Biolearn is a general package for applying probabilistic graphical models to biological applications. Biolearn release 1.0 is concentrated on structure learning for bayesian networks; future releases are expected to include other types of graphical models, support inference applications, and allow plug-and-play addition of new types of probability distributions, scoring functions and search algorithms.

Biolearn is implemented in java, and distributed as a jar. It is compiled in java version 1.6.0, and therefore requires java version 1.6.0 or later. In addition to the biolearn jar, the distribution also includes four jars from open-source providers that are used for specific functions of biolearn: jung-1.7.6.jar, provided by the JUNG Framework Development team at http://jung.sourceforge.net/; commons-collections-3.2.jar, provided by the Apache Commons project at http://commons.apache.org/collections/; colt.jar, provided by the Colt project at http://acs.lbl.gov/~hoschek/colt/ (these three jars are used for the graphical visualization of bayesian networks); and Jama-1.0.2.jar, provided by NIST at http://math.nist.gov/javanumerics/jama/, and used for implementing the MeanSquareError, ElasticNet and OrderedElasticNet scoring functions.

Invoking the biolearn structure-learning application

The biolearn structure learning application reads observation data on the nodes of a bayesian network, optionally performs discretization on the data, and then applies a search algorithm for finding an optimal network structure. The user can choose among several possible search algorithms and several possible scoring functions. The application can be invoked to run just one structure-learning search on the data, or to run any number of structure-learning searches using random samples of the data and perform model averaging. The results are output as text files, and optionally also visualized graphically.

The distribution includes four command files that can be used to invoke the biolearn structure-learning application; two windows bat files and two UNIX sh files. For each system there is one command file that invokes the application as a command-line utility, running non-interactively and providing its output only as a text output file; and one command file that invokes the application as a graphic interactive application, providing its output both as a text output file and through graphic visualization.

For large data sets, the default memory allocated by the java virtual machine may not be sufficient for the application, and it might crash with OutOfMemoryError. If this happens, the command files need to be edited to add the "-Xmx" argument to the java virtual machine to increase the maximum allocated memory. See java documentation for how to use this argument.

A run of the application is guided by a specification file, a text file specifying the input data, the choice of algorithm, scoring function and discretization methods, and other user-controlled options. The command files accept between one and three command-line arguments:

- The first, mandatory argument is the directory containing the five jar files.
- The second, optional argument is the name of the specification file. If the second

argument is omitted, the application by default searches for a file named biolearn.spec.txt in the current directory. The command-line version of the application fails if it cannot find the specification file; the interactive version, if it cannot find the specification file, opens a file chooser window to let the user find the specification file interactively.

- The third, optional argument is for specifying the starting point of the search. It is discussed below in the **search starting point** section.

The Biolearn specification file

Each line in the specification file specifies the user's choice for one of the user-controlled options for the run. In a non-interactive invocation, the input data and all options for the run must be specified in the specification file; in an interactive run, the input data and some of the options can be specified or changed interactively.

Input data

The biolearn structure-learning application expects to receive input data in one or more files in tabular form, with each line in the file consisting numerical values separated by tabs. There two possible arrangements of the input data files:

1. One data point per line, with each column representing one variable. In this arrangement the first line provides the names of the variables, with each subsequent line providing the values, in the specified variable order, for one of the data points. For example, a file with four data points and three variables with this format might look as follows:

A	B	C
0.74	0.03	0.04
0.66		0.09
0.73	0.6	1.11
0.77	0.88	1.07

2. One variable per line, with each column representing one data point. In this arrangement (often used for gene expression data), each line contains the name of the variable, followed by the values for that variable. The variable name may be in either the first of second column of each line; if it is in the second column, the first column is ignored (this accommodates Affymetrix files, in which the first column contains the gene’s ORF and the second column contains its name). In the first line either the first or second field should be “Name” or “GeneName”, to indicate which column contains the variable name; and the following fields specify names for the data points. For example, the data from the example above, in the variable-per-line format, would look as follows:

Name	DP1	DP2	DP3	DP4
A	0.74	0.66	0.73	0.77
B	0.03		0.6	0.88

C	0.04	0.09	1.11	1.07

or

ORF	Name	DP1	DP2	DP3	DP4
YDR1	A	0.74	0.66	0.73	0.77
YAR5	B	0.03		0.6	0.88
YOR3	C	0.04	0.09	1.11	1.07

Both arrangements allow missing values (for example the example above has a missing value for B in DP2); a missing value is represented by an empty cell (i.e. nothing between the two tab characters) or by any string that is not a number, such as a space, a question-mark, or the string "NaN".

Variable names may not contain white spaces, and may not contains the characters '.', ',' or '<'. Other than that all characters are allowed in variable names.

Different data files may have different but overlapping sets of variables. The structure search is constrained so that for any variable, there must always be at least one data file with data for that variable and all its parents.

Specifying input data

In an interactive run, the specification file must contain the line

dataformat *format*

where *format* is one of "DataPointPerLine" or "VariablePerLine". The input files themselves are specified by the user interactively, as described in the **graphic user interface** section.

In a non-interactive run, the specification file has to specify the actual input data. If the data consists of only one input file, it is specified as:

data *format input-file-name*

If there are sevaral input files, they can either be listed directly:

data MultipleInputFiles *format input-file-name-1 input-file-name-2 ...*

or by providing a file containing a list of the file names

data MultipleInputFiles *format list-file-name*

The name of the list file must contain the string ".filelist". Each line in the list file contains the name of one input file, and optionally also lists intervention information.

Interventions

The input data can optionally contain interventions on some of the variables, setting their values independently of their parents in the network (for an explanation of interventions, and how they can be used in a bayesian network structure-learning search, see "Bayesian Network Analysis of Signaling Networks: A Primer", Dana Pe'er, Science STKE 2005).

If input files are specified in a list file, it can be used to specify interventions; a line in the list file (other than the first, fixed line) is of the format

> *file-name intervened-variable-name intervened-variable-name ...*

or

> *file-name* suppress *intervened-variable-name intervened-variable-name ...*

listing the variables, if any, on which there are interventions in the data of this input file. When calculating scores during the structure-learning search, the values of this variable in data points from this data file are assumed to be independent of the variable's parents in the network. If the keyword "suppress" is specified, the values of this variable are assumed to always be the lowest possible value for this variable, and the actual values specified in the data file are ignored; if the keyword "suppress" is not specified, the actual data values for this variable are used.

When the application is run interactively, the interventions in each data files can also be specified interactively, as described in the **graphic user interface** section.

Variable status

Each variable in the input data has a status that is one of the following:

- Normal - this is the default status

- Ignore - the variable is not part of the bayesian network, and is ignored. This is useful if the data is read from input files that contains information about extraneous entities the user is not interested in.

