TGS.R

Saptarshi Pyne, Manan Gupta and Ashish Anand

Chapter 1

A Quick Example - Drosophila

1.1 Running the Datasets

In this example we use Drosophila melanogaster Life Cycle (DmLc) datasets that have been experimentally produced by Arbeitman et. The discretized DmLc dataset is obtained from the KELLER website **link to datasets**.

The datasets are slightly modified to create time series instead of a single timestamp observation. These datasets can be found in the /extdata directory of the package. For more information on the modification of datasets refer to the **main paper** The datasets are named DmLc3E, DmLc3L, DmLc3P, DmLc3A corresponding to embryo, pupil, larva and adult stages of drosophila.

To run LearnTgs on these datasets call the function as follows:-

```
## Learn DmLc3E.RData dataset
> LearnTgs(0,input.data.filename = "DmLc3E.RData", num.timepts = 6, is.discrete = TRUE,
        num.discr.levels = 2, mi.estimator = "mi.pca.cmi", apply.aracne = FALSE,
        clr.algo = "CLR", max.fanin = 14, allow.self.loop = TRUE,
        input.dirname = "location where file is stored",
        output.dirname = "location where output needs to be stored")

## Learn DmLc3L.RData dataset
> LearnTgs(0,input.data.filename = "DmLc3L.RData", num.timepts = 2, is.discrete = TRUE,
        num.discr.levels = 2, mi.estimator = "mi.pca.cmi", apply.aracne = FALSE,
        clr.algo = "CLR", max.fanin = 14, allow.self.loop = TRUE,
        input.dirname = "location where file is stored",
        output.dirname = "location where output needs to be stored")

## Learn DmLc3P.RData dataset
> LearnTgs(0,input.data.filename = "DmLc3P.RData", num.timepts = 3, is.discrete = TRUE,
        num.discr.levels = 2, mi.estimator = "mi.pca.cmi", apply.aracne = FALSE,
        clr.algo = "CLR", max.fanin = 14, allow.self.loop = TRUE,
        input.dirname = "location where file is stored",
        output.dirname = "location where output needs to be stored")

## Learn DmLc3A.RData dataset
> LearnTgs(0,input.data.filename = "DmLc3A.RData", num.timepts = 2, is.discrete = TRUE,
        num.discr.levels = 2, mi.estimator = "mi.pca.cmi", apply.aracne = FALSE,
        clr.algo = "CLR", max.fanin = 14, allow.self.loop = TRUE,
        input.dirname = "location where file is stored",
        output.dirname = "location where output needs to be stored")
```

In order to run the DmLc3E.RData dataset with json file as input, create a json file and give it an appropriate name (say example.json). The json file should look like this:

```
{
   "input.data.filename": "DmLc3E.RData",
   "num.timepts": 6,
   "true.net.filename": "",
   "input.wt.data.filename": "",
   "is.discrete": true,
   "num.discr.levels": 2,
   "discr.algo": "",
   "mi.estimator": "mi.pca.cmi",
   "apply.aracne": false,
   "clr.algo": "CLR",
   "max.fanin": 14,
   "allow.self.loop": true
}
```

And then call the function as follows:

```
## Learn DmLc3E.RData dataset
> LearnTgs(1,json.file = " location of json file. eg - /home/cse/Desktop/example.json",
        input.dirname = "location where file is stored",
        output.dirname = "location where output needs to be stored")
```

The json files for the examples are also available in the /extdata directory.

The net.sif files that are generated can be plotted in **Cytoscape**. The results are as follows.

