Erik Winfree.

Overview

This package predicts the set of possible reactions between a set of initial nucleic acid complexes. Complexes are comprised of strands, which are subdivided into "domains"—contiguous regions of nucleotide bases which participate in Watson-Crick hybridization. The enumerator only considers reactions between complexes with complementary domains. At this point, only unpseudoknotted intermediate complexes are considered.

This document describes basic usage of the software, automatic generation of API documentation, and running of unit tests. There's a separate document, `architecture.pdf`, which describes the internal architecture of the software.

This package is written for Python 2.7; Python must be installed and in the user's `path` in order to run the program.

Usage

```
usage: enumerator.py [-h] [--infile INPUT_FILENAME]
                     [--outfile OUTPUT_FILENAME] [-o OUTPUT_FORMAT]
                     [-i INPUT_FORMAT] [-c]
                     [--max-complex-size MAX_COMPLEX_SIZE]
                     [--max-complexes MAX_COMPLEX_COUNT]
                     [--max-reactions MAX_REACTION_COUNT]

Domain-level nucleic acid reaction enumerator

optional arguments:
  -h, --help            show this help message and exit
  --infile INPUT_FILENAME
                        Path to the input file (default: None)
  --outfile OUTPUT_FILENAME
                        Path to the output file (default: None)
  -o OUTPUT_FORMAT      Desired format for the output file; one of: pil, crn,
                        standard, json, legacy, sbml, graph (default:
                        standard)
```

```
-i INPUT_FORMAT       Desired format for the input file; one of: pil,
                      standard, json (default: standard)
-c                    Condense reactions into only resting complexes
                      (default: False)
--max-complex-size MAX_COMPLEX_SIZE
                      Maximum number of strands allowed in a complex (used
                      to prevent polymerization) (default: None)
--max-complexes MAX_COMPLEX_COUNT
                      Maximum number of complexes that may be enumerated
                      before the enumerator halts. (default: None)
--max-reactions MAX_REACTION_COUNT
                      Maximum number of reactions that may be enumerated
                      before the enumerator halts. (default: None)
```

Input formats

There are 2 input formats available, which may be specified using the `-i` option:

- *Standard input format* (`.enum`) – This is a simple format that is specific to the enumerator. A simple example of the format is included below. The format has three types of statements:
  - `domain` statements declare individual domains, as follows: `domain name : specification`, where:
    - `name` is the name of the domain (e.g. `a`, `1`, `th`, etc.)
    - `specification` is either the length of the domain (e.g. a number of bases, or just `long` or `short`) or a sequence (e.g. `NNNNNNN` or `ATTACG` or even a mixture of specific and degenerate bases `AANATCY`)
  - `strand` statements group domains into strands, as follows: `strand name : domains`, where:
    - `name` is the name of the strand
    - `domains` is a space-separated list of domains
  - `complex` statements group strands into complexes and assign them a secondary structure, as follows:

    ```
    complex name :
    strands
    structure
    ```

    where:
    - `name` is the name of the complex
    - `strands` is a space-separated list of strands

    - `structure` is a domain-wise description of the structure in dot-parenthesis notation

- *Pepper Intermediate Language* (`.pil`) – PIL is a general-purpose format for describing domain-level secondary structures. The Pepper Intermediate Language (PIL) follows the basic constructions of the Pepper language, but disallows some features (sequence constraints, components, etc.). Each PIL line should consist of a directive of one of the following forms:
    - `sequence <name> = <sequence>` – declare a new domain
    - `strand <name> = <list of sequences and explicit constraints>` – declare a strand comprised of domains
    - `structure <name> = <list of strands> : <secondary structure>` – declare a complex comprised of several strands
    - `kinetic <input structures> -> <output structures>` – declare a reaction between several structures

Here is a simple example of the standard input format:

```
# This file describes the catalytically generated 3 arm junction
# described in Yin et. al. 2008

domain a : 6
domain b : 6
domain c : 6
domain x : 6
domain y : 6
domain z : 6

strand I : y* b* x* a*
strand A : a x b y z* c* y* b* x*
strand B : b y c z x* a* z* c* y*
strand C : c z a x y* b* x* a* z*

complex I :
I
....

complex A :
A
.(((..)))

complex B :
B
.(((..)))
```

```
complex C :
C
.(((..)))

complex ABC :
A B C
(((((((. + ))))((((. + ))))))).
```

Output formats

There are 6 output formats available, which may be specified using the `-o` option:

- Graphical results (`json`/`enjs`) – produces a file which can be rendered into a graphical, interactive network by the DyNAMiC Workbench package and exported to SVG. This file is also a valid JSON file and may be suitable for consumption by other tools.
- Pepper Intermediate Language (`pil`) – produces a representation of the network, including reactions, in the Pepper Intermediate Language
- Chemical Reaction Network (`crn`) – produces a list of simple reactions between chemical species
- Systems Biology Markup Language (`sbml`) – produces a representation using the Systems Biology Markup Language, an industry standard format for modeling biological and chemical networks. SBML can be consumed by a reaction simulator, such as COPASI
- Legacy (`legacy`) – produces output in the format of Brian Wolfe's old enumerator
- Graph (`graph`) – produces an EPS file showing the reaction network, laid out using Graphviz

Building documentation

API Documentation is built from comments in the source using Sphinx; Sphinx must be installed. Then you can run:

```
make docs
```

from within the main directory to build HTML documentation; you can find this documentation at `docs/_build/html/index.html`. Additional output formats are available, and can be generated by moving to the `docs/` subdirectory and using `make`. Type `make` within the `docs/` subdirectory to show a list of available output formats. Once you've generated the documentation, it will be available in the folder `docs/_build/{format}`.

This document, and the architecture documentation, are generated from Markdown with Pandoc in PDF or HTML format; Pandoc must be installed; then you can use `make README.pdf` or `make README.html`, or similarly `make architecture.pdf` or `make architecture.html`.

Running unit tests

Unit tests for the project are written using Nosetests. Nosetests must be installed. Then you can run:

```
make tests
```

from within the main directory to run unit tests.