- Alias - this variable should be treated as an alternate name for another variable, and the data under these two names should be treated as belonging to the same variable. This is useful if different input data files refer to the same entity by different names. The user should choose one of these name's as the variable's main name (and give it a status of "Normal", "Root" or "Leaf"); and give all the other names the status of "Alias".

- Root - incoming edges into this variable are not allowed in the network.

- Leaf - outgoing edges out of this variable are not allowed in the network.

The user can provide a file listing the status for each variable. The file is specified in the specification file with the line:

VariableStatusFile *status-file-name*

Each line in the status file lists one variable followed by its status; if the status is "Alias", this is followed by the variable's main name. A variable that is not listed in the status file is given the status "Normal"; if there is no status file, all variables have status "Normal".

In an interactive run of the application, the interface allows the user to interactively change variable status, and to write a new version of the status file for future runs, as described in the **graphic user interface** section.

Per file constants

In addition to the variables listed in the data files, biolearn can also accept variables that have a constant specified value within each data file. These variables are usually used to express experimental conditions that remain the same in all the data points in each data file, but vary among the different files.

To specify per file constants, the data files must be listed in a filelist as described in the **specifying input data** section above. In addition to the lines describing each data file, the filelist should also contain the line

PerFileConstants *constants-file-name*

The constants file consists of a tab-separated matrix in which each row represents a data file and each column represents a variable. The header line starts with the word “File”, followed by the names of the per file constants; each subsequent line starts with the same of a data file, followed by the values of each constant in this file. There must not be any duplication between variable names used as per file constants and variable names used in the data files.

For example, suppose you have four data files named F0, F1, F2, and F3, and two binary per file constants D1 and D2. Both D1 and D2 are off in F0; only D1 is on in F1; only D2 is on in F2; and both are on in F3. The contents of the constants file will then be as follows:

File	D1	D2
F0	0	0
F1	1	0
F2	0	1
F3	1	1

Per file constants can have either discrete or floating-point values. If they have floating-point values and the BDe scoring function (see below) is used, they are discretized in the same way as normal variables.

If in an interactive run the user loads several filelists, each one of them may have its own constants file. In that case the names of the per file constants in all these files must be identical.

If a data file is not listed in the constants file, the values of the per-file constants in that data file are treated as missing values. Biolearn issues a warning for each such file.

By default, per file constants have a status of Root. The user may change this status in the same way as for normal variables as described in the **variable status** section above.

data sampling

The application can either read all the data and use it in calculating scores during the search (the default option); or it can take a random sample of the data points.

A random sample of the data is specified with the line

> Sample *N* [NoReplacement]

The application takes N data points at random from each of the input data files. By default, the random sampling is done with replacement (i.e. the same data point may be included in the sample more than once); if the "NoReplacement" parameter is specified, the random sampling is without replacement. If "NoReplacement" is specified, each data file must have at least N data points.

Output

The name of the network is specified in the specification file in the line:

> DefaultNetworkName *network-name*

This name is used to create the names of the application's output files. By default, the application runs one structure-learning search, and writes its results in a file named *network-name*.network1.

The output file contains three sections.

- A header consisting of lines beginning with the # character and containing information on the run parameters. This information includes a listing of the input data files used in the search, and the interventions in each file; the discretization parameters if discretization was used (see **discretization** section below); and the various run parameters specified either in the specification file or interactively, as described in the sections below. This header is designed to help users keep track of the various outputs produced by biolearn and the data and parameters that resulted in each one.

- Depending on the scoring function used, a section listing the CPD of each variable in the network. If BDe scoring was used (see the **scoring functions** section below) this section is omitted. If NormalGamma scoring was used, this section lists a regression tree for each variable; if MeanSquareError scoring was used, this section lists a linear regression formula for each variable.
- A list of the network's edges. Each edge is written in the form

  variable1 --- *variable2*

  or

  variable1 --> *variable2*

  depending on whether the edge is directed or not.

The user can specify running several searches, and doing model averaging, by using the line:

NumRuns *N*

The application will then perform N searches, and write their results into files named *network-name*.network1, *network-name*.network2, ... *network-name*.networkN. If random sampling of the data was specified, each run is done with a different random sample.

When running several searches, the program also creates a file named *network-name*.confidences, listing network edges that have appeared in a majority of the search results and for each one the percentage of search results in which it appeared. By default, an edge will appear in the confidences file if it appears in the results of more than half the searches; the user can specify a different threshold with the line

ConfidenceThreshold *N*

An edge will then appear in the confidences file if it appears in the results of more than N% of the searches.

The confidences file also starts with the run-parameters header like the individual network files.

In an interactive run, the network name and the number of runs can be changed interactively, and in addition to the output files the program also displays a graphic visualization of the results. This is described in the **graphic user interface** section.

Scoring functions

The application provides a choice of three scoring functions that can be used in the structure-learning search; each scoring function is associated with a different type of probability distribution on the variables.

For a non-interactive run, exactly one Score line must appear in the specification file, and the application uses the specified scoring function. For an interactive run, the user chooses interactively among the first three scoring functions (BDe, NormalGamma or MeanSquareError), as described in the **graphic user interface** section; the specification file can contain any or none of the three possible Score lines, in order to specify non-default parameter values for the scoring functions.

- BDe scoring function, using conditional probability tables for the probability distributions on the variables. This is specified in the specification file with the line:

  Score BDe [*phantom-data-size*]

The default phantom data size is 5.

- Normal Gamma scoring function, using regression trees for the probability distributions on the variables. This is specified in the specification file with the line:

  Score NormalGamma [alpha=*alpha-value*] [gamma=*gamma-value*] [minSplit=*min-split-size*]

Default parameters are 1 for both alpha and lambda. Each leaf in a regression tree must always have at least min-split-size data points; default min-split-size is 5. If this scoring function is used, the second section of the output file lists the regression tree for each node.

- Mean Square Error as the scoring function, using linear gaussian probability distributions on the variables.

  Score MeanSquareError

If this scoring function is used, all nodes are assumed to have a linear gaussian probability distribution with a standard deviation of 1, and the linear formula for the mean is obtained by linear regression on the data values of the node and its parents. The second section of the output file lists the linear regression formula for each node.

- Elastic net scoring function, using linear Gaussian probability distributions on the variables.

Score ElasticNet Lambda2=*lambda2-value* Lambda1=*lambda1-value*

Elastic net scoring is described in the paper “Regularization and variable selection via the elastic net”, Zou and Hastie, J. R. Statis. Soc. B (2005). The score has two parameters, Lambda1 and Lambda2. Elastic nets are a generalization of Lasso (“Regression shrinkage and selection via the lasso”, Tibshirani, J. Royal. Statist. Soc B. (1996)); Lasso is a special case of elastic nets in which Lambda2=0.
When using this scoring function, the search is done using one of the search algorithms described in the **search algorithms** section below, usually using GreedyHillClimbing. Whenever a step is taken in the search, of adding or removing a parent from a node, the new coefficients are computed to optimize the elastic net score, which is then used to evaluate the step.

