

Mining Bayesian Networks from Streamed Data

Facundo Bromberg, Brian Patterson, and Sandeep Yaramakala

CS 561 Final Report, Spring 2003

Introduction

Bayesian networks, created by Judea Pearl [Pea98], have become important as a model of uncertainty, giving us an easily understandable way to see relationships between attributes of a set of records. Computationally, Bayesian networks provide us with a way to store relationships between attributes that is efficient and still allows reasonably fast inference of probabilities we may be interested in.

A lot of recent research has been oriented towards learning these networks from raw data as the traditional designer of a Bayesian network, the domain expert, may not be able to see (or even know) all of the relationships between the attributes. Therefore, verifying (or even initially generating) these relations with data hopefully leads to more accurate models. One example solution is Lam and Bacchus's [LB93] use of Rissanen's Minimal Description Length (MDL) principle [Ris78] to detect relationships between attributes without adding unnecessary complexity to the model returned.

Most of these learning algorithms rely on having full access to the whole database throughout the learning process. However, in some application areas it may be infeasible or undesirable to store the data we are analyzing after we've looked at it once. One example is a large database with large records stored on a distant server such that we cannot locally store more than a few records (as our computer has a much smaller capacity) and do not want to retransmit records (as the connection is slow, unreliable, or the larger database periodically drops old records).

Another example where we don't have full access to all of the data at one time is that of usage patterns of a web site: by analyzing the web server's log, we can extract information about such attributes as browser type, time of access, documents downloaded, etc. and infer some relations (such as, if users of a certain browser do very few page views, perhaps the web site is not rendering correctly in that browser). However, for busy servers, the web log is periodically purged to save space on the server so any individual web log we can access only represents a small sample from the life of the server. If we wanted to get a larger data set to draw from, we could mine the web log for an initial network and then alter that network if future web logs indicate that changes are warranted.

There are several possible approaches to incremental learning explored in the literature [FG97]. The simplest approach is to simply store all previously seen data and invoke the learning procedures used on static databases each time we have more data. However, this approach forces us to store a possibly

prohibitive amount of data and process the initial records a huge number of times. The other extreme is to remember none of the previously received data and only remember the current-best Bayesian network structure. While space-efficient, this approach leads to a strong bias towards data encountered early in the process because this data will have an easier time altering the structure of the network (when, if anything, we would rather the model be biased towards new data if the underlying distribution of the data were shifting).

As a compromise between these two approaches, Friedman and Goldszmidt [FS97] proposed the search of “frontier” Bayesian networks. These frontier networks would be Bayesian networks that differ by one “alteration” from the current best (candidate) network (where an “alteration” is adding, deleting, or transposing an edge from the network). To track this frontier and evaluate them all, they propose maintaining conditional probability tables (CPTs)¹ for all of the nodes in the candidate network plus CPTs for each network in the frontier (representing the CPT generated by the alteration of the network that network on the frontier represents).

The way that the search for the best network proceeds is that the current best network and the frontier are reexamined in the light of new data, a new candidate network is selected, new frontier networks are generated from this new candidate by altering it in some fashion, and then new data is read and the process repeats itself. Lam and Bacchus [LB93] used an ordered queue of arcs to add to candidate networks (ordered by the description length of each possible arc) so, to let the process run forever, when the worst arc is tested, we circle back to the best one.

In this way, we still have a “best” network that we can export and use in other applications at any time but are maintaining the frontier to allow for easier alteration of the network. Also, the size of the frontier can be set by the user to provide a space/accuracy trade-off (i.e. in theory, the larger the frontier, the more accurate we are but more space is used to store the frontier).

However, we have an additional problem introduced by the transience of the data – normally, CPT row probabilities are computed from the complete data set. As we do not have access to the entire database at one time, Friedman and Goldszmidt [FS97] proposed that we instead store sufficient statistics (a count, in this case) about each row of the CPT in the following fashion: When the CPT is created, each statistic is zero; when we get new data, we increment the statistic representing each record and the total number of records that CPT represents. For example, if our CPT stored sufficient statistics related to attributes A, B, and C, to update the CPT upon seeing a record representing $A = a_i$, $B = b_i$, and $C = c_i$, the sufficient statistic

¹ A CPT is stored for each attribute in the dataset (aka node in the network representing that data set). For example, if our network was an edge from attribute A to attribute B, the CPT of A would store values of A and probabilities of each while the CPT of B would store the probability of values for B given values for A. The probability of $A = a_i$ and $B = b_i$ in the latter table may be called the “row” of $A = a_i$ and $B = b_i$ in this CPT.

for that row in the CPT would be incremented and the overall statistic for that CPT would be updated also. If we need the probability of a row, we can compute that value from the sufficient statistics.

Note that, as the search proceeds, CPTs corresponding to changes no longer on the frontier (due to a candidate network change) will be forgotten and new CPTs will be added to replace it – this results in some CPTs being based on more records than others. Because of this adding and dropping of CPTs (and the accompanying loss of information), we can no longer guarantee generation of an optimal Bayesian network given the data (as is the case when learning from a rereadable database) but, given that we have access to more data overall through this process than possible otherwise, the loss of optimality seems acceptable [FS97].

