RAI Algorithm – Matlab Implementation

References:

1. Yehezkel, R. and Lerner, B. (2009). "Bayesian network structure learning by recursive autonomy identification." Journal of Machine Learning Research, 10, 1527-1570.
2. Yehezkel, R. and Lerner, B. (2005). "Recursive autonomy identification for Bayesian network structure learning." 10th International Workshop on Artificial Intelligence & Statistics, AISTATS 2005, eds. R. G. Cowell and Z. Ghahramani, Barbados, pages 429-436.

Function Call

Example:

```
[pdag_rai, seps, Nci_rai] = ...                 % Outputs #1, #2 and #3
        learn_struct_pdag_rai(...
                setdiag(ones(n_nodes), 0), ...  %  Argument #1
                [], ...                         %  Argument #2
                (1:n_nodes), ...                %  Argument #3
                cell(n_nodes), ...              %  Argument #4
                0, ...                          %  Argument #5
                'mutualC_f_e', ...              %  Argument #6
                training_data, ...              %  Argument #7
                CI_Threshold);                  %  Argument #8
```

Outputs

Output #1 – *pdag_rai*

This is the learned PDAG.

For a perfect CI testing, a CB structure-learning algorithm (e.g., PC or RAI) will provide a completed partially directed acyclic graph (CPDAG) that is the most we can learn using a CB algorithm. However, when CI testing is not perfect (e.g., due to finite sample), the resultant learnt graph is not a CPDAG but a PDAG, as it may contain cycles (Kalisch and Buhlmann, 2007). The user can use the following post-processing procedure for ensuring a CPDAG result.

```
cpdag_rai = remove_cycles_data(pdag_rai, training_data); % removes edges from cycles
cpdag_rai = dag_to_cpdag(cpdag_to_dag(cpdag_rai));       % assures a CPDAG result
```

Note:
The post-processing procedure removes edges from cycles using heuristic rules. Use this procedure only in situations where a CPDAG result is necessary. For example, in classification, if the cycles do not affect the Markov blanket (MB) of the class variable, the post processing is not required (we do not care if there are cycles that do not affect the class variable's MB). Also, whenever we are interested in inference on a specific variable or part of the graph, which are not involved with cycles.

Output #2 – *seps*

A cell matrix containing the d-separation sets found during the learning process. Row and column indexes correspond to the variables.

Output #3 - *Nci_rai*

An array containing the number of CI tests performed sorted according to the order of the CI-test. *Nci_rai*(1) contains the number of CI tests with empty condition set (order 0), *Nci_rai*(2) contains the number of CI tests with a single variable in the condition set (order 1) etc.

Note that this array has the length of [#Nodes-1] elements.

Inputs

Argument #1 – Initial Graph

In common initial conditions, the initial graph should be a fully connected graph, set as:

```
Initial_Graph = setdiag(ones(n_nodes), 0);
```

Argument #2 – Exogenous Set of Nodes

In common use, this is an empty set ("[ ]") (e.g., for an initial graph, which is complete or expert-derived). In internal recursive calls to the algorithm, this set includes the ancestor nodes of the sub-structure the RAI currently learns.

Argument #3 – Set of Nodes for Learning

This is the set of nodes for which the graph should be learned. Note that the exogenous nodes given at argument #2 should not be members of this set and vice versa. In common use (e.g., for an initial complete graph), this set includes all the nodes. In internal recursive calls, this set includes the specific nodes that compose the current descendent sub-structure that is learned.

Argument #4 – Initial Matrix of d-Separation Sets

In common initial conditions (for a complete graph), this input should be a matrix of empty cells:

```
Initial_DSepSets = cell(n_nodes);
```

When learning for orders larger than 0 and the initial graph is not fully connected, the d-separation sets corresponding to the missing edges should be supplied here.

Argument #5 – Initial CI test order

In common initial conditions, this input should be 0. Otherwise, the lowest order of CI tests conducted by RAI should be stated here. Note that there is a limitation of a maximal CI test order of 10 in function 'learn_struct_rai' that is called by 'learn_struct_pdag_rai'. In learning large networks, this limitation may be lifted.

Argument #6 – CI Test

This is a string containing the name of the function that performs the CI test (e.g., 'mutualC_f_e' for a CI test based on conditional mutual information, which is the test commonly performed with RAI in the above references). The function performing the CI test should be able to handle five arguments (for more information type "*doc mutualC_f_e*" in the Matlab command line):

- Arguments 1 & 2: The nodes that are being tested for independence.
- Argument 3: The condition set.
- Argument 4: Data samples.
- Argument 5: CI test threshold.

Argument #7 – Training Data

The training data set where each column corresponds to a variable and each row to a data sample. This data should include only categorical or discrete variables where the values of each variable ranges in [1..Max(Var)] (contrary to 0-based indexing).

Argument #8 – CI Test Threshold

A threshold value (Argument #5) that is provided to the CI test function (Argument #6).

Here is a suggested heuristic method for selecting the threshold of mutual-information-based CI tests:

1. Calculate the mutual information (MI) between each pair of nodes.
2. Create a sorted list, L, in an ascending order of the MI values.
3. Calculate the function values: $f(n) = [L(n+1) - L(n)] / [0.5*[L(n+1) + L(n)]]$, for every $n>0$.
4. Apply a low-pass filter, F, such as a moving average $g = F(f)$ to smooth the function values.
5. Find one or more peaks in g and mark them as candidate thresholds, $\{th_1,\ldots th_k\}$. These candidates initialize the search for the optimal threshold.
6. Start from the highest threshold value (th_k) and proceed towards the lowest threshold:
    a. Using the training set, calculate the BDeu score of the graph that is generated using this threshold.
    b. If the current BDeu score is significantly lower than the previously tested threshold, stop testing candidates and select the one (or more) that has the highest score.
7. Further fine-tuning of the threshold value can be done by maximizing the BDeu score.

An example is given in the images below for an ALARAM database of size 10,000 samples. The images were generated using the function:

```
SortedMI = MIStats(data);
```

In the left figure, we see the sorted MI values. In the right figure, we see the normalized differences between adjacent MI values in the sorted list and we can see a peak at index 401, which corresponds to an MI value of 0.003 in the left figure. The peak around index 0 should be always ignored as it may reflect an unreasonable low threshold value leading to a very dense structure.