Elastic net scoring is only available in non-interactive runs.

- Elastic net scoring function over an ordering of the nodes.

  Score OrderedElasticNet Lambda2=*lambda2-value* Lambda1=*lambda1-value* L1NormLimit=*L1-norm-limit-value*

This score provides an alternative way of using elastic nets. This function causes the search to be made, not over the possible Bayesian network, but over the possible orderings of the nodes. The nodes of the network are arranged in an order; each search step consists of switching two nodes in the order. The linear Gaussian formula for each node is then calculated using the Larsen algorithm (described in Zou and Hastie), and evaluated using elastic net scoring.

Note that the Larsen algorithm has the two parameters Lambda2 and L1NormLimit (a limit over the sum of the absolute values of all coefficients); elastic net scoring has the two parameters Lambda2 and Lambda1. Using ordered elastic nets therefore requires specifying all three parameters.

Ordered elastic nets are available only in non-interactive runs. They work only with the GreedyHillClimbing algorithm. They do not allow specifying any constraints (other than the implicit constraints of variables with Root or Leaf status).

By default, the search starts with a random ordering over the variables. Alternatively, the BestForest algorithm (described in the **search algorithms** section below) can be used to find the initial ordering, by specifying the lines

InitialStructure BestForest
InitialStructureScore MeanSquareError

The BestForest algorithm is then used to find the best initial forest (i.e. the best network in which each variable can have no more than one parent), using simple linear regression to calculate the coefficients and MeanSquareError scoring. A topological ordering over this initial network is then used as the starting-point of the search.

Edge and split penalties

The user may specify a score penalty on adding edges to the network. This is specified by the line

Prior EdgePenalty *N*

During the search process, N is subtracted from the score for each edge in the network. (Effectively, an edge will be added to the network only if it improves the score of the main scoring function by more than N).

The appropriate value for N will often be hard to decide on, and will be different for different scoring function used. To deal with this difficulty, an alternative way to specify the edge penalty is

Prior EdgePenalty Fraction *F*

F is a small number, usually between 0 and 1. When the edge penalty is specified in this way, biolearn examines all possible single edge additions, discards those that worsen the score, and for those that improve the score (not taking the edge penalty into account) finds the median score improvement; the edge penalty is then set at this median score improvement multiplied by F.

When using the NormalGamma scoring function, the user may also specify a penalty on splits in a regression tree, using the line

Prior SplitPenalty *N*

or

Prior SplitPenalty Fraction *F*

During the search process, N, or the median single-edge improvement multiplied by F, is subtracted from the score for each split in each regression tree CPD.

If both the edge penalty and the split penalty are specified, then when splitting a regression tree with a new parent both penalties are subtracted from the score; when splitting a regression tree with an existing parent, which has already been used in other splits on this regression tree, only the split penalty is subtracted from the score.

Discretization

The NormalGamma and MeanSquareError scoring functions are suitable for dealing directly with continuous input data. The BDe scoring function, however, requires discrete input data; if the BDe scoring function is used and the input data is continuous, it is automatically discretized. In an interactive run, the form of discretization can be specified interactively, as described in the **graphic user interface** section; in a non-interactive run, it has to be specified in the specification file, as described below.

Continuous data is discretized by dividing the range of values into a number of buckets; the BDe score is calculated using joint counts, which are obtained by counting the data points that have values in each bucket. The number of buckets can be any number between 2 and 6, or it can be 10, 15 or 20.

The boundaries between buckets can be either hard or soft. If the boundary is hard, it consists of one threshold value; if the value of a variable in any data point is less than the threshold, it is counted as being in the lower bucket; if the value is larger than or equal to the threshold, it is counted as being in the higher bucket.

If the boundary is soft, it consists of two threshold values. If the value of a variable in any data point is less than the lower threshold, it is counted as being in the lower bucket; if the value is larger than or equal to the higher threshold, it is counted as being in the higher bucket; if the value is between the two thresholds, it is counted partly in each bucket, in proportion to its distance from the two thresholds. For example, is the soft threshold is 0.9:1.1 and the variable at a given data point has a value of 0.95, it adds .75 to the count for the lower bucket and .25 to the count for the higher bucket.

The user can either specify the boundaries between buckets as absolute numbers, or have biolearn choose them automatically based on the values of each variable. The boundaries are specified as absolute numbers with the line:

DiscretizationDefault *boundary1 boundary2 ...*

Each boundary is either a single number to specify a hard boundary, or of the form *low-number*:*high-number* to specify a soft boundary.

Automatic calculation of the bucket boundaries is specified with the line:

DiscretizationBuckets *N boundary_type division_method*

Where *N* is the number of buckets, *boundary_type* is either "Hard" or "Soft", and *division_method* is either "ByDistance" or "BySize". Division by distance means that the range of values of the variable is divided into N equal sub-ranges, so that the difference between the upper boundary and the lower boundary is equal for each bucket. Division by size means that the set of values of the variable in the data is divided into N sets of equal size, so that for every variable the number of values in the data in each bucket is the same. When automatically setting a soft boundary, the distance between the low and high thresholds is set to be 40% of the size of a full bucket.

If there is no line in the specification file to specify default discretization, the default discretization parameters are 3 buckets with hard boundaries divided by distance.

The user may also specify the discretization boundaries separately for specific variables, using the line:

Discretization *variable1 variable2 ...* : *boundary1 boundary2 ...*

The specification file may contain any number of Discretization lines, as long as any variable is mentioned in at most one such line. Variables not mentioned by name in a Discretization line are discretized according to the default parameters specified in the DiscretizationDefault or DiscretizationBuckets line.

Search algorithms

The application provides a choice of several search algorithms used in searching for the optimal structure. The specification file must contain a line specifying the choice of algorithm (in an interactive run, the choice of algorithm still needs to to be specified in the specification file and cannot be changed interactively).

The available algorithms are as follows:

- Greedy Hill Climbing - this is likely to be the most useful algorithm for most applications. It is an enhanced version of the classic greedy hill climbing algorithm, using random restarts to avoid getting stuck too easily in a local maximum, and using tabu search to search plateaus (i.e. regions of the search space in which a single step leaves the score unchanged).

To use the greedy hill climbing algorithm, the specification file should contain the line:

Algorithm GreedyHillClimbing [restarts=*N*] [plateauMax=*M*] [searchLog=L]

N is the number of random restarts. Default is 0 (i.e. simple greedy hill climbing without random restarts).

