Supplementary Material: Inferring Boolean Networks from Single-Cell Human Embryo Datasets

Mathieu Bolteau, Jérémie Bourdon, Laurent David and Carito Guziolowski

Contents

1 Supplementary Method

1.1 Developed pipeline

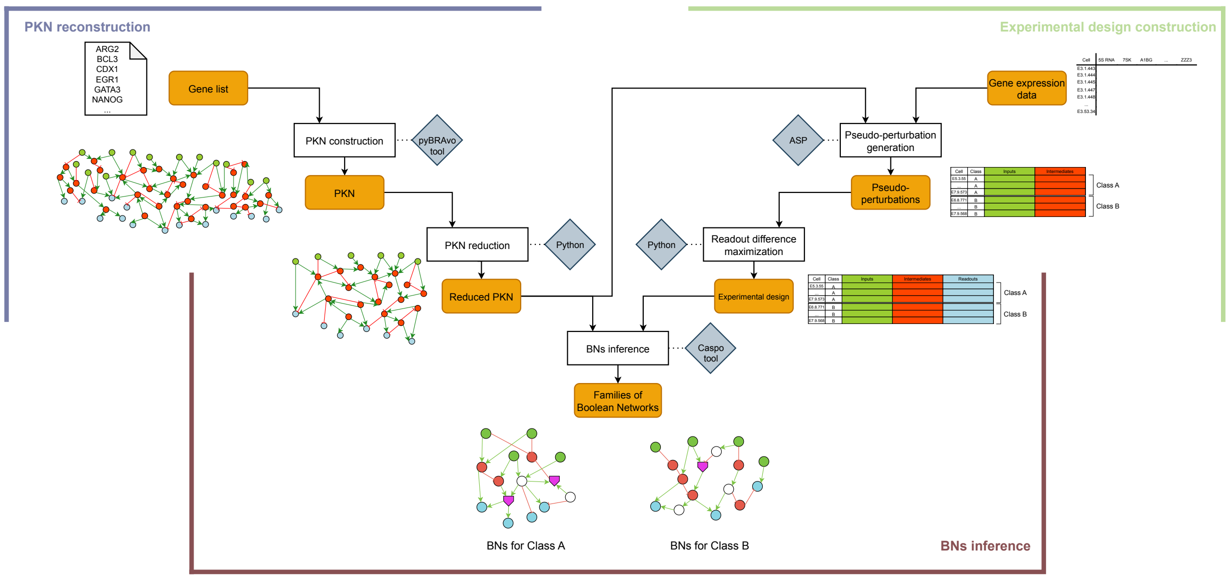

Figure 1: Developed pipeline.

In Figure 1, we present the different steps of our implemented pipeline. The pipeline consists of three main steps: (*i*) the PKN reconstruction, (*ii*) the experimental design construction, and (*iii*) the BNs inference.

1.1.1 PKN reconstruction

We construct a PKN using pyBRAvo [4] from a list of genes. We reduce this PKN according to the scRNAseq data.

1.1.2 Experimental design construction

Given gene expression data of cells belonging to two classes, an ASP program calculates pseudo-perturbations for selected genes and cells (see Section 2.3, *Maximizing the number of pseudo-perturbations*, in the main paper). We use pseudo-perturbations to maximize the readout differences (see Section 2.3 , *Maximizing readout difference*, in the main paper); the output of this process is the optimal experimental design. We present an example of an experimental design in Table 1.

Table 1: Example of an experimental design.

cell	class	pseudo-perturbations				readouts		
		g_1	g_2	g_3	g_4	g_5	g_6	g_7
c_1	A	1	1	0	0	0.15	0.45	0.6
c_2	A	0	1	1	0	0.85	0.6	0.1
c_3	A	1	0	0	1	0	0.7	0.85
c_4	B	1	1	0	0	0.25	0.65	0.9
c_5	B	0	1	1	0	0.3	0.95	0.45
c_6	B	1	0	0	1	0.2	0.5	0.5

Genes $g_1 - g_4$, named pseudo-perturbation genes, are input and intermediate genes. Each binarized pseudo-perturbation vector for a specific cell in class A is identical to the one of another cell belonging to class B. Note that the normalized values of readouts are different between the two classes.

1.1.3 BNs inference

Lastly, Caspo [6] is used to infer, given a PKN and an experimental design, specific BNs to each class. Each BN in the speficic families is both compatible with the PKN topology and the gene expression present in the experimental design.

