Biological Regulatory Inferential Additive Network (BRIAN) Specification

Introduction

The human body is a carefully engineered biological system, consisting of approximately 30 trillion cells — all which must perform a diverse and dynamic set of functions to support daily living. But how do all these cells know which role to perform and successfully carry out their functions? The answer lies in biological regular networks, or BRNs, consisting of interconnected cellular components, such as genes, proteins, and metabolites, that interact through regulatory mechanisms to control the expression of genes, cellular activities, and organismal functions.

With a single cell in humans often containing over 10,000 different types of protein, as well as each protein having a host of possible post-translational modifications, constructing a BRN becomes a graph-navigating problem which can quickly evolve to be a computational nightmare. This is where Biological Regulatory Inferential Additive Network (BRIAN) language hopes to offer improvements in the workflow of a lab scientist by automatically constructing possible biological regulatory networks from the user's input of simple existing relationships between cellular components discovered through experimental work. The language would be capable of translating this experimentally established relationship data into an adjacency matrix representation of a BRN, each row/column representing a specific component and indices representing relationships. Ultimately, the BRIAN language could use this matrix to construct a useful graphical representation of the described biological system and present possible new relationships that were gleaned after evaluation to the user.

As a programming language, BRIAN's strength lies in its ease of use: programs from a text file consist of a list of observed relationships between cellular components, which does not require the user to be acquainted with any level of coding. In this sense programs are extremely modular in which new experimental results can simply be appended to an already existing program. The language itself can also identify logical conflicts in the relations provided in the program through its evaluation of the adjacency matrix (described in further detail below).

Design Principles

On the technical side, BRIAN hopes to be a highly scalable and flexible language in the following manner:

- Since relationships between cellular components are ultimately stored in an adjacency matrix, adding new relationships between existing components has a very low storage and computational cost. Additionally, adding new components is performed by simply expanding the existing matrix. As only a single relationship between two components can exist (i.e. unknown, activates, or inhibits), a matrix of ints is sufficient to represent the BRN.
- BRIAN accommodates all existing proteins, genes, phenotypes (as well as an [other] category for components such as small RNAs or metabolites) and allows interactions between all these types freely. The language also does not strictly define the type of relationship between two cellular nodes: while the expected relationship types are abstract interactions (e.g. activation, inhibition), if these are not known, the biochemical modification (e.g. phosphorylation, ubiquitination, acetylation, methylation) are also accepted.

Aesthetically, BRIAN aims to be a clear and expressive language. Since BRIAN is a strongly-typed language, it would help users avoid confusion of variables. For example, insulin is both a protein and a gene, using BRIAN, the user can be sure which variable is under consideration at all times. However, in the user's input this distinction can easily be made by following the common biological syntax of gene names starting with a lowercase letter and protein names with an uppercase.

Examples

Sample Program #1:
Input:

```
eval test1.txt
    test1.txt file:
    "(Component Interaction) ROCK inhibits MYPT1
    (Component Modification) MYPT1 phosphorylates MLC
    (Modification Effect) Phosphorylation inhibits MLC"
```

Output (in addition to graphical output):

```
(Found) Protein_A activates Protein_A
(Found) Protein_A activates Protein_C
(Found) gene_b activates Protein_C
(Found) gene_b inhibits Protein_D
```

Sample Program #2:
Input:

```
eval test2.txt
    test2.txt file:
    "ROCK inhibits MYPT1
    MYPT1 phosphorylates MLC
    ROCK activates MLC
    MLC activates *Focal_Adhesion_Formation"
```

Output (in addition to graphical output):

```
(Found) Phosphorylation inhibits MLC
(Found) MYPT1 inhibits *Focal_Adhesion_Formation
(Found) ROCK activates *Focal_Adhesion_Formation
```

Sample Program #3:
Input:

```
eval test3.txt
    test3.txt file:
    "rock activates mlc
    mlc inhibits rock
    rock activates rock"
```

Output (in addition to graphical output):

```
ERROR: Initial information in BRN is contradictory.
```

Language Concepts

A typical user is expected to be familiar with cellular components, such as proteins, genes and phenotypes, which serve as primitives for BRIAN. When the user inputs a text file describing observed relationships between components, BRIAN constructs a fitting BRN and infers new relationships between the variables. In essence, BRIAN's evaluator is designed to consider all possible combinations between the identified primitives and offer any new found ones as a result.

The combining forms of cellular components mainly include more abstract interactions (e.g. activation, repression), or if these are not known, the biochemical modification (e.g. phosphorylation, ubiquitination, acetylation, methylation) between two cellular components (the primitives). Upon receiving a correct file of relationship descriptions from the user, BRIAN parses and stores all relationships between cellular components in a list of relations.