M is the maximum number of search steps that can be taken in a plateau. If M steps have been taken and no step has been found yet to increase the score, the search is ended. Default is no limit (i.e. any plateaus completely until a step is found to increase the score or until the entire plateau has been explored). To completely turn off searching of plateaus, specify "plateauMax=0".

If the “searchLog” parameter is specified, biolearn writes a log of the search to the standard error stream; at each step it records the step that was taken, the improvement that step made to the score, and compares it to all other edges that could have been added to the same variable and their effect on the score. This log can be used to get a better understanding of the workings of the scoring function, and to help in deciding on appropriate values for future runs for the edge or split penalties. L is the maximum number of lines in the log; biolearn stops producing the log when L lines have been printed.

By default, when using the BDe or MeanSquareError scoring functions, a single step in the search consists of adding, removing or reversing a single edge.

With the MeanSquareError scoring function, when the program adds a new edge it also checks the p-value of the score improvement of this edge; it creates a number of random permutations of the values of the source node, runs linear regression for each permutations, and infers the p-value of the edge by the fraction of permutations that resulted in a smaller mean square error than the actual data; the edge is allowed only if the p-value is below the threshold. This check is necessary since otherwise mean-square-error scoring will result in severe over-fitting.

By default the p-value threshold is .05, and is checked using 1000 permutations. The user can change these parameters by including in the specification file the line

```
LinearChoiceTest PermutationTest permutations=N threshold=P
```

Where P is the desired p-value threshold and N is the desired number of permutations. This line has no effect when using the BDe or NormalGamma scoring functions.

When using the NormalGamma scoring function, a single step in the search consists of either splitting a leaf in one node's regression tree or of unsplitting an existing split. In some cases such a step changes the node's set of parents, and thus causes an edge to be added or removed in the network; in other cases (for example if we split using a parent that's already been used in other splits in the same regression tree) the edges of the network do not change.

When using the BDe or MeanSquareError scoring functions, the user may enhance the possible steps to also allow parent exchange. A parent exchange consists of removing an edge A->B and adding a new edge C->B (and if MeanSquareError is used, also checking the p-value of the new edge). To allow parent-exchange steps in the search (in addition to edge addition, removal or reversal), the user should include in the specification file the line

```
Modifier AddRemoveReverse WithExchangeParent
```

This will cause the program to run slightly longer, since there are more possibilities to check at each step. It is especially useful if using the RequiredPath constraint (described below in the **constraints** section). Note that adding the exchange-parent step is not allowed if using the NormalGamma scoring function.

- Sparse Candidate.

The Sparse Candidate algorithm is designed for structure learning in very large networks. It is described in *Learning Bayesian network structure from massive datasets: The "sparse candidate" algorithm*, Friedman, Nachman and Pe'er, *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence* 1999, 206-215. As a rule of thumb, if the search involves more than 50 variables, use of the sparse candidate algorithm is likely to be necessary.

To use the sparse candidate algorithm, the specification file should contain the line:

> Algorithm SparseCandidate *P* GreedyHillClimbing [restarts=*N*] [plateauMax=*M*] [searchLog=L]

Where P is the size of the candidate sets.

The sparse candidate algorithm incorporates the greedy hill-climbing algorithm, and so has all of the same parameters. It uses the same search steps, and also allows use of the "Modifier AddRemoveReverse WithExchangeParent" line to allow parent exchange as a search step.

- Exhaustive search - this algorithm is prohibitively expensive except in very small networks; but when the network is very small, it is guaranteed to find the optimal structure. It exhaustively checks all possible network structures, and outputs the one with the highest score.

To use the exhaustive search algorithm, the specification file should contain the line:

> Algorithm ExhaustiveSearch

The exhaustive search algorithm can be used only with discrete models, using the BDe scoring function.

- Best Forest - this algorithm searches specifically for a forest-form network, i.e. a network in which each node can have at most one incoming edge. It finds the optimal such network using a variation on the classical minimum-spanning-tree algorithm.

The best forest algorithm can be used in two ways. It can be used as the main search algorithm, with the application producing the optimal forest-form network as its output; to do that, the specification file should contain the line:

> Algorithm BestForest

Alternatively, the best forest algorithm can be used to create a starting-point for the search; the application starts the search by applying the BestForest algorithm, and then uses this optimal forest-form network as the starting point for a search using the greedy-hill-climbing or the sparse-candidate algorithm; in some cases this will help the search find better structures than if it started from an empty network. To use the best-forest algorithm in this way, the specification file should contain the "Algorithm GreedyHillClimbing" or the "Algorithm SparseCandidate" line, as described above, and also contain the line

```
InitialStructure BestForest
```

If BestForest is used for finding the initial structure, by default the same scoring will be used in the BestForest search as in the later search algorithm. If the user wishes to use different scoring functions in the two searches, he can do so by specifying the line

InitialStructureScore *score-function*

specifying a scoring function and parameters as described in the **scoring functions** section above.

In the current version, the implementation of the BestForest algorithm is not compatible with specifying any constraints with the exception of ParentMaximum (see the **constraints** section below). If any other constraints are specified, the BestForest algorithm cannot be used.

Search starting point

When using the greedy hill-climbing or sparse candidate algorithms, the search starts from a given point in the search space. By default the search starts from an empty network; biolearn provides the option for the user to specify the starting point. This can be useful if the user has reason to believe some specific combination of edges should be in the network. The heuristic search starting from the empty network may end up missing this combination of edges and never trying it; by starting the search with an initial network containing this combination of edges, the user makes sure this combination is tried.

In an interactive run, the user can specify the starting network interactively, as described in the **graphic user interface** section.

In a non-interactive run, the user specifies the starting network as an optional third command-line argument, pointing to a file containing the input network, as described in the **Invoking the biolearn structure-learning application** section.

In this version, non-empty starting points work only with the BDe scoring function. The input network must be either the output of a run with the BDe scoring function, or a manually edited list of edges. The first section of biolearn output, containing run-parameters information, is optional in the input and is ignored.

Specifying a starting point is meaningful only if the greedy hill-climbing or the sparse-candidate search algorithm is being used.

Constraints

There several types of constraints that the user can specify over the structure of the network. Constraints are specified in the specification file (they cannot be changed interactively). The specification file may contain any number of lines specifying constraints; the application will issue an error if there is no possible network that can satisfy all specified constraints.

The possible constraints are as follows:

Constraint ParentMaximum *N [variable1 variable2 ...]*

The named variables are limited to an in-degree of N. If no variable names are given, all variables in the network are limited to an in-degree of N.

Constraint RequiredEdge *var1 var2*

The edge var1->var2 must be in the network.

Constraint RequiredPath *var1 var2*

A path from var1 to var2, direct or indirect, must be in the network.

Constraint NoEdge *var1 var2*

The edge var1->var2 is not allowed in the network.