1.2 ASP introduction

Answer Set Programming (ASP) [2] is a declarative programming paradigm used to specify knowledge bases in the form of rules. Programs are built from atoms, which consist of a predicate with arguments that are terms. The simplest types of terms are integers, constants, and variables. Constants and variables are distinguished by their first letter, respectively, a lower case and an upper case. Atoms are elementary constructions for representing knowledge and are used to make simple verbal assertions.

Rules in ASP consist of a *head* and a *body*, where the head, composed of a set of atoms, specifies the goal or conclusion to be derived, and the body specifies the conditions under which the conclusion is true. *Literals*, which can be positive or negative, represent propositions in the body of a rule.

The rules of an ASP program are as follows:

- Fact: H.
- Rule: $H \leftarrow L_1, ..., L_n$.
- Integrity constraint: $\leftarrow L_1, ..., L_n$.

where $H, L_1, ..., L_n$ are literals, part of the head and the body. Each L_i is a literal of the form `A` or `not A`, where `A` is an atom and the logical connector `not` is the default negation. We say that a literal L is positive if it is an atom (`A`) and negative otherwise (`not A`).

Facts in ASP are rules with an empty body and only a head. They are principally used to represent knowledge. Integrity constraints are used in ASP to specify restrictions on the possible interpretations of a knowledge base, ensuring that only valid models are considered.

We introduce a small logic program example, illustrating concepts seen below (see Listing 1).

```
cell(c1). cell(c2). cell(c3).
class(early_TE). class(medium_TE). class(late_TE).
be_part(c1,early_TE). be_part(c2,medium_TE).   be_part(c3,late_TE).
gene(g1). gene(g2).
expr(c1,g1,0). expr(c1,g2,0).
expr(c2,g1,0). expr(c2,g2,1).
expr(c3,g1,1). expr(c3,g2,1).
pert(C,G,S,CL) :- expr(C,G,S), cell(C), gene(G), be_part(C,CL).
{sel_pert(C,G,S,CL) : pert(C,G,S,CL)}.
```

```
10      :- sel_pert(_,_,_,early_TE).
11      #show sel_pert/4.
```

Listing 1: Example of a logic program.

This logic program defines knowledge via the facts on lines 1-7. For example, the predicate `expr(c1,g1,0)`, line 4, states that in cell `c1`, the gene `g1` is expressed with a value `0`. In line 8, a rule defines a predicate `pert/4` (of arity 4 and composed of 4 terms), modeling the experimental data, in which gene G is expressed at a value S in cell C, which belongs to class CL. In line 9, the program selects a subset of predicates `pert/4` using the predicates `sel_pert/4`, representing the selected perturbations. This type of construct is called *choice rules* and is central in ASP modeling since it generates the possible combinations of candidate solutions. Candidate solutions are usually then filtered using constraints. For example, in line 10, the program forbids the solution candidate `sel_pert/4` associated with the class *early_TE*. Finally, in line 11, the program shows the answer to this program, focusing only on the predicate `sel_pert/4`. This answer is given by a set of answers (answer sets) of assignments of constants to the terms of predicate `sel_pert/4`; each assignment verifies all program rules.

In conclusion, ASP provides a powerful way of specifying complex rules and constraints, making it a valuable tool for solving a wide range of problems in areas such as artificial intelligence, natural language processing, and planning.

1.3 Pseudo-perturbations maximization program

In this section, we explain line by line the program, implemented in ASP, used to maximize the number of pseudo-perturbations, which we expose the algorithm in our paper (see Section 2.3, *Maximizing the number of pseudo-perturbations*). First, this program is based on the method proposed in [3], but mainly differs in the rule for generating different Boolean pseudo-perturbation vectors imposed in our program. Our logic program is specific to scRNAseq data, which is expected to be redundant as some cells in the same developmental stage have the same gene expression. Another specificity of scRNAseq data is the strong abundance of zero values.