Syntax

$$
\begin{aligned}
\langle gene \rangle &::= \langle \alpha \in \{a...z\} \rangle \langle \alpha \in \{a...z\} \cup \{A...Z\} \cup \{\mathbb{Z}\} \cup \{\_, -\} \rangle^* \\
\langle protein \rangle &::= \langle \alpha \in \{A...Z\} \rangle \langle \alpha \in \{a...z\} \cup \{A...Z\} \cup \{\mathbb{Z}\} \cup \{\_, -\} \rangle^* \\
\langle phenotype \rangle &::= * \langle \alpha \in \{a...z\} \cup \{A...Z\} \cup \{\mathbb{Z}\} \cup \{\_, -\} \rangle^+ \\
\langle other \rangle &::= \# \langle \alpha \in \{a...z\} \cup \{A...Z\} \cup \{\mathbb{Z}\} \cup \{\_, -\} \rangle^+ \\
\langle variable \rangle &::= \langle gene \rangle \\
&\quad | \langle protein \rangle \\
&\quad | \langle phenotype \rangle \\
&\quad | \langle other \rangle \\
\langle relationship \rangle &::= \langle variable \rangle \text{ activates } \langle variable \rangle \\
&\quad | \langle variable \rangle \text{ inhibits } \langle variable \rangle \\
\langle sequence \rangle &::= \langle relationship \rangle^+
\end{aligned}
$$

Semantics

(i) About Primitives:

- `Gene` is a primitive of type `string`. `Gene` types must start with a lowercase letter.
- `Protein` is a primitive of type `string`. `Protein` types must start with a capital letter.
- `Phenotype` is a primitive of type `string`. `Phenotype` types must start with a "$*$" symbol.
- `Other` is a primitive of type `string`. The aim of `Other` is to capture any exceptional cellular components a user might wish to enter that do not currently have a formal variable representation. `Other` types must start with a "#" symbol.

(ii) About Combining Forms:

The primary combining form of the language is `Relationship` of type `Variable*Variable` (and thus type `string*string`). `Relationship` currently has two subtypes: `Activation` and `Inhibition`. The combining form `Sequence` is a list of `Relationships`.

(iii) About Evaluation

Aside from the program which is contained in a .txt file, no additional input is provided.

To understand how evaluation works for BRIAN, let's consider our first example. When we call

```
eval test1.txt
```

BRIAN parses the text file and classifies the cellular components into their appropriate primitive types.

```
test1.txt file:
    "Protein_A activates gene_b
    gene_b activates Protein_A
    gene_b activates Protein_C
    Protein_C inhibits Protein_D
    Protein_D inhibits Protein_D"
```

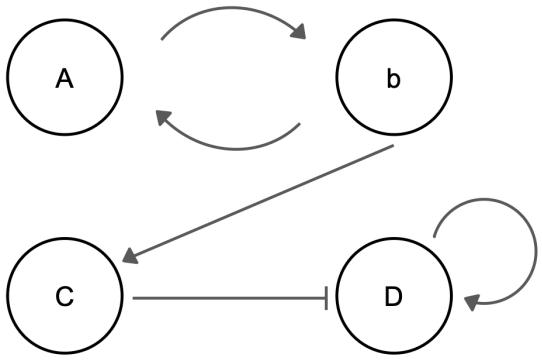
After parsing, BRIAN compiles a list of all unique cellular components, which in our case we have four:

```
[Protein "Protein_A"; Gene "gene_b"; Protein "Protein_C"; Protein "Protein_D"]
```

Then, a matrix (call it P) is constructed from all the parsed relationships:

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

, where 1 signifies activation, 2 signifies inhibition and 0 means no relationship. The P matrix successfully captures the following BRN graph:



Using the P matrix, we are now ready to search for new relationships using matrix multiplication. The next step is using matrix multiplication to find new relationships. This operates under the biological logic of:

```
(A activates B, B activates C) implies A activates C
    In the matrix: 1 * 1 = 1
(A activates B, B inhibits C) implies A inhibits C
    In the matrix: 1 * 2 = 2
(A inhibits B, B activates C) implies A inhibits C
    In the matrix: 2 * 1 = 2
(A inhibits B, B inhibits C) implies A activates C
    In the matrix: 2 * 2 = 4, 4 is then converted to 1 (activation)
```

Since the underlying relationship between cellular components does not change, we may model this traversal problem as a quasi-Markov Chain process. The transition matrix for our MC model would contain information on existing relationships, which we may probabilistically interpret as a sure event that state i would go to state j. So, at each step (of multiplying P to itself), we capture the transfer of relationship ij to relationship jk in our next step, which is true to how BRN relationships function.

However, to avoid the summation of multiple values, each row and column of a matrix can only contain one value before being multiplied by itself. Following this logic, the following matrix derivations are made by zeroing certain values:

$$\text{Matrix1} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\text{Matrix2} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

$$\text{Matrix3} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\text{Matrix4} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

And the results of multiplication:

$$\text{Matrix1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\text{Matrix2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

$$\text{Matrix3} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\text{Matrix4} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

The new values from these matrices can be added back to the original (in cases where the multiplied matrices present a different value at the same index as the original matrix, the original value is kept).

$$P' = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

From this new matrix, the following relationships are found:

```
(Found) Protein_A activates Protein_A
(Found) Protein_A activates Protein_C
(Found) gene_b activates Protein_C
(Found) gene_b inhibits Protein_D
```

Using this new graph, the same matrix derivation and multiplication process can be repeated to find new relationships, up to n times where n is the number of unique elements.