Constraint NoPath *var1 var2*

Any path from var1 to var2, direct or indirect, is not allowed in the network.

Constraint SplitMaximum *N*

If using the NormalGamma scoring function, the regression tree of any node is limited to N splits. This constraint has no effect when used with the BDe or MeanSquareError scoring functions.

Constraint MissingValuesMaximum *P*

P must be a number between 0 and 1. If more than a P fraction of the values of a variable are missing, it cannot be be a parent in the network; if P is 0, any variable with any missing values is forbidden from being a parent in the network. When using the BDe or MeanSquareError scoring functions, the restriction is on the fraction of missing values in the entire data (so any variable with more than P fraction of missing values cannot have any outgoing edges); when using the NormalGamma scoring function, the restriction is on the fraction of missing values in the data for the leaf that is being split.

In general, all available scoring functions and all available algorithms can be used with any of the constraints. There are, however, some specific restrictions:

- If the NormalGamma scoring function is used, the RequiredEdge and RequiredPath constraints cannot be used.

- If the BestForest search algorithm is being used, the only allowed constraint is ParentMaximum (specifying this constraint makes no sense when using BestForest as the main algorithm, since the result is already constrained to an in-degree of 1 for all nodes; but it may be useful when BestForest is used for creating the initial structure). Other constraints cannot be used.

Probability of specific network features

When using the exhaustive search algorithm, the application provides the option of specifying certain features of the network - the presence of specific edges, or conditional independence of pairs of variables given some subset of the other variables – and calculating their probability. If the specification file contains one or more feature specifications, and the exhaustive search algorithm is used, then in addition to producing the optimal network as output the application also writes to the standard output the probability of each feature. The probability of a feature is the sum of the probabilities of all possible networks that have this feature; the probability of a network is e to the power of the network's score, normalized so that the sum of probabilities of all possible networks is 1.

In this version, calculation of feature probabilities is handled entirely non-interactively. In an interactive run, features still have to be specified in the specification file, and their probabilities are still written to the standard output.

Since calculation of feature probabilities requires use of the exhaustive search algorithm, it is only possible on very small networks, and only with discrete models using the BDe scoring function.

Each feature is identified by a feature number. The two types of basic features are the presence of an edge or the conditional independence of two variables; these are specified by the following lines:

Feature *N* Edge *variable1 variable2* [undirected]

N is the feature number. If "undirected" is not specified, the feature is the presence of the edge variable1->variable2; if "undirected" is specified, the feature is the presence of an edge connecting variable1 and variable2, in either direction.

Feature *N* CI *variable1 variable2 variable3* ...

N is the feature number. The number of variables listed in the line must be three or more; the feature is the conditional independence of variable1 and variable2 given the other listed variables.

A Feature may be a conjunction of one or more basic features, specified by one or more lines with the same feature number. The specification file may list any number of different features with different feature numbers, and the application outputs the probability of each of them.

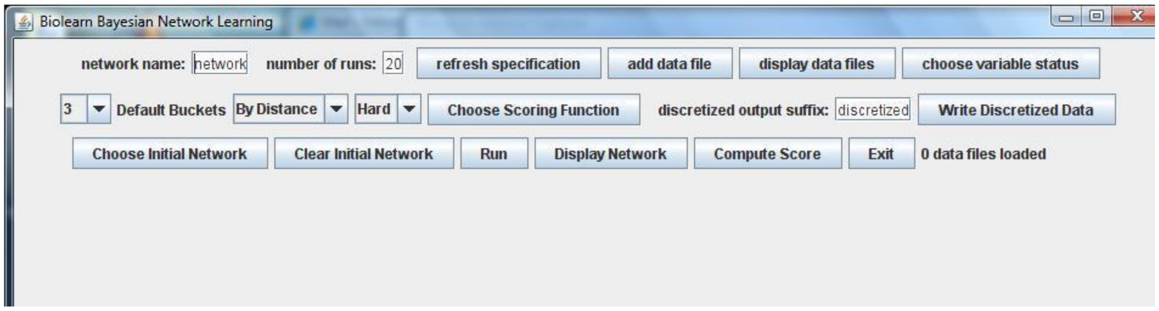
Note that if you run several search runs using exhaustive search, with model averaging, and also specify a list of features, the probability of the features will be output for each of the search runs. The probabilities in the different runs may be different, depending on the random sample taken of the data in each run.

Graphic User Interface

In the preceding sections we described capabilities available in both interactive and non-interactive runs of the applications. This section described the additional capabilities provided by the application's graphical user interface, when the application is invoked interactively.

The graphical user interface is limited in the number of variables that it can handle. It will not work for data that involves a very large number of variables; as a rule of thumb, do not use the graphical user interface for network searches dealing with more than 50 variables.

When the application is invoked interactively, after reading the specification file, it displays the following window:



Network name

The "network name" text field provides that name that will be used as the network name in creating the output files, as described above in the **output** section. The user can choose the file names by changing this text field.

Number of runs

The “number of runs” text field allows the user to specify the number of search runs that will be done each time the “Run” button is clicked, and used for model averaging.

Refresh specification

Clicking the “refresh specification” button re-reads the specification file. Since in the current implementation there are some options (such as the choice of search algorithm, or constraints) that have to be specified in the specification file, this allows the user to change them during a run, by editing the specification file and then clicking the “refresh specification” button.

Add data file

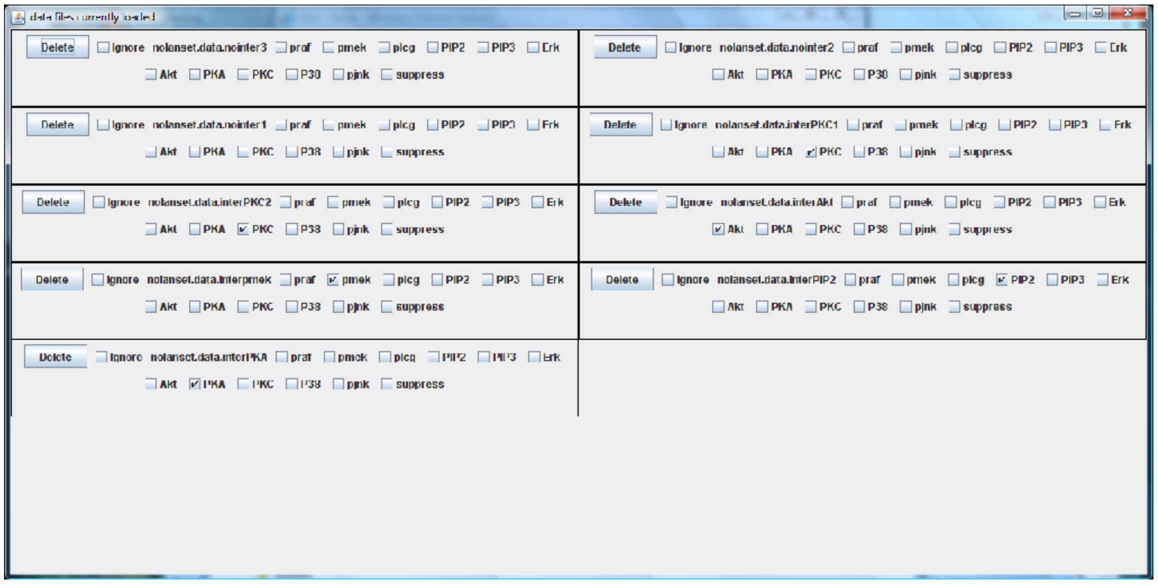
As discussed above in the **specifying input data** section, in an interactive run the specification file only specifies the format of the input data. The input data files themselves have to be chosen interactively.