So the basic idea of our project is to alter a code base written to learn from a static database to read from a simulated streaming data source using the technique of maintaining a frontier (figure 1 (below) is a diagram of these processes). As mentioned above, this algorithm is most appropriate for situations where the data is transient (e.g. a web site) or where we cannot easily access the data more than once (e.g. large database with a poor connection and large records).

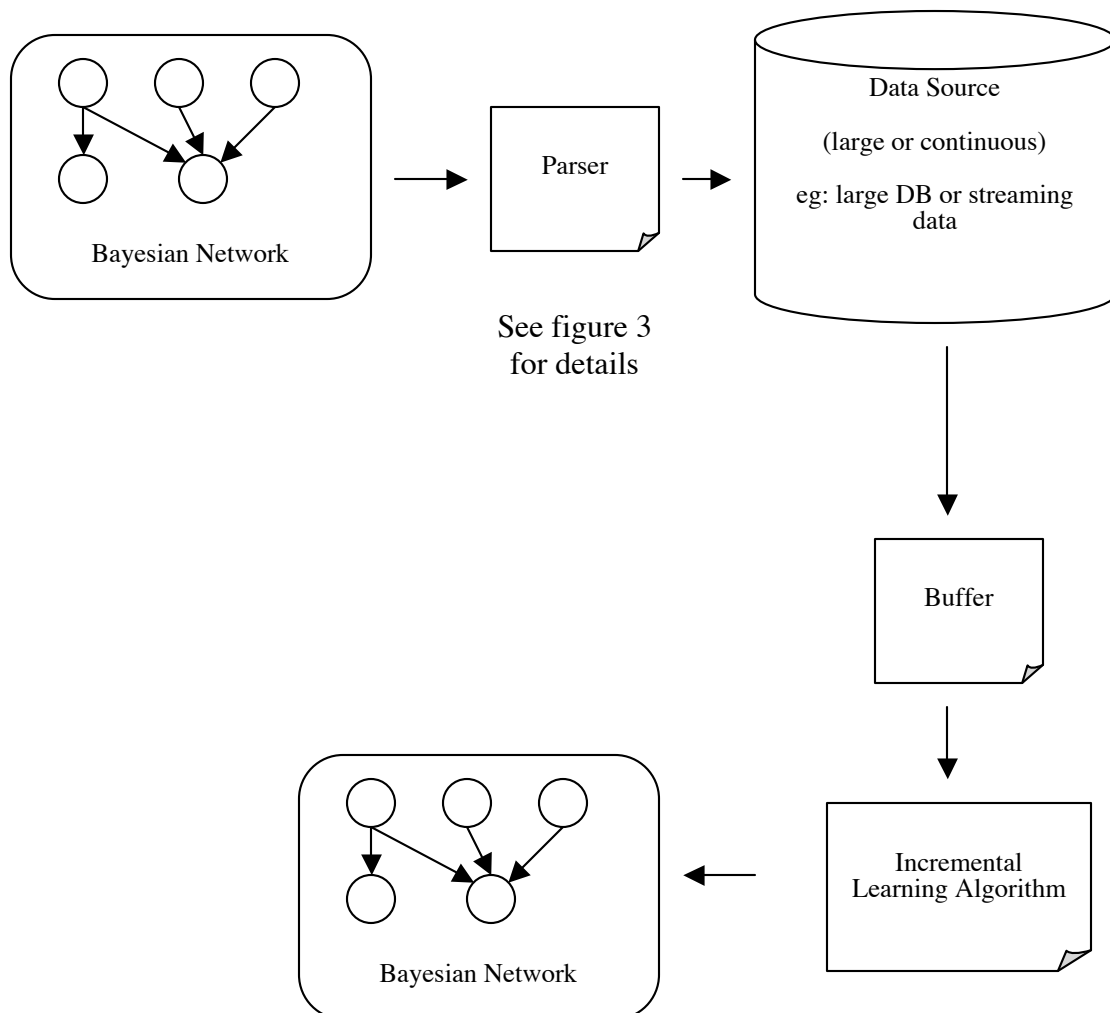


Figure 1:

Project Design

Because we started with a Java implementation of an algorithm to learn Bayesian networks from static data in text format, we had an inference algorithm already written to evaluate queries to a network and a framework for updating CPTs written in Java. As such, we decided to use Java to write the bulk of our code (in addition to JDBC providing a convenient way to access the data from a database). The following was the changes we needed to implement to get an incremental learning algorithm:

- Given a Bayesian network, generate a database table of values representative of that network. This was used to evaluate the accuracy of the learning procedure – if the procedure returns a network very close to our original network, we know that our procedure is working correctly. As this portion is only used for testing and didn't interact with other parts, we used Perl to write a quick-and-dirty processor to change a description file to a network that was then used to infer a dataset that was put into an Oracle database.
- Create a class (dataSource) that links the learning algorithm to the data set. This dataSource was implemented modularly with respect to the rest of the code to allow easy changing of the data source if needed. For our implementation, we put the data in an Oracle database and used JDBC to access it via dataSource.
- Change the scoring function used by the learning algorithm to MDL [Ris78]. Because we are interested in evaluating the ideas of Friedman, Goldszmidt, Lam, and Bacchus [FS97] [LB93], we thought it was most appropriate to use their scoring function. Also, a later paper by Lam and Bacchus [LB94] suggested that we might be able to keep track of the MDL scores of the frontier networks via simple calculations based on the candidate network, thereby avoiding the recalculation of the MDL score for each network when we compared them. We did not implement this specific method but instead calculated the MDL score for each network and edge that experienced a change due to the new data.
- Implement a system of tracking the CPT values and what the frontier networks are as well as how to update all of them (i.e. perform the search of the possible network space). In the original code, the CPT values were associated directly with a node in the Bayesian network but, in our case, we have a variety of networks so having a central repository for all the CPTs allows for quick updating of the parameters (the nodes in each network were changed to references to the CPTs in the set of CPT values). The frontier networks could be implemented as just a list of single changes from the candidate but when the candidate changes, updating all of these would be complex so we instead just created separate Bayesian networks for each of the frontier networks. This also allows us to possibly parallelize searching and data input more easily in the future.

We decided to use an Oracle database to store our data because it was available to us, commercially popular (so our code would work with fewer changes on other Oracle databases so it will be more widely applicable), and has level 4 JDBC drivers to allow for native Java access to the database.

System Implementation

The system consists of four packages: Datasets, Graphs, BayesianNetworks and searchengine (see figure 2 below). Datasets contains classes that abstract dataset management such as Instance, Attribute, Schema, Dataset (a Schema plus a vector of Instances) and a dataSource (which fills in the Dataset from an external source, e.g., a database, a text file in ARFF format, etc). Datasets also contains the discreteProbabilityDistribution class and a utility class.

The Graphs package contains two classes: Graphs and GraphNode. The first abstracts the most general case of a graph (with an adjacency list representation for the edges), with a set of GraphNodes and some utility functions such as cycle checking and checking for existence of edges and/or nodes (BayesianGraph, from the BayesianNetworks package, extends this class). The other class of the package, GraphNode, abstracts a graph node, which is the one who contains the parents and children vectors, that is, the edges of the graph, and offers basic functionalities to deal with the parent and children vectors, like adding, removing and counting.

GraphNode is extended by BNNode, the node used by BayesianGraph, which represents a random variable on one side and an attribute from the other. It therefore contains the corresponding attribute index in the schema and also a conditional probability table (CPT). For this project, we generalized BNNode so it actually does not maintain the CPT: this task is performed by cptSet in interaction with dataSource and the search engine. We also extended BNNode so it computes its own description length. Lam and Bacchus [LB94] is presented a localized algorithm for the description length such that the description length of the network is simply the sum of the description lengths of each node. We reflected that fact into our classes design, letting each node compute its own description length. We should also remark that the BNNode sends message to and from the CPTs in such a way that it only re-computes the description length if the CPT was changed, which happens after each read of new data from the data source.

The condProbabilityTable class in the BayesianNetworks package abstracts a CPT. It is represented by a multidimensional array, coded in ArrayD. It includes functionalities like updating the values of the table given a new instance (an object instance of the class Instance generated from a record). It also offers observer methods to retrieve the probability of a given instantiation of the conditioning attributes/variables, joint probabilities and marginal probabilities. The values in the table are updated when new data is read from the dataSource, which asks cptSet to update all its CPTs every time it is requested to read a chunk of new instances from the data source.

The last package, searchengine, consists of three abstract classes: Search, SearchNode and searchQueue. The first abstracts the search functionality, which consists of a main loop that continuously expands the candidate node, the first node in the search frontier (named OPEN in the code) and inserts the nodes resulting from the expansion back in the frontier. The nodes are inserted in the frontier in some ordering, either according to a scoring function (Best First Search), at the end (Depth First Search) or at the beginning of the frontier (Breadth First Search). In our case, we needed a best first search algorithm (with the description length as the score), as suggested by [LB93], but for generality we included the other two commonly used possibilities. In order to fully abstract the search class, we needed an abstract searchNode class, which should be later on extended for particular search algorithms. In our case, we extended Search into LBSearch and SearchNode into LBSearchNode. Finally, the frontier was also abstracted in a couple of classes, an abstract class searchQueue and a Vector like implementation. The idea was to eventually implement a priority queue for the frontier, which would be much more efficient, but time analysis of our algorithm showed that the bottleneck was in the retrieval of data instances from the DB instead of local computation, allowing some flexibility in this part of the implementation.

The program's starting point is the frontEndSearch class that has only one main method with command line input parameter management plus initialization and running of the search engine.

More detailed information on the packages and each of the classes can be found in the [documentation](#)². Note that the documentation contains links to the source code on the name of each of the class methods. Figure 2 (below) provides a graphical representation of the dependencies between the packages and the hierarchies inside them.

² The documentation is contained in the /doc folder of the project's CD.

