MetaFlux: A Dank Framework

iGEM UofT

Julian Mazzitelli

1 Introduction

Given the rapid advancements in genome-scale reconstructions and metagenomics over the past several years [ref.], there is a need to effectively and succintly analyze and interpret this data. We present *MetaFlux*, a web based tool for visualing, modifying, and comparing analyses of genome scale metabolic models. *MetaFlux* can dynamically present community level interactions as well as single species models. [How to write as one sentence?]

2 Data

MetaFlux visualizes genome models. Currently, the models available within our app are *Systems Biology Markup Language* (SMBL) formatted models as `xml` files taken from the literature [refs.]. We are interested in incorporating autogenerated models from tools such as MicrobesFlux and Model SEED in the future. Morever, the ability for a user to upload a model will be implemented.

2.1 Model Schema For Single Species

The following is a BSON implementation of an SMBL metabolite model schema which can be saved in a MongoDB NoSQL database. The following snippet is directly operable within a NodeJS environment.

```
module.exports = mongoose.model('metabolicmodel', new mongoose.Schema({
        reactions: [{
                subsystem: String,
                name: String,
                upper_bound: Number,
                lower_bound: Number,
                objective_coefficient: Number,
                variable_kind: String,
                id: String,
                gene_reaction_rule: String,
                metabolites: [{
                        id: String,
                        stoichiometric_coefficient: Number
                }]
        }],
        description: String,
         notes: String,
          genes: [{
                name: String,
                id: String
        }],
          metabolites: [{
                name: String,
```

```
                notes: String,
                annotation: String,
                _constraint_sense: String,
                charge: Number,
                _bound: Number,
                formula: String,
                compartment: String,
                id: String
        }],
          id: String
}));
```

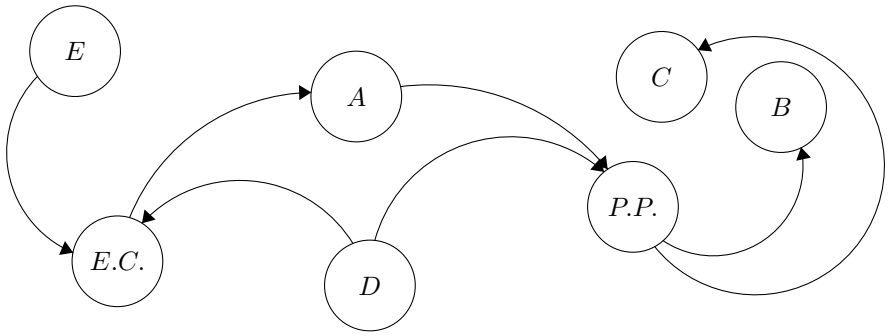
These are all of the properties present in the literature retreived models. However, further attributes (such as `flux_value`) will be appended into the model.

2.2 Model Schema for Community Structure

In our community level representations, different species are described using seperate compartments. Reactions involving shared extracellular metabolites are constructed for each specie to mediate meteabolite exchange through the extracellular space.

3 Visualization

From a community level, one may see the following visualization:



In the above graphic each arrow represents an enzmyatic mediated reaction. *E.C.* represents the species *E. Coli* and *P.P.* represents *P. Putida*. *A* is a metabolite exported by *E.C* and imported by *P.P.* *B* and *C* are metabolites exported by *P.P.* *D* is a metabolite which is imported by both species, and potentially conferring to a competitive relationship. Finally, *E* is a metabolite imported into *E. Coli*. We arrive at set of potential relationships between two species *a,b* and a metabolite *m*:

- Single species exporting a metabolite: $\{a, b\} \rightarrow m$
- Single species importing a metabilte: $m \rightarrow \{a, b\}$
- Synergistic export/import between two species: $\{a \rightarrow m \rightarrow b, b \rightarrow m \rightarrow a\}$
- Competition for a metabolite between two species: $\{m \rightarrow a, b\}$

Given the above definition, we can generalize to any number of "compartments" and any number of metabolites. Furthermore, this additionally provides the means to segregate the intracellular compartments of a single species model, such as the cytosol, periplasm, and internal subsystems. The ability to create new subsystems from within the web interface will be added in a later release.

4 Graph Construction

Once we have been provided with model data in the schema format as described above, so begins the graph construction process. This process aims to be as parametizable and generalizable as possible, in order to accomodate the creation of custom groupings.

The first thing to do is construct `Metabolite` and `Reaction` objects.

```
buildPeices = (model) ->
    nodes = {}, reactions = {}

    for metabolite in model.metabolites
        nodes[metabolite.id] = new Metabolite(metabolite, ...)

    for reaction in model.reactions
        reactions[reaction.id] = new Reaction(metabolite, ...)

        for metabolite of reaction.metabolites
            if reaction.metabolites[metabolite] > 0
                source = reaction.id
                target = metabolite
            else
                source = metabolite
                target = reaction.id

            reactions.addLink(new Link(source, target, ..))
```

After this is done, we have two dictonaries storing all metabolites and reactions that will be used to construct our system. The next step is to use the prototype methods of the `Reaction` object to construct "compartments" as necessary. This will be done recursively, and with paramaters and defaults outling the sets of compartments. For example, one set could be $\{E.C., extracellular, P.P.\}$, while another, nested within $E.C.$ could be $\{cytosol, periplasm, extracellular\}$.

```
# Will be called recursively
# The first call will pass in nodes, reactions
# set of reaction types? can be built w/ reactions
# Constructs a graph object
compartmentatilization = (system) ->
    # TODO
```