To read in a new data file, the user clicks on the “add data file” button. This opens a file chooser window, and the user can either choose a single data file, or a list file, which the application then reads in, and updates the displayed count of data files loaded.

Display data files

Clicking on the “display data files” button opens a window displaying a list of the data files currently loaded, and the list of the variables in each one. Each variable is associated with a check-box which is checked if the data file has an intervention on that variable.

For example, if you load the nine files listed in science2005.filelist, and then click on “display data files”, the following window will appear:


data files currently loaded

Delete ☐ Ignore nolanset.data.nointer3 ☐ praf ☐ pmek ☐ plcg ☐ PIP2 ☐ PIP3 ☐ Erk ☐ Akt ☐ PKA ☐ PKC ☐ P38 ☐ pjnk ☐ suppress

Delete ☐ Ignore nolanset.data.nointer2 ☐ praf ☐ pmek ☐ plcg ☐ PIP2 ☐ PIP3 ☐ Erk ☐ Akt ☐ PKA ☐ PKC ☐ P38 ☐ pjnk ☐ suppress

Delete ☐ Ignore nolanset.data.nointer1 ☐ praf ☐ pmek ☐ plcg ☐ PIP2 ☐ PIP3 ☐ Erk ☐ Akt ☐ PKA ☐ PKC ☐ P38 ☐ pjnk ☐ suppress

Delete ☐ Ignore nolanset.data.interPKC1 ☐ praf ☐ pmek ☐ plcg ☐ PIP2 ☐ PIP3 ☐ Erk ☐ Akt ☐ PKA ☑ PKC ☐ P38 ☐ pjnk ☐ suppress

Delete ☐ Ignore nolanset.data.interPKC2 ☐ praf ☐ pmek ☐ plcg ☐ PIP2 ☐ PIP3 ☐ Erk ☐ Akt ☐ PKA ☑ PKC ☐ P38 ☐ pjnk ☐ suppress

Delete ☐ Ignore nolanset.data.interAkt ☐ praf ☐ pmek ☐ plcg ☐ PIP2 ☐ PIP3 ☐ Erk ☑ Akt ☐ PKA ☐ PKC ☐ P38 ☐ pjnk ☐ suppress

Delete ☐ Ignore nolanset.data.interpmek ☐ praf ☑ pmek ☐ plcg ☐ PIP2 ☐ PIP3 ☐ Erk ☐ Akt ☐ PKA ☐ PKC ☐ P38 ☐ pjnk ☐ suppress

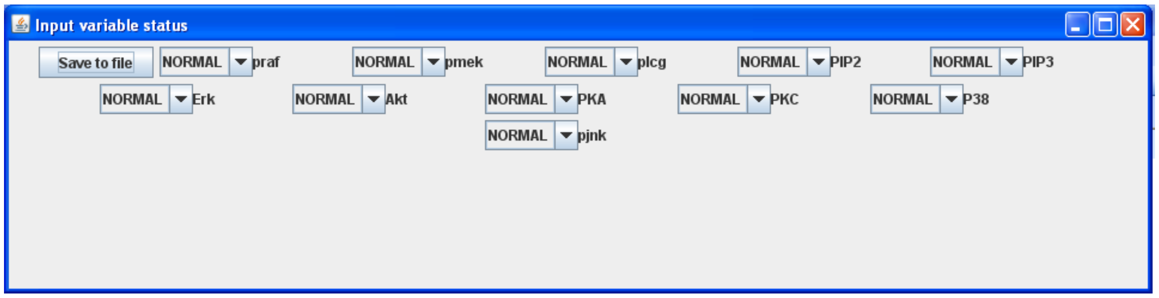
Delete ☐ Ignore nolanset.data.interPIP2 ☐ praf ☐ pmek ☐ plcg ☑ PIP2 ☐ PIP3 ☐ Erk ☐ Akt ☐ PKA ☐ PKC ☐ P38 ☐ pjnk ☐ suppress

Delete ☐ Ignore nolanset.data.interPKA ☐ praf ☐ pmek ☐ plcg ☐ PIP2 ☐ PIP3 ☐ Erk ☐ Akt ☑ PKA ☐ PKC ☐ P38 ☐ pjnk ☐ suppress

The user can instruct biolearn to ignore some of the data files, by checking the "Ignore" check-box for the unwanted data files; or remove some or all of the data permanently, by clicking on the "delete" button. The user can change the interventions in each file by checking or unchecking the boxes for intervened variables, and checking or unchecking the "suppress" checkbox, which is the equivalent of the "suppress" keyword in the list file (see the **specifying input data** section).

Choose variable status

Clicking the "choose variable status" button opens a window that lists all variables, with a choice menu for each one allowing the user to choose its status; for example:


Input variable status

Save to file

NORMAL praf | NORMAL pmek | NORMAL plcg | NORMAL PIP2 | NORMAL PIP3 | NORMAL Erk | NORMAL Akt | NORMAL PKA | NORMAL PKC | NORMAL P38 | NORMAL pjnk

The status options are the ones discussed above in the section: Normal, Ignore, Alias, Root and Leaf. If Alias is chosen, another choice menu appears for choosing the variable's main name for which this name is an alias.

If the user clicks on the "save to file" button, the current choice of status for each variable is written into the status file, so future invocations of the application in the same directory will automatically be initialized to these status choices.

Default discretization options

The main window displays threes choice menus that allow the user to change the default discretization options, i.e. the discretization options to be used for variables that don't have discretization boundaries specified separately. The user can choose the number of buckets, whether the division is by distance or by size, and whether the boundaries are hard or soft.

These discretization options have an effect only if the input data is continuous and the BDe scoring function is chosen. If the input data is discrete, or if the user chooses the NormalGamma or MeanSquareError scoring functions, discretization options have no effect.