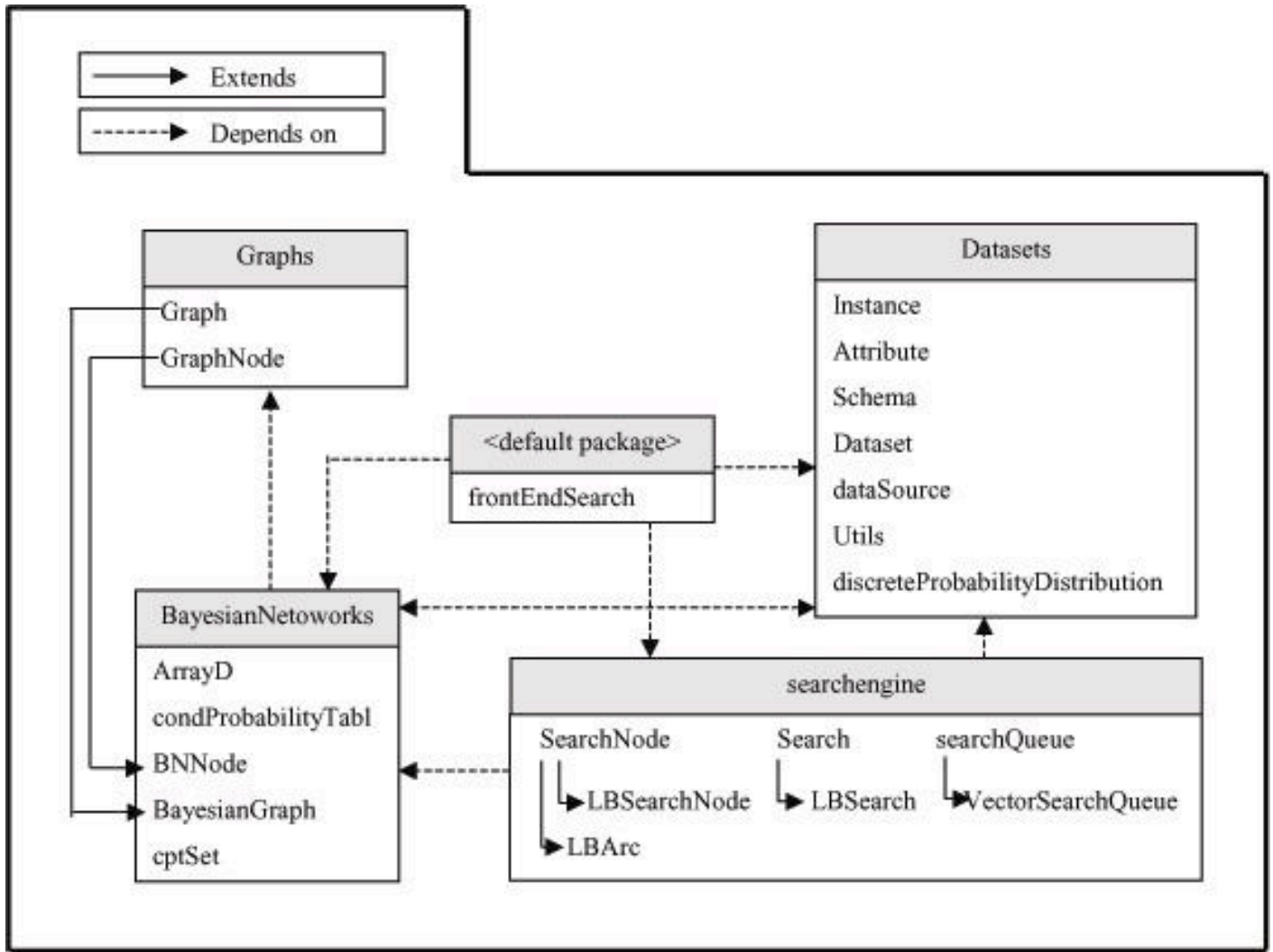


Figure 2 – Dependencies between packages and classes

Generation of the Datasets (aka Database Layout)

Existing Bayesian networks were used to generate the datasets “fraud” and “asia” for testing. The dataset generation phase consists of three main phases. In the first phase, a Bayesian network that is encoded as a BIF (Bayesian Interchange Format) file is parsed and the network is loaded into memory. The loaded network is then used to generate intermediate files that aid us in generating meta-data and the dataset. In the second phase, the meta-data (information about how many values an attribute can take on, how many attributes there are, etc.) is loaded into the database and a Java program is created. In the third phase, the Java program is used to generate the dataset, connect to the Oracle database via JDBC and load the dataset into it. Please see figure 3 for a schematic drawing of this process.

The Bayesian Interchange Format is one of the most widely used formats for encoding Bayesian networks. Each BIF file is a text file that has two main parts: one part encodes the structure of the network while the other describes the Conditional Probability Tables (CPT’s) associated with it.

We used PERL to write a rudimentary BIF parser because PERL is very powerful in processing text content. Once a BIF file is parsed and the network loaded into memory, the parser generates three files (the *intermediate* files) – a Java source file, an SQL file and a *names* file – that help us not only generate the dataset, but also load meta-data about the network into the database. The generated Java file contains code to build the network and the CPT's associated with it.

The SQL file that is produced contains a bunch of SQL constructs, which when executed load information about the network into the Oracle database. This information (aka meta-data) is stored as a relation. The relation holds information about all the nodes (attributes) present in the network.