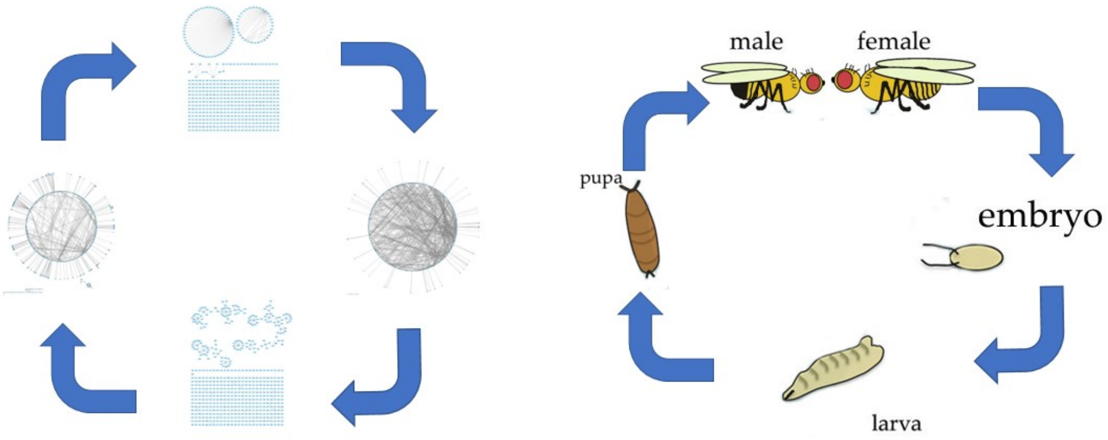

Figure 1.1: Resulting graphs from LearnTgs alogorithm

1.2 Analysis of the results

Before diving into the analysis, we introduce what prd gene is. ’prd’ is a gene that is known to have a positive cell specificity in the embryo stage. It is also known that ‘prd’ participates in the regulation of anterior-posterior segmentation of the embryo.

Therefore there should be an edge between ’eve’ and ’prd’. This information has been retrieved from **TRANSFAC Public Database version 7.0** which is claimed to be the gold standard in the area of transcriptional regulation.

We know look at the graph obtained from DmLc3E.RData dataset.

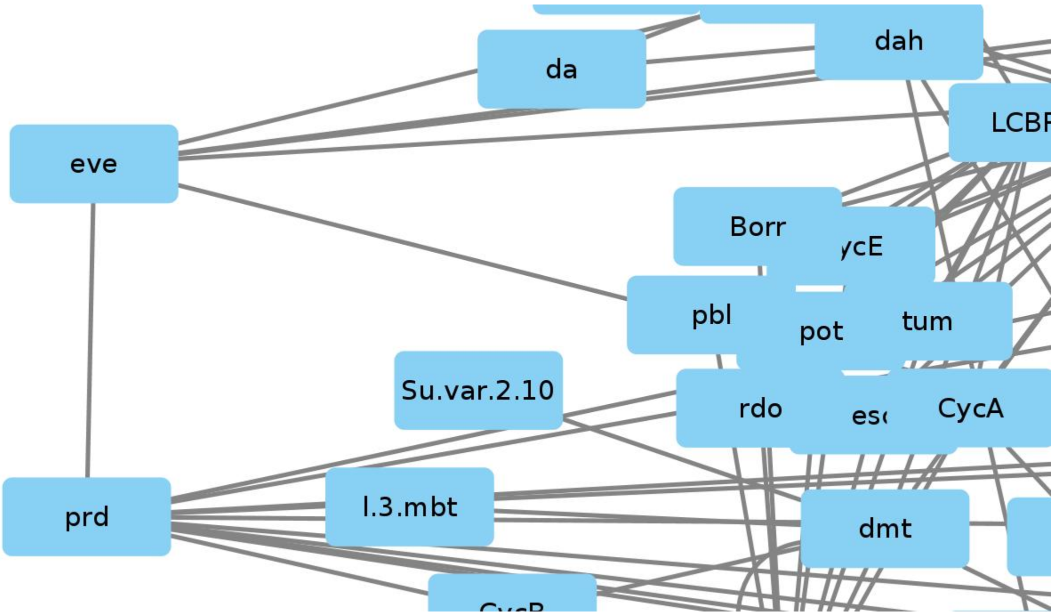

Figure 1.2: Magnified graph of DmLc3E dataset

The magnified image reveals that there indeed is an edge between ’prd’ and ’eve’ in the graph obtained from running LearnTgs on DmLc3E.RData.

This provides a biological support for the results of the algorithm.

For further analysis refer to section 4.8 of the **main paper**.

Chapter 2

Introduction

2.1 Rapid Reconstruction of Time-varying Gene Regulatory Networks

Rapid advancement in high-throughput gene expression measurement technologies has resulted in genome-scale time series datasets. Uncovering the underlying temporal sequence of gene regulatory events in the form of time-varying Gene Regulatory Networks (GRNs) demands computationally fast, accurate and highly scalable algorithms. To provide a flexible framework in a significantly time-efficient manner, a novel algorithm, namely TGS, is proposed here. TGS is shown to consume only 29 minutes for a microarray dataset with 4028 genes. Moreover, it provides the flexibility and time-efficiency, without losing the accuracy. Nevertheless, TGS's main memory requirement grows exponentially with the number of genes, which it tackles by restricting the maximum number of regulators for each gene. Relaxing this restriction remains an important challenge as the true number of regulators is not known a prior.

