Enumerator Architectural Documentation

Application lifecycle

The enumerator application is bootstrapped from the `main` function defined in `enumerator.py`. This function:

- Parses command line arguments using the Python `argparse` module
- Invokes an input parser from the `input` module to parse the passed input file. Input parsers generate `Enumerator` objects.
- Sets various parameters on the `Enumerator` object that is created, such as the maximum complex size and the maximum number of reactions to be enumerated.
- Instructs the `Enumerator` object to run the reaction enumeration.
- Invokes an output generator to serialize the output from the enumerator to a file.

Key data structures

The `utils` module defines several classes that are used to represent the state of the enumeration:

- `Domain` objects represent single domains (continuous regions of nucleotides that hybridize as a unit).
- `Strand` objects represent strands—ordered collections of `Domain`s.
- `Complex` objects represent complexes of strands—comprised of an ordered list of `Strand`s and a secondary structure specifying how strands are hybridized together.
- `RestingState` objects represent collections of `Complex`es that form a resting state—that is, a complex with no outgoing fast reactions.

Additionally, the `reactions` module defines the `ReactionPathway` class:

- `ReactionPathway` objects represent a reaction between some number of substrate `Complex`es and some number of product `Complex`es.

Reaction functions

The `reactions` module defines a number of "reaction functions"; these functions correspond to different types of reactions—for instance, unimolecular binding (`bind11`), bimolecular binding (`bind21`), 3-way branch migration (`branch_3way`), etc. Reaction functions all accept some number of `Complex`es as arguments, and

return a list of `ReactionPathway` objects. The number of arguments accepted by a reaction function (which is therefore also the number of substrates for the reaction) is its *arity*.

Reaction functions are grouped by their arity and their "speed" in two dictionaries within the `enumerator` module: `fast_reactions` and `slow_reactions`. Each is a dictionary where the keys are arities (e.g. 1 for unimolecular, 2 for bimolecular, etc.), and the values are lists of reaction functions. Currently, all unimolecular reaction functions are fast, and the `bind21` reaction is slow. The fast/slow distinction is used by the enumerator to group complexes into "resting state complexes" and "transient complexes"—resting state complexes exist in strongly-connected neighborhoods with no out-going fast reactions ("resting states"), while transient complexes are those complexes in neighborhoods with one or more outgoing fast reactions.

Reaction enumeration

The main reaction enumeration is performed by the `Enumerator` class. The `Enumerator` class is initialized with a set of `Domain`s and `Strand`s, as well as a set of initial `Complex`es. The `enumerate` method is called to begin the following procedure, which exhaustively enumerates the network of possible reactions starting from those initial `Complex`es:

- First, fast reactions are considered between the initial complexes: For each complex in `initial_complexes`, the `process_neighborhood` method is called and passed that `Complex`.
  - This method generates the "neighborhood" of complexes reachable by fast reactions from the passed reaction, and classifies the resulting complexes as either transient or resting state. This is a depth-first search, which proceeds as follows:
    * The starting complex is placed in a list `_F` by itself.
    * While that list is not empty:
      · an element is taken from the list and `get_fast_reactions` is passed that complex. `get_fast_reactions` executes each of the reaction functions in `fast_reactions` on the passed complex.
      · Resulting `ReactionPathway` objects are added to the list `N_reactions`.
      · Product `Complex`es from these reactions are added to the list `_F` and the list `_N`.
    * The `segment_neighborhood` function is passed the list `_N`, which contains all complexes that have been enumerated through fast reactions. `segment_neighborhood` breaks the complexes in `_N` into strongly-connected components using

Tarjans' algorithm. The resulting `transient_state_complexes`, `resting_state_complexes`, and `resting_states` are collected into the appropriate lists and stored in the `Enumerator` object.

- Slow reactions are then considered between all complexes classified as resting states. For each complex classified as a resting state:
  - `get_slow_reactions` is called and passed the `Complex`. `get_slow_reactions` works similarly to `get_fast_reactions`: it executes each of the reactions in `slow_reactions`. However, while `fast_reactions` can only be unimolecular, reaction functions in `slow_reactions` may be bimolecular; therefore, `get_slow_reactions` must iterate through all complexes classified as resting states and attempt slow reactions between those complexes and the passed `complex`.
  - Resulting `ReactionPathway`s and product `Complex`es are collected. For each of the product `Complex`es, `process_neighborhood` is invoked. `process_neighborhood` enumerates fast reactions and segments strongly-connected components, as described above.

Input and output

The `input` module defines a number of *input parsers*, which are functions that accept input filenames and return `Enumerator` objects. The `output` module defines a number of *output generators*, which are functions that accept `Enumerator` objects and output filenames, and serialize the `Enumerator` objects to the output file as text.

Input parsers are collected in the `text_input_functions` and `load_input_functions` dictionaries of the `input` module. `text_input_functions` generate new `Enumerator` objects which can then be used to enumerate reactions; `load_input_functions` genereate `Enumerator` objects which have been serialized from a *previous* run of the enumerator.

Output parsers are collected in the `text_output_functions` and `graph_output_functions` dictionaries of the `output` module. `text_output_functions` contains text-based output generators, while `graph_output_functions` generate graphical output.