The *names* file contains meta-data that is used by the Java program. This file contains one entry for each attribute present in the network. Each entry encodes the name of the attribute and the values it can take.

The generated Java source file is a program called Datagen.java. This program does all of the dataset generation. The program is best described by the pseudo-code below:

```

bdaYdS_2VSfSYW : x{ ~w2L` S_We?X[ ^Wb2` 2L[ ` f;
  A<2` 2{ †2` zw2†{ ' w2fx2` zw2vs^ s†w 2` f2t v2yW w† s^ w2<A
  : C; 2^aSV2x{ ~w

  : D; 2Tg[ ^V2Ts' w†{s, 2` w < f†}

  : E; 2Xad2wsuz 26[ ` ef S` UW2{ , 2s--2, f††{ t ~w2{ , †^ s, uw†
  Va
    : E@C; 2b†2O2[ , xw††w2b†ftst{ ~{ ^' 2fx26[ ` ef S` UW2x†f€2^ zw2Ts' w†{s,
      ` w < f†}
    : E@D; 26Uab[ W2O2: b†<` ; 2uf, { w† 2fx26[ ` ef S` UW
    : E@E; 2b%o26Uab[ W2s^ 2†s, vf€2, f†{ ^ { f, †2{ , 2^ zw2vs^ s†w 2: †w, †w†w ^ w
      t' 2%o{ ..%o>2†s, vf€~' ?yW w† s^ w2{ v†;

  W V?Xad
  W V?bdaYdS_

```

Placing the datasets at random locations in the dataset is the tricky part: we attach each instance with an ID that denotes the position of it in the dataset. Then all we need to do is randomize the ID and make sure that all possible IDs are covered (we used a Boolean array to accomplish this). Then the reader of the data set can read the IDs in order to get the generated sample in random order.

What follows are a couple diagrams of this process:

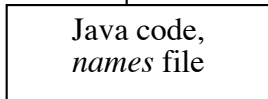
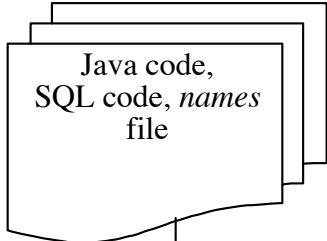
BIF stands for Bayesian Interchange Format. It is the most widely used format for encoding a Bayesian Network. The encoding stores the structure of the network and the Conditional Probability Tables.



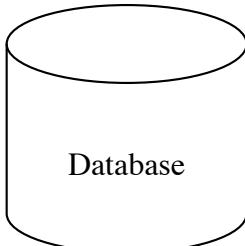
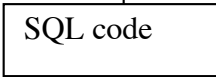
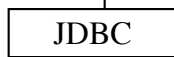
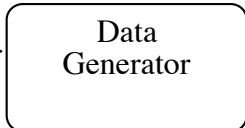
This is a rudimentary parser for parsing BIF files. PERL was used to code this because of its power in processing text content.