Choose scoring function

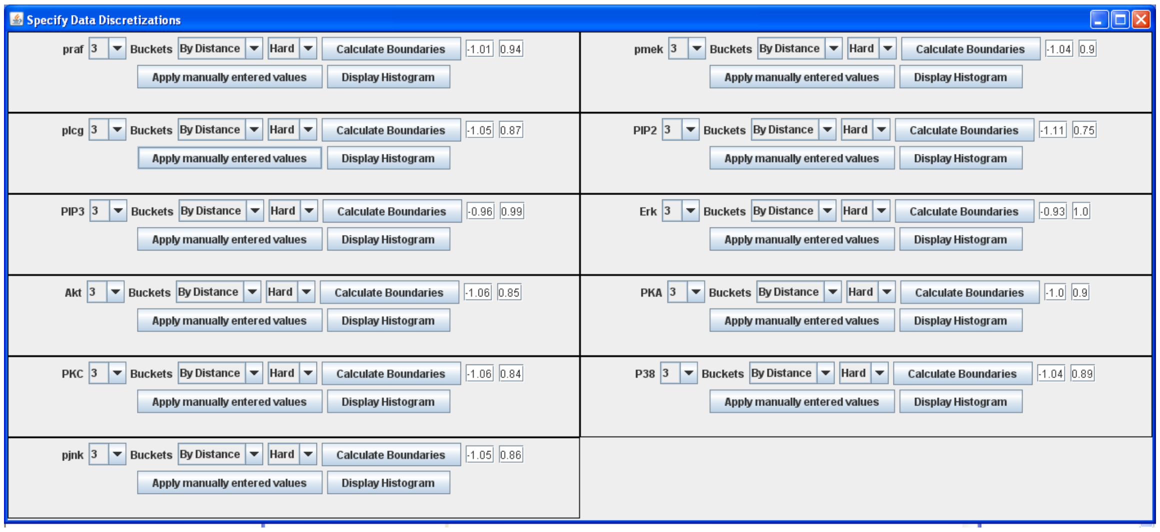
The user can choose the scoring function to be used by clicking the "choose scoring function" button. If the user does not click on the "choose scoring function" button, the default scoring function is BDe.

If all data values in the input are whole numbers, and the range of different values for each variables is less than 20, then the data is considered discrete, and BDe is the only scoring function that can be used; clicking the "choose scoring function" button displays a message stating that the data is discrete.

If input data is continuous, then clicking on the "choose scoring function" opens a window with three buttons, allowing the user to choose one of the three scoring functions.

Specifying discretization on each variable

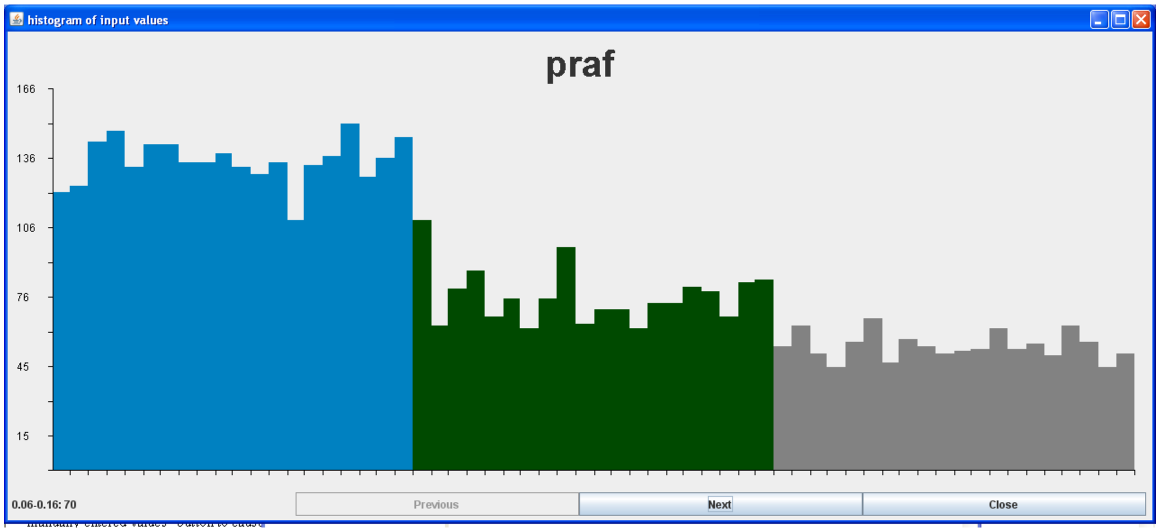
If the data is continuous, and the user clicks on the "choose scoring function" button and then chooses BDe, this means that the data will have to be discretized. The application then opens a window for specifying the discretization options for each variable:



praf	3 Buckets	By Distance	Hard	-1.01	0.94
plcg	3 Buckets	By Distance	Hard	-1.05	0.87
PIP3	3 Buckets	By Distance	Hard	-0.96	0.99
Akt	3 Buckets	By Distance	Hard	-1.06	0.85
PKC	3 Buckets	By Distance	Hard	-1.06	0.84
pjnk	3 Buckets	By Distance	Hard	-1.05	0.86
pmek	3 Buckets	By Distance	Hard	-1.04	0.9
PIP2	3 Buckets	By Distance	Hard	-1.11	0.75
Erk	3 Buckets	By Distance	Hard	-0.93	1.0
PKA	3 Buckets	By Distance	Hard	-1.0	0.9
P38	3 Buckets	By Distance	Hard	-1.04	0.89

This window contains for each variable three choice menus that allow the user to choose the number of buckets, the division method and the boundary type for that variable. It also displays the absolute boundary values for each variable (the absolute boundary values specified in the specification file, or, if none are specified, the values calculated using the default parameters). The user can change the number of buckets, the division method and/or the boundary type for a specific variable, and then click the "calculate boundaries" button to re-calculate the absolute values of the boundaries. The user can also manually change the absolute values of the boundaries, and then click the "apply manually entered values" button to cause these changes to take effect.

In some cases, deciding on where to place the boundaries can be helped by viewing a histogram of the data values of each variable. If the user clicks on the "display histogram" button, the application displays such a histogram:



0.06-0.16: 70

The histogram divides the variable's range of values into 60 equal-distance buckets, and displays a column for the number of values in each bucket; the histogram is divided into colors based on the variable’s discretization buckets. When the user moves the cursor over the histogram, the panel displays the range and height of the column to which the cursor is currently pointing.

Writing discretized data

An interactive run of the application provides the option of writing out a discretized version of the data, so future applications can read discrete data and need not do the discretization again.

A discretized version of the data is written out when the user clicks on the "write discretized data" button. For each input data file, the application creates an output file with a discretized version of the same data. All discretization parameters that the user has specified (the default parameters as well as any boundaries separately specified for individual variables) are used in the discretization. By default, a discretized output file has the same name as the input data file with the suffix ".discretized" added; the user can change the suffix in the "discretized output suffix" text field.