In Listing 2, we present the ASP encoding.

```
1       {selgene(G):pert(C,G,S,CL)} = k.
2       selpert(C,G,S,CL) :- selgene(G), pert(C,G,S,CL).
3       equal(I,J,G) :- selpert(I,G,S1,C1), selpert(J,G,S2,C2), C1<C2, S1 = S2.
4       countequal(I,J,M) :- M={equal(I,J,_)}.
5       0{affinity(I,J)}1 :- countequal(I,J,k).
6       nbInputOnes(C, N) :- N={pert(C,G,1,_) : selinput(G)}, affinity(C,_).
7       :- affinity(C,_), nbInputOnes(C,N), N<1.
8       diff(I1,I2,G) :- selpert(I1,G,S1,C1), selpert(I2,G,S2,C2), C1==C2, S1!=S2,
            I1<I2.
9       countdiff(I1,I2,M) :- M={diff(I1,I2,_)}.
10      :- countdiff(I1,I2,0), affinity(I1,_), affinity(I2,_), I1<I2.
11      :- countdiff(I1,I2,0), affinity(_,I1), affinity(_,I2), I1<I2.
12      #maximize{1,I: affinity(I,_)}
```

Listing 2: ASP encoding of pseudo-perturbation generation

Predicate `pert/4` is the instance of our program, referring to the experimental data. It is derived from the discretization of scRNAseq data related to input and intermediate genes. It states that gene G is expressed at a value S in cell C, which belongs to class CL. Our logic program begins (line 1) with selecting a set of k genes from all possible (input and intermediate) genes using the `selgene/1` predicate. This step generates $\binom{m}{k}$ answer sets, where m is the total number of input and intermediate genes. The following rules are meant to filter these candidate answer sets. In line 2, the `selpert/4` predicate summarizes the experimental data for the selected genes. Then, in line 3, using the `equal(I,J,G)` predicate, we select pairs of cells I and J that belong to different classes (`C1<C2`), in which gene G is measured with the same value S (`S1=S2`). In line 4, the `countequal(I,J,M)` predicate counts the number of genes M for which their values are equal across cells I and J. Recall that we are interested in finding k identical values associations for k genes. Therefore, in line 5, we define the predicate `affinity(I,J)`, which will be generated 0 or 1 times when there are k similarities for cells I and J. This predicate will identify the candidate optimal pseudo-perturbations. Note that the left and right terms of predicate `affinity/2`, defined in this case by variables I and J, refer to cells in the first, respectively, second class.

To address data sparsity, we added the rules in lines 6-7. Line 6 defines `nbInputOnes/2`, which counts the number of input genes whose value is `1` for a cell C that has been selected by the predicate `affinity/2`. Then, in line 7, we forbid selecting an `affinity(C,_)` if the number of 1-signed input genes in cell C is less than 1 (`N<1`). These rules ensure that at least one input gene is expressed to 1 for each pseudo-perturbation. This also implies that vectors with all genes inactive (equal to 0) are not allowed.

To identify distinct pseudo-perturbations, we introduced new rules (lines 8-11). Line 8 defines `diff(I1,I2,G)` predicate, which selects cells $I1$ and $I2$, from the same class, with different values for gene G. Then, in line 9, a predicate `countdiff/3` stores the differences in the selected genes' expression values for cells $I1$ and $I2$. In line 10 (resp. line 11), the constraint forbids predicates `countdiff(I1,I2,0)`, where there is no difference in expression values for the selected genes, for cells $I1$ and $I2$ selected to be affinities in line 5 for the first class (resp. for the second class). Combined lines 5, 10, and 11 keep only one cell association when the same cell is associated with other cells, such as retaining only `affinity(c1,c2)` and not `affinity(c1,c3)` from possible associations.

Finally, in line 12, we search to maximize the associations given by predicate `affinity/2` concerning the first class, left term.

After obtaining the optimal pseudo-perturbations, we use Python to find the readout values, which maximize the expression difference in both classes. Altogether, the optimal pseudo-perturbations associated with readouts having a maximal difference between classes constitute the experimental design for cells in classes A and B which will be the input to Caspo [6] for inferring BNs specific to each class.

To sum up, our approach involves augmenting the initial logic program proposed by Chebouba *et al.* [3] with various constraints and rules, thereby imposing further restrictions on the problem and facilitating its solution for the solvers. Note that our proposed approach addresses a more constrained problem, removing redundant pseudo-perturbations retained in Chebouba *et al.*'s version.

2 Supplementary Results

2.1 Used parameters for method application on medium and late TE stage discrimination

Here, we provide more information about the parameters used for each step of the method to discriminate the medium and late trophectoderm stages in our paper.