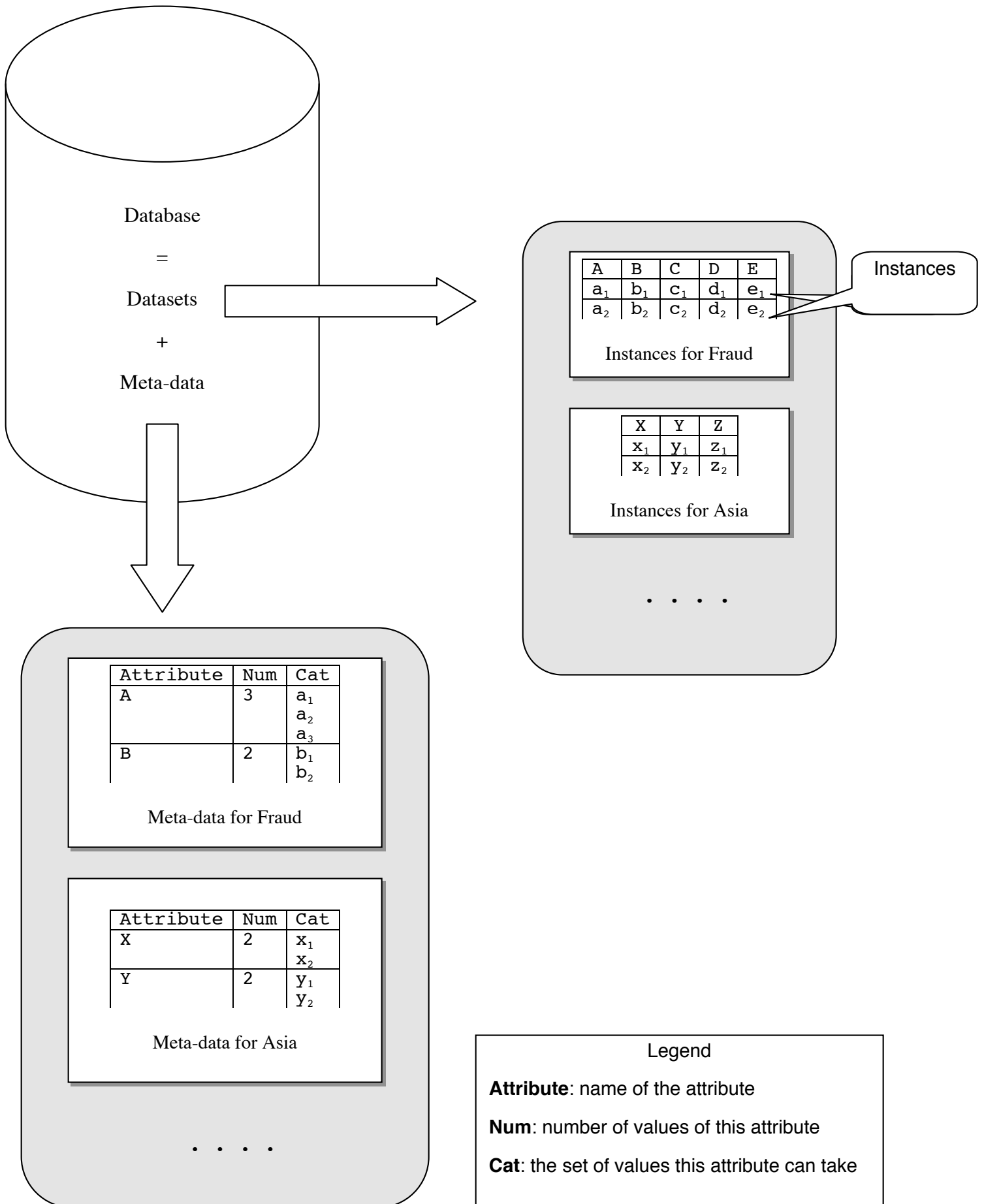


The BIF parser outputs a *names* file (that describes the attributes in the Bayesian Network), a Java source code file, and a file containing SQL constructs that is used for loading meta-data into the database.



The Data Generator is the Java source file that is used in generating the large datasets. It uses JDBC to communicate with an Oracle database server.





Results

We tested the following hypotheses with regard to how parameters in the search algorithm alter the accuracy of the output network:

1. Shrinking the buffer size (number of records read per iteration of the search algorithm) will result in longer running times for the same amount of data but a more accurate and complex network (as the search algorithm is run more times than in the case of a large number of records in the buffer and a finite data set to draw from).
2. The order in which the data is read will not significantly affect the candidate network produced.
3. Frontier size will mediate the effect of the order in which the data is read. That is, the larger the frontier of networks that we maintain, the less it will matter what order we read the input. This is because larger frontiers will enable us to effectively save a larger portion of the data for learning, offsetting input order.
4. The number of times that PAIRS is trained before developing a network will have a small but measurable effect on the shape of the resulting network. This is because training PAIRS more will enable the arc description lengths to be more accurate but will also use up some of the data that we learn the network from. If the source of data were truly infinite, we could train PAIRS for as long as we like but we are interested in what the trade-off is between PAIRS and network training.
5. Again, the size of the frontier is expected to mediate the effect of a lower number of times PAIRS is trained for the reasons given in hypothesis 2.
6. Altering the effective sample size of the input will have no effect on the resulting network.

Here are the results of our experiments (followed by analysis of them):

	<u>Fraud Edge Results</u>			<u>Asia Edge Results</u>		
<u>Buffer Size</u>	Correct	Close	Incorrect	Correct	Close	Incorrect
10	3	1	6	N/A ³	N/A	N/A
100	3	1	5	4	0	7
1000	2	0	1	3	1	2

Table 1 – Effect of Buffer Size

³ Due to poor timing, some tests did not run to completion so their results have been marked as N/A.

Frontier Size:	<u>5</u>			<u>25</u>			<u>500</u>		
<u>Read Method</u>	Correct	Close	Incorrect	Correct	Close	Incorrect	Correct	Close	Incorrect
1	3	1	5	3	1	5	3	1	3
2	4	0	3	3	1	3	4	0	3
3	3	0	3	3	1	3	2	0	1
4	1	3	3	3	1	1	0	2	4

Table 2 – Effect of Read Method by Frontier Size on Fraud

Frontier Size:	<u>5</u>			<u>25</u>			<u>500</u>		
<u>Read Method</u>	Correct	Close	Incorrect	Correct	Close	Incorrect	Correct	Close	Incorrect
1	4	0	7	3	1	10	4	0	5
2	4	2	13	3	2	6	2	1	3
3	5	2	5	5	1	8	4	1	6
4	6	2	10	4	1	10	3	1	3

Table 3 – Effect of Read Method by Frontier Size on Asia

Frontier Size:	<u>5</u>			<u>25</u>			<u>500</u>		
<u>PAIRS Runs</u>	Correct	Close	Incorrect	Correct	Close	Incorrect	Correct	Close	Incorrect
3	3	1	5	3	1	5	3	0	5
10	3	1	5	3	1	5	3	1	3
25	3	1	3	3	0	2	3	0	2
100	4	0	2	3	1	3	3	1	2