Writing out discretized data is not allowed if soft bucket boundaries have been specified.

Discretized output data is always in DataPointPerLine format, even if the input was in VariablePerLine format. Variables that have a status of "Ignore" are omitted from the discretized output; a variable with several aliases appears in the output always under its main name.

Choosing the search starting-point

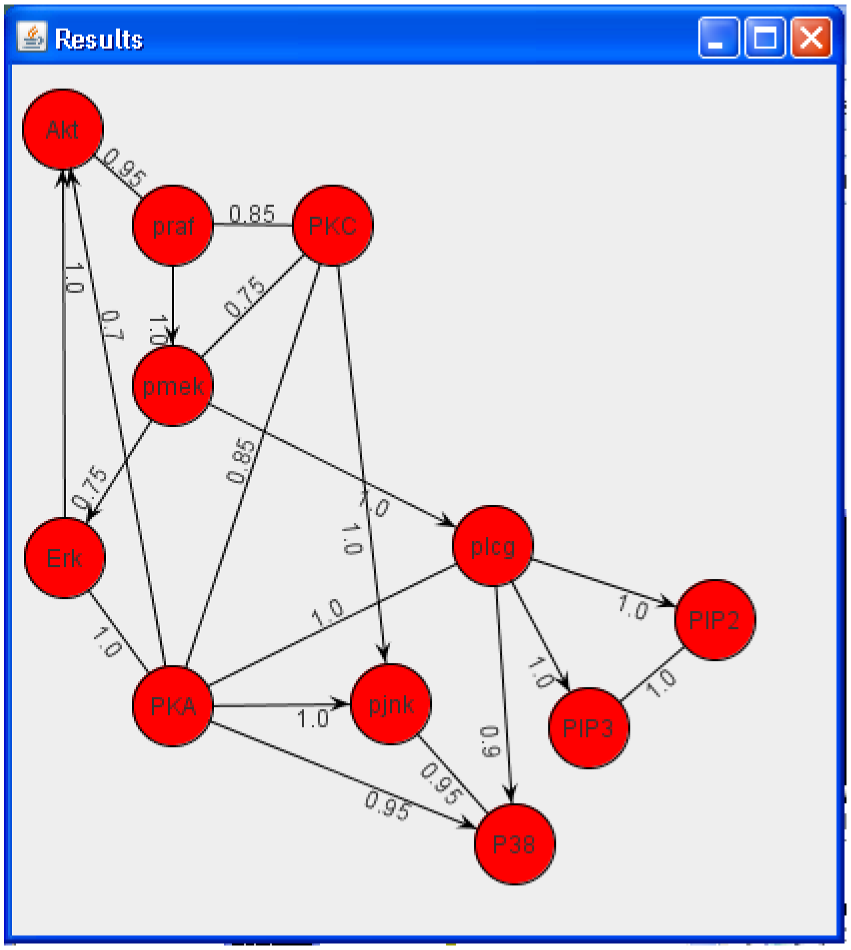
As discussed above in the **search starting point** section, the user may specify a network to serve as a starting-point for the search. In an interactive run, this is done by clicking on the “Choose Initial Network” button; biolearn opens a file chooser window which allows the user to specify a file containing the initial network. To change the choice of initial network, click on “Choose Initial Network” again and choose a different file; or to go back to the default starting point of an empty network, click on “Clear Initial Network”.

Running structure-learning searches

The main purpose of running the biolearn application is usually to run the structure-learning search. Once all data has been loaded, and the desired scoring function and, if needed, the discretization parameters have been chosen, the user then clicks on the "Run" button to run the searches.

The number of searches run is as specified in the "number of runs" text field. The window displays a message indicating how many searches have been completed so far. The search results are written into output file - a file for the network resulting from each search, and a confidences file for the results of model averaging - as discussed above in the **output** section.

When all the runs are finished, the results are also displayed graphically:



If several searches were run for model averaging, the graphic display shows the edges that had confidences above the threshold, with each one labeled with its confidence level (1.0 means the edge appeared in the results of all runs; .95 means it appeared in 95% of runs; etc.). If only one search was done, the graphic display shows the resulting network. The display indicates whether the edge is directed or not. Note that if the NormalGamma or MeanSquareError scoring functions are used, the network output files contain both the network's edges and probability distributions for each variable; the graphic display still shows only the network's edges.

If a variable has no edges connecting to it in the network, either in or out, it does not appear in the graphical display.

The application tries to lay out the nodes so as to make the network as readable as possible, but the layout is heuristic and is not always good; the user can use the mouse to move the nodes around to manually create a more readable layout.

Displaying a previouly-created network

The application can also be used to display a network or a confidences file that was created previously, either in this run or in a previous run of biolearn. When the user clicks on the "display network" button, the application opens a file chooser window that allows the user to choose a network or confidences file, and then graphically displays the edges in that file. This can be used either to display again the results of the current run, or to display the results from a previous run of biolearn.

Calculating the score of a network

The score of a network is normally used by the application only internally, to compare networks to each other during a search. In some cases, however, in order to better understand the results provided by biolearn, the user may wish to know the scoring results for specific networks.

When the user clicks on the "compute score" button, the application opens a file chooser window that allows the user to choose a network file. The application then computes the score for the network, and displays the component score computed for each variable and the total score for the network.

For all three scoring functions used, the component scores and the total network score are always negative numbers; a better network has a higher score, i.e. a negative score of smaller magnitude.

The type of score used to compute the score of the network depends on the type of probability distribution contained in the network file. If the network file contains regression tree distributions, the score is computed using the NormalGamma scoring function; if the network file contains linear regression approximations, the score is computed using the MeanSquareError scoring function; if the network file contains only the network edges, the input data is discretized and the score is computed using the BDe scoring function.

This function can be used to compute the score of a network created either by the current run or by a previous run (the user can also manually create network with the editor to check the score). The input data currently loaded must have the same variable names as the variable names used in the network.

If the specification file specifies random sampling of the data (i.e. contains a "Sample" line), a new random sample of the data is taken each time the "compute score" button is clicked; it is therefore to be expected that several score computations on the same network would give slightly different results. If the specification file does not contain a "Sample" line, score computation is node using the entire input data, and so several score computations on the same network and with the same data will give precisely the same results.

The component scores displayed for individual variables do not take the edge and split penalties into account; the total network score displayed does take these penalties into account. Thus if the specification file does not specify any edge or split penalties, the total score displayed is equal to the sum of the component scores; if there are edge or split penalties, the total score will be less than (i.e. of greater magnitude than) the sum of the component scores.

Exiting the application

Any number of search runs can be made during one interactive run, with the same input data or with different data. The application ends when the user clicks the "exit" button or closes the main window.