2.2 The Time Complexity of the *TGS+* Algorithm

$$\begin{aligned}
T_{\text{TGS+}}(V) &= T_{\text{ARACNE}}(V) + T_{\text{TGS}}(V) \\
&= \mathcal{O}\left(V^3\right) + \Big(\mathcal{O}\left(V^2\right) + o\Big((T-1)\,V M_f^2\; 2^{\left(M_f - 2\right)}\Big)\Big) \qquad (2.1) \\
&\quad \text{(By Algorithm 5 and Equation 1 of the main paper)} \\
&= \mathcal{O}\left(V^3\right) + o\Big(V^3\;(\lg V)^2\Big) \qquad \text{(By Equation 2 of the main paper)} \\
&= \mathcal{O}\left(V^3\right) + \mathcal{O}\Big(V^3\;(\lg V)^2\Big) \\
&= \mathcal{O}\Big(V^3\;(\lg V)^2\Big) \qquad (2.2)
\end{aligned}$$

Reference to the main paper `http://dx.doi.org/10.1109/TCBB.2018.2861698`

2.3 A Flowchart of the *TGS* Algorithm

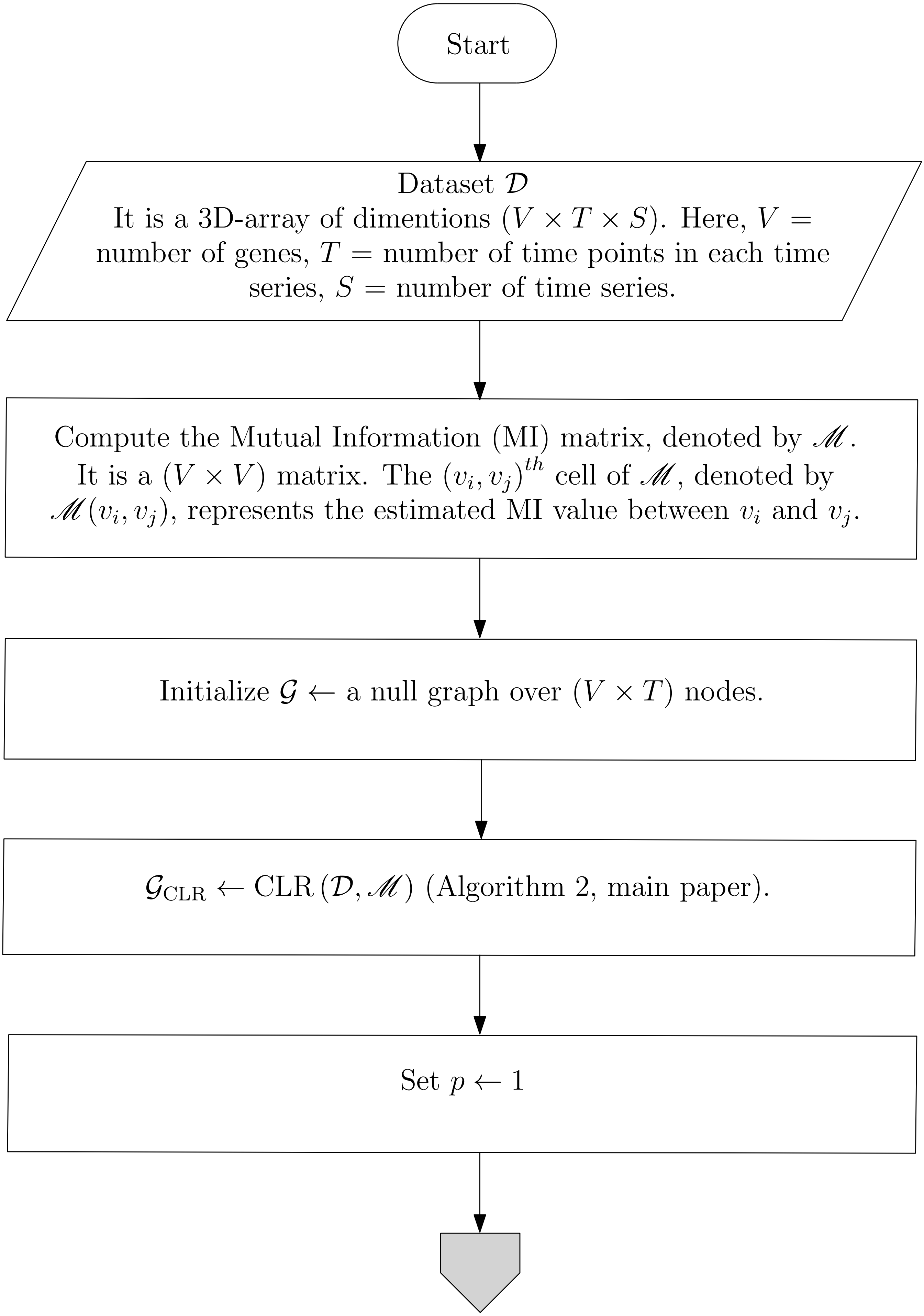
Start
Dataset $\mathcal{D}$
It is a 3D-array of dimentions $(V \times T \times S)$. Here, V = number of genes, T = number of time points in each time series, S = number of time series.
Compute the Mutual Information (MI) matrix, denoted by $\mathscr{M}$. It is a $(V \times V)$ matrix. The $(v_i, v_j)^{th}$ cell of $\mathscr{M}$, denoted by $\mathscr{M}(v_i, v_j)$, represents the estimated MI value between v_i and v_j.
Initialize $\mathcal{G} \leftarrow$ a null graph over $(V \times T)$ nodes.
$\mathcal{G}_{\text{CLR}} \leftarrow \text{CLR}(\mathcal{D}, \mathscr{M})$ (Algorithm 2, main paper).
Set $p \leftarrow 1$