Table 4 – Effect of Number of PAIRS Training Runs by Frontier Size on Fraud

Frontier Size:	<u>5</u>		
<u>PAIRS Runs</u>	Correct	Close	Incorrect
3	6	1	5
10	4	0	7
25	6	0	9
100	4	2	12

Table 5 – Effect of Number of PAIRS Training Runs by Frontier Size on Asia

	<u>Fraud Edge Results</u>			<u>Asia Edge Results</u>		
<u>ESS</u>	Correct	Close	Incorrect	Correct	Close	Incorrect
0	3	0	2	N/A	N/A	N/A
10	3	1	3	3	1	6
250	1	1	4	3	0	5

Table 6 – Effect of Effective Sample Size

Notation:

Note: All measurements were in terms of edges correctly or incorrectly labeled. The “fraud” network had 4 edges with 4 attributes and the “asia” network had 8 edges with 8 attributes. Both data sets were generated with 100,000 instances each.

Table 1: The buffer size column indicates the size of the buffer used for that test, the “correct” column indicates the number of edges correctly added and oriented, the “close” column indicates the number of edges correctly added but not oriented, and “incorrect” indicates the number of extraneous edges (those not in the original graph). Results of a test with the “fraud” data set are followed by results from a test on the “asia” data set. The “fraud” set had a total of 4 edges and “asia” had 8 – if an edge is not listed as “correct” or “close,” the algorithm didn’t detect the relationship at all.

Table 2: The read method down the left side indicates which read method was used and the frontier size across the top indicates the size of the frontier used. The following different read methods were tested:

1. Read the records in the order they appeared in the database.
2. Read the records in the reverse of the order they appeared in the database.
3. Skip the first 1000 records, read to the end of the database, and then read those 1000 records.
4. Read every other block of records (in this case, records 0-99, 200-299, etc.) then, at the end, go back and read the skipped records

Table 3: Same as table 2 but over the “asia” data set

Table 4: Similar to earlier tables but the number of times PAIRS is trained prior to learning the network is listed in the leftmost column (note that PAIRS was also updated as the network space was searched).

Table 5: Same as table 4 but over the “asia” data set.

Table 6: As in table 1, both data sets are listed and the variable being tested (in this case the parameter effective sample size) is listed in the leftmost column.

I will now evaluate our hypotheses concerning various parameters given the tables above:

Hypothesis 1:

This hypothesis appears to be correct although we were unable to conclude test results where the buffer size was 10 because it took more than 8 hours to complete (compared to buffer size 1000 which finished in 20-30 minutes). If we compare the results of buffer sizes of 100 and 1000, there appears to be less accuracy (as the larger buffer runs fewer times) but also fewer extraneous edges and a much faster run time. Therefore, if we are more interested in a quick and dirty estimate in a short amount of time (or small number of records), larger buffer sizes seem warranted but, otherwise, a smaller buffer results in a more accurate network (the extra edges will mainly result in larger CPTs and longer run times for inference using that network).

Hypotheses 2 and 3:

As you can see in tables 2 and 3 (above), frontier size and read method clearly had an effect on the network produced. So, although our algorithm is designed to be robust to the effect of order that the data is input, it did bend surprisingly depending on the input order: in the “fraud” data set, read methods 1, 2, and 3 resulted in very similar networks but breaking up the data into different chunks (as done in method 4) resulted in a surprising loss of accuracy. The “asia” data set had similar results except that method 4 resulted in different but better results than the other methods.

However, frontier size did mediate this effect a bit – as you can see with all read methods in “fraud”, as the frontier size went from 5 to 25, more edges were chosen correctly or nearly correctly across all read methods. Moving from frontier size of 25 to 500 resulted in less accuracy in some cases but definitely less complex networks overall –we are unsure as to why accuracy did not increase uniformly but instead depended on the read method. The “asia” data set displays even more muddled results, giving an overall trend of decreasing accuracy and complexity as the frontier grew.

Hypotheses 4 and 5:

As Lam and Bacchus [LB93] learned their PAIRS description lengths from a static set of data, we were unsure of how to train them incrementally and the measure implemented was to train PAIRS a bit before learning the network. As we can see from table 4 about the “fraud” table, it appears that this caution was unwarranted: Regardless of how many times the PAIRS list is trained, we get similar results for a given frontier size. As there was no effect to mediate, the frontier size didn’t really affect the accuracy much in this case. The “asia” data set was a little more interesting in that it appears that complexity was increasing as the number of times we trained the PAIRS list increased but this may simply be due to the search part of the algorithm having fewer buffers of data to work with to eliminate unneeded edges.

Hypothesis 6:

Contrary to our hypothesis, as the effective sample size increased, the accuracy of the resulting network definitely fell off. As we can see in table 6, in the “fraud” data set most noticeably, we lose more and more correct and close edges as the effective sample size increases (this effect is not as pronounced in the “asia” data set). This effect is probably because increasing the effective sample size effectively tells the search algorithm that we’ve already ran X samples to get the empty network (where X is the effective sample size), resulting in the search being more inclined to stay at the empty network for longer than it would if the effective sample size were smaller.