2.1.1 PKN reconstruction

We used 438 transcription factor (TF) genes involved in human embryonic development as input for pyBRAvo software [4] to reconstruct a PKN. These TF genes were identified through SCENIC [1] analysis of scRNAseq data, and their list can be found on the GitHub repository. Queries were made on Pathway Commons v.13 [5], excluding miRTarBase, MSigDB, and CTD databases to remove miRNA and toxicogenomics interactions. The exploration depth parameter was fixed to 2, *i.e.* up to 2 levels upstream of the initial TFs. Only gene transcription events were queried, yielding a PKN of 327 nodes and 475 edges, with only 28 of the 438 initial TFs found in the database (see supplementary material on our GitHub repository[1]). We then reduced the network to 191 nodes (84 input genes, 27 intermediate genes, 14 readout genes, and 66 complexes) and 285 edges (Figure 2), limited to genes measured in scRNAseq data and complexes linked to these genes. Green nodes are input genes, red nodes are intermediate genes, blue nodes are readout genes and gray nodes are protein complexes. Green arrows mean activation, red arrows mean inhibition and black arrows mean part of the complex. The PKN is also available on our GitHub.

2.1.2 Experimental design construction

To generate the pseudo-perturbations, we fix the k value to 10. We ran the program on a computer cluster comprising 160 CPUs and 1.5 To of RAM for 65 hours to obtain 20 pseudo-perturbations.

2.1.3 BNs inference

We used the generated experimental design combined with the reduced PKN to infer BNs specific to medium and late TE using the Caspo software. Caspo proposes BNs that match the PKN topology and have an optimal (minimal) mean square error (MSE) between the Boolean prediction of readout nodes (given the Boolean input states) and their experimental measurement.

[1]https://github.com/mathieubolteau/scRNA2BoNI/tree/master/ ISBRA_2023_Supp

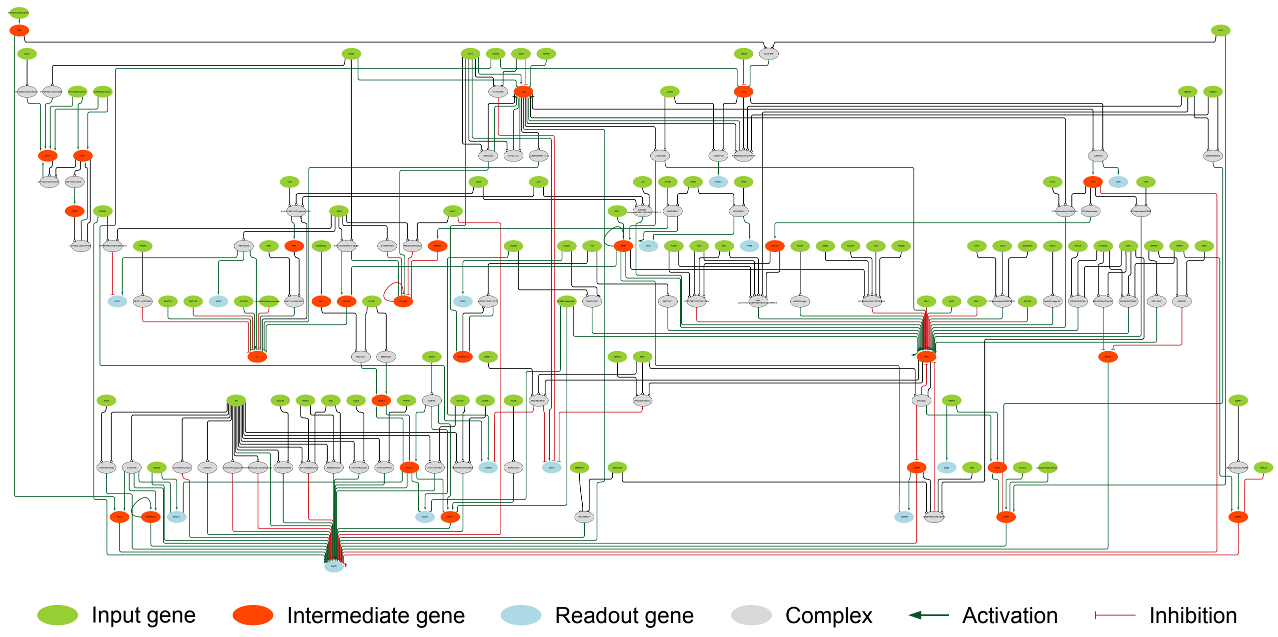

Figure 2: Reconstructed PKN.

To restrict the number of Boolean functions of type "AND" that a node can receive to 2, we set (i) $length = 2$, and to fix the optimal MSE fitness tolerance, we set (ii) $fit = 0.001$. Essentially, larger fitness values result in further exploration from the optimal BN.