Figure 2.1: Flowchart of the *TGS* algorithm (Algorithm 3, main paper). The flowchart is continued in Figure 2.2.

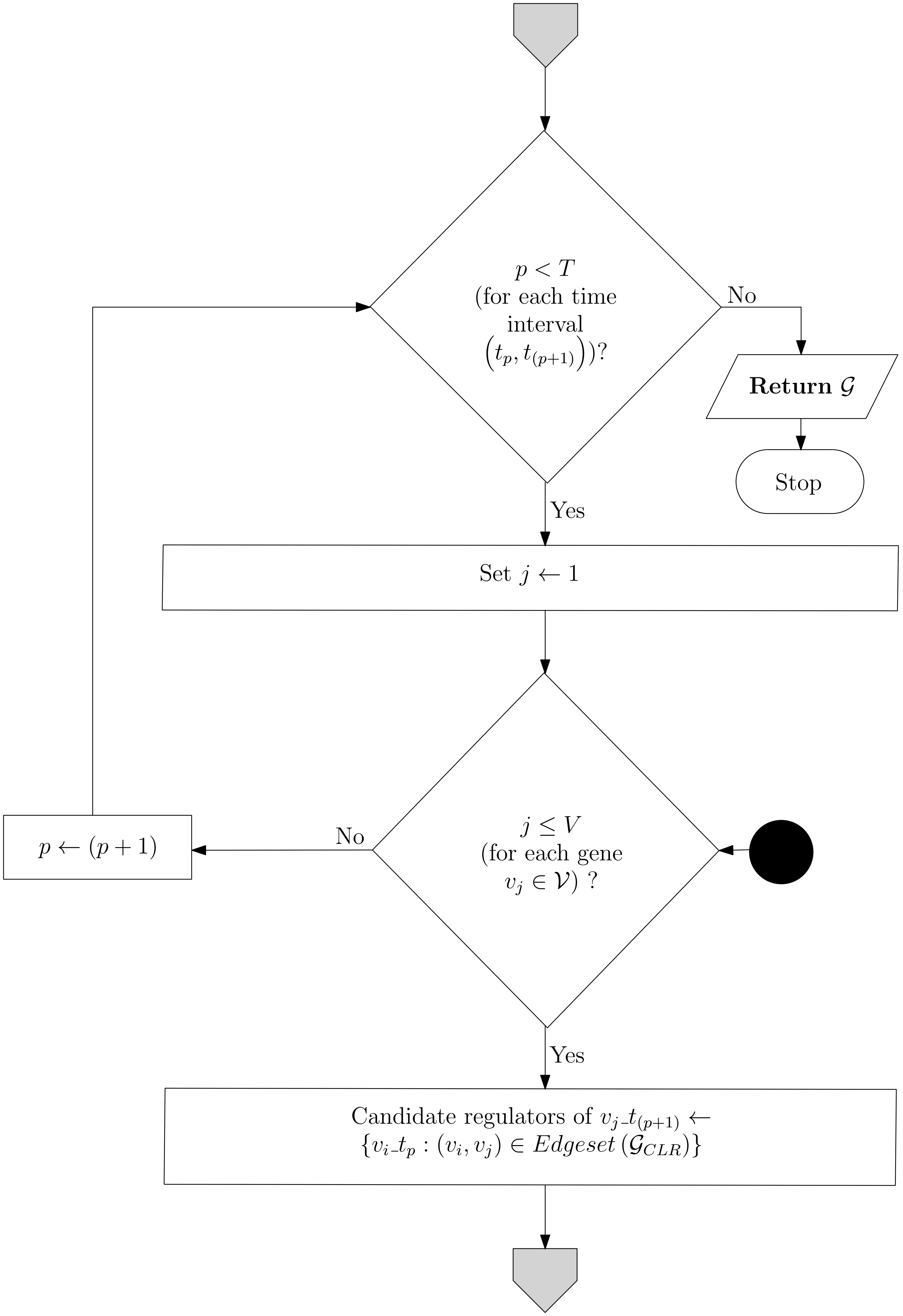

Figure 2.2: Flowchart of the *TGS* algorithm (Algorithm 3, main paper). The flowchart is continued from Figure 2.1 and further continued to Figure 2.3.

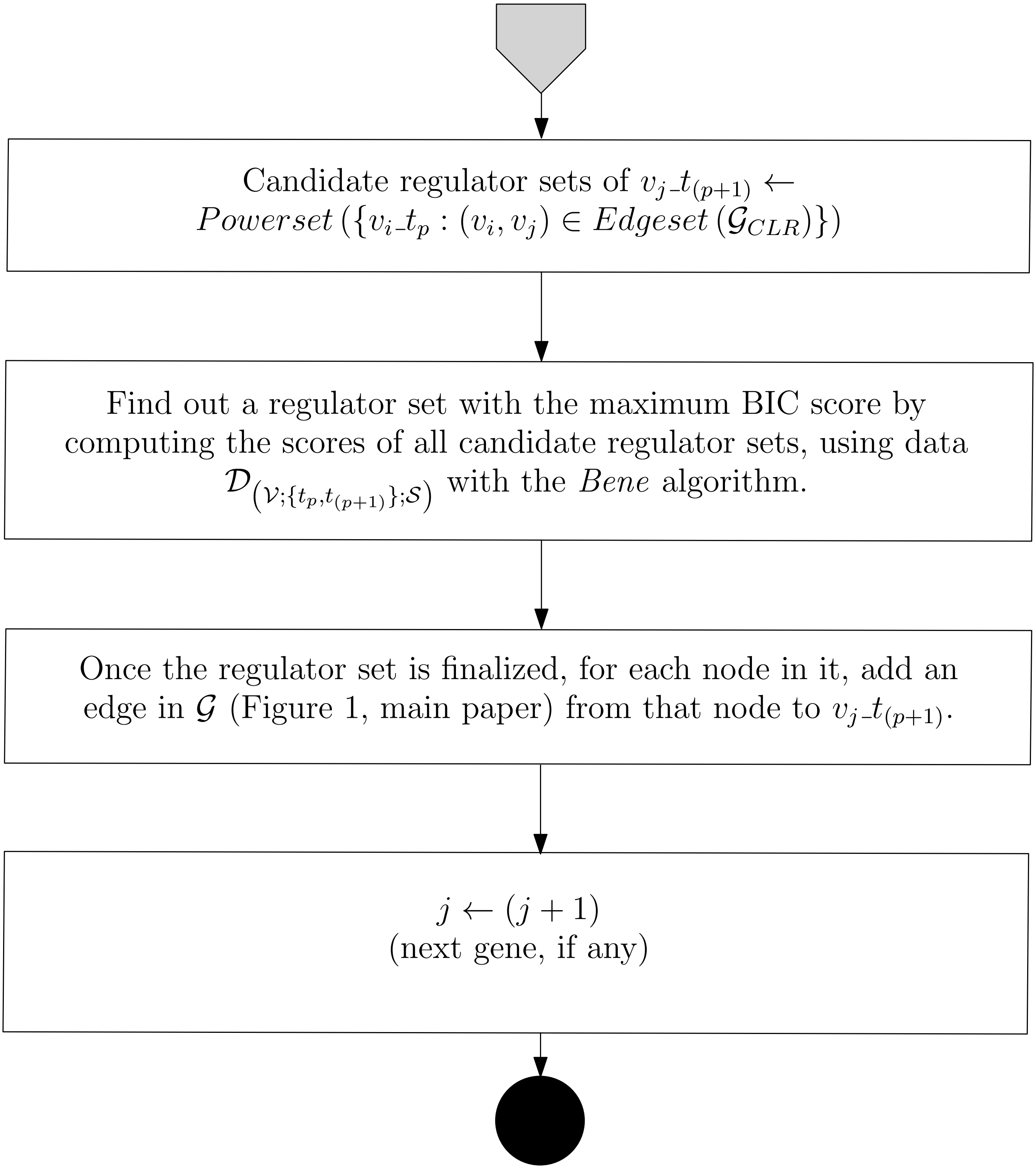
Candidate regulator sets of $v_j\_t_{(p+1)} \leftarrow$
$Powerset\left(\{v_i\_t_p : (v_i, v_j) \in Edgeset\left(\mathcal{G}_{CLR}\right)\}\right)$
Find out a regulator set with the maximum BIC score by computing the scores of all candidate regulator sets, using data $\mathcal{D}_{\left(\mathcal{V};\{t_p,t_{(p+1)}\};\mathcal{S}\right)}$ with the *Bene* algorithm.
Once the regulator set is finalized, for each node in it, add an edge in $\mathcal{G}$ (Figure 1, main paper) from that node to $v_j\_t_{(p+1)}$.
$j \leftarrow (j+1)$
(next gene, if any)