One note not obvious from the tables – many edges produced incorrectly in the candidates did follow a causal path in the original Bayesian network. That is, if $A \rightarrow B$ is in the candidate network from our algorithm, it is nearly certain that there is a causal path from A to B (if not a direct link). This hints that our

data set was not large enough to generate the correct networks reliably (or else the paths would have been broken down correctly into direct links).

Also, it should be noted that each Bayesian network that we tested is only one of a class of Bayesian networks that are statistically indistinguishable. Specifically, there is another network that “fraud”’s Bayesian network is equivalent to and 11 other networks that the “asia” network is equivalent to. Many of the edges listed as “close” above may actually be correct given the MDL method (while some are incorrect) – each “close” edge was clearly part of the underlying undirected graph but its may or may not have been essential.

In any case, our tests on these two data sets are insufficient to provide any sort of proof as to the effectiveness of our algorithm. Several more data sets with more and less complex distributions would have to be run (most notably, other networks that are closer to complete networks than “fraud” or “asia”) in order to truly evaluate accuracy.

Conclusions

Given that most networks identified all the correct edges in “fraud”’s network and a strong majority of the edges in “asia”’s network, we are confident that our algorithm at least identifies these effects. What our algorithm found more difficult to address is how to reduce the number of incorrect edges so that the “true” network shape of a real distribution will easily emerge to the viewer. However, we believe that removing the extra complexity will depend primarily on the frontier size, buffer size, and the size of the data set, the former two of which are resource-controlled in the area that this algorithm would be used best (as record size and number of attributes would be large so this directly influences how large the frontier can be and how many records we can hold in the analyzing machine’s buffer).

In the production of this algorithm, we came up with several possible areas for future work. We realized early on that the algorithm proposed by Lam and Bacchus [LB93] would need to be altered to learn incrementally, leading to our implementation of how to remove arcs (Lam and Bacchus only allowed transposition or addition) and PAIRS (as a circularly linked list – Lam and Bacchus simply had their algorithm conclude when it ran out of elements in PAIRS). It would be interesting to try other implementations of how to handle PAIRS (such as by reading backwards through PAIRS when we get to the end until the beginning) to see if time or performance improves.

Another question is: What if the underlying distribution changes over time? Friedman and Goldszmidt [FS97] assumed that the underlying distribution would not change over time but this strikes us as unrealistic. For example, as an e-business changes the product that they’re selling via the web, the distribution of pages that customers look at will change. We believe that our algorithm should handle changes in the distribution (as the method of keeping a frontier makes our search more sensitive to new data

than other methods) but it would be interesting to see how long it takes for the network to recognize a change under various conditions.

In addition to changes to increase accuracy or test accuracy under certain conditions, we also believe that there are more efficient ways to perform incremental learning. One change would be to parallelize data input and searching for the best network – right now the search algorithm only runs for one step before getting new data but it could run until it required more data, at which point it would retrieve whatever is in the buffer of the data input method. We also could more efficiently store the networks in the frontier or work on more efficient data structures throughout our algorithm - we left some code in our project somewhat inefficient or did not analyze efficiency because the time to read the data outweighed the computation by several factors.

The easiest alteration to our code would be to increase flexibility even more – we could produce a standard plug-in format for the dataSource class such that any input method could be used (rather than just an Oracle database, as our code is setup now). Once this is done, we could enable the user to switch data sources in the middle of processing or write a wrapper to read data from many sources.

References

- [FS97] N. Friedman and M. Goldszmidt. Sequential Update of Bayesian Network Structure. *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, 165-174, 1997.
- [LB93] W. Lam and F. Bacchus. Using Causal Information and Local Measures to Learn Bayesian Networks. *Uncertainty in Artificial Intelligence*, 243-250, 1993.
- [LB94] W. Lam and F. Bacchus. Using New Data to Refine a Bayesian Network. *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 383-390, 1994.
- [Pea86] J. Pearl. Fusion, Propagation, and Structuring in Belief Networks. *Artificial Intelligence*, 32:245-288, 1986.
- [Ris78] J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14:465-471, 1978.

Appendix A – User’s Guide

The set of packages created by this project are designed for use by a Java programmer to implement learning of a Bayesian network incrementally. As such, we think it should be sufficient to just provide [documentation](#) of the various functions of our classes.

Appendix B – Team List and Responsibilities

Facundo Bromberg – Original Bayesian network learning code (written for Machine Learning course, altered in Introduction to Artificial Intelligence course), changing the scoring function to be MDL-based, implementation of tracking frontier networks, search algorithm code (as described in [FS97]), System Implementation part of the report (as well as editing the rest of it), generation of the report CD structure, and generally helping other team members with understanding his original code.

Brian Patterson – Reading data from a database, updating and maintaining the CPT set, providing Facundo with a way to access the data, running a series of tests for the Results section, Introduction, Project Design, Results, and Conclusion parts of this report (as well as editing the rest of it), and general help with implementing the search algorithm.

Sandeep Yaramakala – Getting Bayesian networks from a public repository into a format understood by Facundo's inference algorithm, using that inference algorithm and his own code to generate data instances automatically, designing the database schema, deciding which database system to use, Data Generation part of this report (as well as editing the rest of it), and general help with implementing the database system.