2.2 Comparing pseudo-perturbation generation programs

Our pseudo-perturbations identification algorithm, inspired from [3], proposes an additional constraint (see Section 2.3, *Maximizing the number of pseudo-perturbations – Problem formulation*, constraint 2, in the main paper), which ensures that different pseudo-perturbations of k expressed genes are found within the same class. While increasing computational time, it proves valuable in handling redundant scRNAseq data. We apply both programs to datasets $A - P$ (see Table 2). For comparison purposes, we post-processed the results of [3] (time not shown) by removing redundant solutions (values in parenthesis in Table 2).

Both programs achieved optimal solutions for datasets $A - B$ (see Table 2), while suboptimal results were obtained for datasets $C - P$ with fixed timeouts (identified with a *). For more details about the dataset specifities, we refer the reader to Table 1 in the main paper. Chebouba *et al.*'s version [3] (C in the Table) exhibited shorter execution times compared to our version (O in the Table) when no timeout was fixed. Regarding the number of different pseudo-perturbations generated by each program, we observe two behaviors depending on the nature and complexity of the dataset. First, for single-cell datasets ($A - D$), Chebouba's program computes an equal or smaller number of pseudo-perturbations after redundancies post-processing. For example, for dataset C, Chebouba's program computes an optimal answer comprising only 1 distinct pseudo-perturbation, while our program produces sub-optimal answers comprising more distinct pseudo-perturbations. Moreover, we observe a larger difference for more complex datasets such as D or C with $k = 10$. From these results, it can be observed that Chebouba's program infers several redundant solutions without allowing us to distinguish them. This highlights our program's better capacity to handle scRNAseq data than Chebouba's. Second, our version produces comparable or better results for phosphoproteomics data (dataset P), confirming its applicability to single-cell or averaged cell population gene-expression datasets.

Table 2: Comparison of logic programs on different datasets.

Dataset	k	Execution time		Distinct Pseudo-Perturbations	
		C	O	C	O
A	3	0.008s	0.008s	3 (4)	3
B	3	0.048s	0.223s	1 (132)	4
C	3	1.420s	10 min*	1 (625)	6
	10	1.424s	10 min*	1 (600)	11
D	10	10 min*	10 min*	10 (2,436)	22
P	10	50h*	50h*	23 (64)	25
SC	10	5h 2min	65h*	3 (77,618)	20

C corresponds to the Chebouba *et al.*'s logic program, while O corresponds to our logic program. For Chebouba *et al.*'s program, in parenthesis, the total number of pseudo-perturbations vectors (redundancy comprising). * Execution time corresponds to the fixed time-out.

References

[1] Aibar, S., González-Blas, C.B., Moerman, T., Huynh-Thu, V.A., Imrichova, H., Hulselmans, G., Rambow, F., Marine, J.C., Geurts, P., Aerts, J., et al.: Scenic: single-cell regulatory network inference and clustering. Nature methods **14**(11), 1083–1086 (2017)

[2] Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, New York, NY, USA (2003)

[3] Chebouba, L., Miannay, B., Boughaci, D., Guziolowski, C.: Discriminate the response of acute myeloid leukemia patients to treatment by using proteomics data and answer set programming. BMC Bioinformatics **19**(2), 15–26 (2018)

[4] Lefebvre, M., Gaignard, A., Folschette, M., Bourdon, J., Guziolowski, C.: Large-scale regulatory and signaling network assembly through linked open data. Database **2021** (2021)

[5] Rodchenkov, I., Babur, O., Luna, A., Aksoy, B.A., Wong, J.V., Fong, D., Franz, M., Siper, M.C., Cheung, M., Wrana, M., Mistry, H., Mosier, L., Dlin, J., Wen, Q., O'Callaghan, C., Li, W., Elder, G., Smith, P.T., Dallago, C., Cerami, E., Gross, B., Dogrusoz, U., Demir, E., Bader, G.D., Sander, C.: Pathway Commons 2019 Update: integration, analysis and exploration of pathway data. Nucleic Acids Research **48**(D1), D489–D497 (10 2019)

[6] Videla, S., Saez-Rodriguez, J., Guziolowski, C., Siegel, A., Wren, J.: caspo: a toolbox for automated reasoning on the response of logical signaling networks families. Bioinformatics **33**(6), 947–950 (2017)