Figure 2.3: Flowchart of the *TGS* algorithm (Algorithm 3, main paper). The flowchart is continued from Figure 2.2.

Chapter 3

Executing Learn.TGS function

3.1 Descriptions of Parameters as inputs

input.data.filename The ‘input.data.filename’ parameter can have filenames with either the ‘.tsv’ or the ‘.RData’ extension. If the file has the ‘.tsv’ extension, then the first column should contain the time point IDs except the $(1,1)^{th}$ cell. The first row should contain the gene names, except the $(1,1)^{th}$ cell. The $(1,1)^{th}$ cell does not carry any meaning. If the file has the ‘.RData’ extension, then the underlying object must have the name ‘input.data’. In ‘input.data’, the column names and the row names represent the gene names and the time point IDs, respectively. For either of ‘.tsv’ or ‘.RData’ input, Multiple rows with the same time point ID represent multiple replicates at the same time point. In other words, those rows belong to the same time point but at different time series. The time points belonging to the same time series must be together and in ascending order. An exemplary dataset with three genes {G1, G2, G3}, two time points {t1, t2} and two time series is shown below.

```
Time  G1           G2           G3
t1    0.8272480342 0.7257430901 0.3894130418
t2    0.6542518342 0.6470658823 0.5088904888
t1    0.3519554463 0.3551279726 0.3207993604
t2    0.4871730974 0.3706990326 0.447523615
```

num.timepts The ‘num.timepts’ parameter defines the number of distinct time points in the input data file.

true.net.filename If ‘true.net.filename’ is an empty string, then it is implied that the true rolled network is not known a prior. But if it is not an empty string, then it must be a ‘.RData’ file. In that case, the underlying object must be of name ‘true.net.adj.matrix’. In ‘true.net.adj.matrix’, the row names and the column names represent the gene names. It is a binary matrix. If $(i,j)^{th}$ cell contains 1, then there exists a directed edge from the i^{th} gene to the j^{th} gene in the true network. Else if $(i,j)^{th}$ cell contains 0, then that edge does not exist in the true network.

input.wt.data.filename The ‘input.wt.data.filename’ parameter provides the name of the file which has the Wild Type (WT) values of the genes. If its value is an empty string, then the WT values of the genes are not known. Otherwise, it must be a file with ‘.tsv’ extension. Inside the file, the first row should contain the gene names, except the $(1,1)^{th}$ cell. Then the second row should contain the WT values of those genes, except the $(2,1)^{th}$ cell. Rest of the file does not carry any meaning.

is.discrete The ‘is.discrete’ parameter takes value ‘true’ or ‘false’, depending on whether the input data is already discretized or not.

num.discr.levels The 'num.discr.levels' parameter provides the number of discrete levels each gene has if the input dataset is already discretized ("is.discrete": true). On the other hand, 'num.discr.levels' provides the number of discrete levels into which each gene needs to be discretized if the input dataset is not yet discretized ("is.discrete": false).

discr.algo The 'discr.algo' parameter is used to specify the name of the discretization algorithm to be used in case the input data needs to be discretized. For the time being, only two options are available: 'discretizeData.2L.wt.l' and 'discretizeData.2L.Tesla', which represent algorithms *2L.wt* and *2L.Tesla*, respectively.

mi.estimator The 'mi.estimator' parameter specifies which method to use for estimating the mutual information matrix. For all the experiments, the 'mi.pca.cmi' estimator is used. The original source code of 'mi.pca.cmi' is written in MATLAB and available as function 'cmi()' at `http://www.comp-sysbio.org/grn/pca_cmi.m`.

apply.aracne 'apply.aracne' is a boolean parameter. *ARACNE* is applied to refine the mutual information matrix only if this parameter is set to 'true'. Therefore, if 'apply.aracne' is true, then the *TGS+* variant is executed. Otherwise, the original *TGS* variant is executed.

clr.algo This parameter indicates which *CLR* variant to employ. Currently, only one variant is available, namely 'CLR'.

max.fanin The 'max.fanin' parameter provides the maximum number of regulators each gene can have.

allow.self.loop The 'allow.self.loop' parameter takes value 'true' or 'false', depending on whether to allow self loops in the predicted rolled network or not.

3.2 Description of *TGS* Output Files

output.txt This file saves the console output generated by 'TGS.R' and its callee R scripts. Please open it in a text editor. At the very beginning, there is a line stating 'elapsed.time just after CLR step=' followed by the time taken by the *CLR* step in seconds. For example:

```
elapsed.time just after CLR step=

user  system elapsed
0.000   0.000   0.003
```

In this case, the 'elapsed' time is considered as the completion time for the *CLR* step, which is 0.003 seconds.

At the very end of the file, there is a line stating 'elapsed.time =' followed by the time taken by the whole *TGS* algorithm in seconds. For example:

```
elapsed.time =

user  system elapsed
5.684   0.100   5.789
```

In this case, the 'elapsed' time is considered as the runtime of the *TGS* algorithm, which is 5.789 seconds.

Only for the experiments where the true rolled network is provided as an input for evaluating the learning power of *TGS*, there are four lines just before the 'elapsed.time =' line. Among these four lines, the first line states 'Result TGS vs True =' and the second line is a blank line; finally, the last two lines depict the evaluation metrics of *TGS*. For example:

```
Result TGS vs True =

     TP TN FP FN TPR       FPR       FDR       PPV  ACC       MCC         F
[1,]  3 80 10  7 0.3 0.1111111 0.7692308 0.2307692 0.83 0.1684986 0.2608696
```

In this case, the metrics are read as ‘TP = 3’, ‘TN = 80’ and so on.

The rest of the lines in the file are for debugging purposes only.

di.net.adj.matrix.RData and net.sif File ‘di.net.adj.matrix.RData’ contains the predicted rolled network’s adjacency matrix. The matrix can be analysed by loading the file in a R session:

```
## Load object 'di.net.adj.matrix'
> load('di.net.adj.matrix.RData')
```

The row names and the column names correspond to the input gene names. The $(i, j)^{th}$ cell takes value 1 if and only if there is a directed edge from the i^{th} gene to the j^{th} gene in the predicted network; otherwise, it takes value 0.

‘net.sif’ is the Cytoscape compatible SIF format of ‘di.net.adj.matrix.RData’.

unrolled.DBN.adj.matrix.list.RData This file contains the unrolled time-varying Gene Regulatory Networks. The networks can be analysed by loading the file in a R session:

```
## Load object 'unrolled.DBN.adj.matrix.list'
> load('unrolled.DBN.adj.matrix.list.RData')
```

The object ‘unrolled.DBN.adj.matrix.list’ is a list of $(T - 1)$ matrices, where T denotes the total number of time points in the input dataset. The t^{th} element of the list is the adjacency matrix of the network that represents the predicted gene regulations during the time interval between time points t and $(t + 1)$. For example, when $T = 21$:

```
## Length of the list = (T - 1)
> length(unrolled.DBN.adj.matrix.list)
[1] 20

## Print the first network i.e.
## predicted gene regulations during
## the time interval between the 1st
## and the 2nd time points
> unrolled.DBN.adj.matrix.list[[1]]
    v1 v2 v3 v4 v5 v6 v7 v8 v9 v10
v1   0  0  0  0  0  0  0  0  0   0
v2   1  0  0  1  0  0  0  0  0   0
v3   0  0  0  0  0  0  0  0  0   0
v4   0  1  0  0  0  0  0  0  0   0
v5   0  0  0  0  0  0  0  0  0   0
v6   0  0  0  0  0  0  0  0  0   0
v7   0  0  0  0  0  0  0  0  0   0
v8   0  0  0  0  0  0  0  0  0   0
v9   0  0  0  0  0  0  0  0  0   0
v10  0  0  0  0  0  0  0  0  0   0
```

It needs to be noted that each original gene name is replaced with ‘v’ followed by its index. For example, the original gene names in dataset Ds10n are {G1, G2, ..., G10} in the given order. Therefore, they are replaced with {v1, v2, ..., v10}. Such name replacement strategy is performed for all intermediate outputs (i.e. all outputs but the final output ‘di.net.adj.matrix.RData’) to avoid unexpected symbols in the original gene names that might cause errors. If required, the original names can be restored with the help of ‘di.net.adj.matrix.RData’:

```
## Load object 'di.net.adj.matrix'
> load('di.net.adj.matrix.RData')

## List objects in the current workspace
> ls()
[1] "di.net.adj.matrix"            "unrolled.DBN.adj.matrix.list"

## Get the original gene names
> orig.names <- colnames(di.net.adj.matrix)
> orig.names
[1] G1"  "G2"  "G3"  "G4"  "G5"  "G6"  "G7"  "G8"  "G9"  "G10"

## Restore the original gene names
> rownames(unrolled.DBN.adj.matrix.list[[1]]) <- orig.names
> colnames(unrolled.DBN.adj.matrix.list[[1]]) <- orig.names

## Print the first network
> unrolled.DBN.adj.matrix.list[[1]]
    G1 G2 G3 G4 G5 G6 G7 G8 G9 G10
G1   0  0  0  0  0  0  0  0  0   0
G2   1  0  0  1  0  0  0  0  0   0
G3   0  0  0  0  0  0  0  0  0   0
G4   0  1  0  0  0  0  0  0  0   0
G5   0  0  0  0  0  0  0  0  0   0
G6   0  0  0  0  0  0  0  0  0   0
G7   0  0  0  0  0  0  0  0  0   0
G8   0  0  0  0  0  0  0  0  0   0
G9   0  0  0  0  0  0  0  0  0   0
G10  0  0  0  0  0  0  0  0  0   0
```

In each predicted time-varying network, the $(i, j)^{th}$ cell takes value 1 if and only if the i^{th} gene regulates the j^{th} gene during that time interval. The predicted rolled network is the union of all the predicted time-varying networks; in other words, the $(i, j)^{th}$ cell in the rolled network adjacency matrix takes value 1 if and only if there exists at least one time-varying network whose adjacency matrix's $(i, j)^{th}$ cell contains value 1, otherwise the former takes value 0.

mut.info.matrix.RData, mut.info.matrix.pre.aracne.RData, mut.info.matrix.post.aracne.RData, mi.net.adj.matrix.wt.RData and mi.net.adj.matrix.RData These files save intermediate outputs during the *CLR* step. When 'apply.aracne' is set to false, the raw mutual information matrix is saved in 'mut.info.matrix.RData'. When 'apply.aracne' is set to true, the raw mutual information matrix is saved in 'mut.info.matrix.pre.aracne.RData'. In the latter case, the raw mutual information matrix is then refined by passing it through *ARACNE*. The refined mutual information matrix is saved in 'mut.info.matrix.post.aracne.RData'.

CLR takes the raw mutual information matrix (when 'apply.aracne' is false) or the refined mutual information matrix (when 'apply.aracne' is true) as input and produces a weighted *CLR* network adjacency matrix 'mi.net.adj.matrix.wt' (stored in 'mi.net.adj.matrix.wt.RData') where the $(i, j)^{th}$ cell contains a non-zero value if and only if an undirected edge exists between the i^{th} and the j^{th} genes; otherwise, it contains value 0. This matrix is used to compute an unweighted undirected network adjecncy matrix 'mi.net.adj.matrix' (stored in 'mi.net.adj.matrix.RData') by retaining only the top 'max.fanin' (see Paragraph 'max.fanin', Section 3.1) number of neighbours for each gene based on the edge weights. In this matrix, the $(i, j)^{th}$ cell contains value 1 if and only if an undirected edge exists between the i^{th} and the j^{th} genes; otherwise, it contains value 0.

input.data.discr.RData This file is generated only if the input dataset is not discretized already. In that case, this file represents the input dataset after discretization with the user-defined discretization settings. Please load this file in a R session to analyze the undelying R object ‘input.data.discr’:

```
## Load a R object named 'input.data.discr'
> load('input.data.discr.RData')
```

‘input.data.discr’ is a data matrix with the $(i, j)^{th}$ cell representing the discretized expression level of the j^{th} gene in the i^{th} observation of the original input dataset.

sessionInfo.txt This file provides the R session information during a specific experiment, as generated by the R function ‘sessionInfo()’.