CSC242 Third AI Project: Uncertain Inference

Jason Natale

4/12/16

Background

Bayesian Networks are useful tools for making probabilistic inferences. Because of this, they become extremely valuable in many problems posed by AI which deal with uncertain knowledge. A Bayesian Network is defined over random variables and the relationships between them. Each random variable can be interpreted as a node in a directed acyclic graph (DAG). The links between nodes are generated based on what a single variable's influencers are. For example, if your uncertain world can be characterized with random variables A, B, and C, and C is influenced by the values of A and B, then A and B will be connected to C as its parent nodes. Based on these incremental inferences, a table is also defined at each node which represents the probability of a given outcome for that variable, based on the values of its parents.

When performing a query on a Bayesian Network to discover the probabilistic distribution across a variable's domain, based on logical truths already derived, the set of random variables can be broken into three different categories. There is the query variable (X), which is the variable whose distribution is being asked. There are the evidence variables (E), which have already been observed and are concluded to be representing a certain value within their own domain. Lastly, there are unobserved variables (Y), which have unknown values. A common query to a Bayesian Network is P(X | e), which results in a vector describing the probability for the query variable to be each value in its domain based on the evidence variables provided. This project is based upon that conditional query.

This query can be mathematically solved with the expression:

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha \sum_{\mathbf{y}} \prod_{i=1}^{n} P(x_i \mid \mathit{parents}(X_i))$$

In English, this means that the probability distribution across a variable X, based on all evidence variables e, can be solved by processing through all variables of a network in topological order. Starting from the root nodes and working one's way through the graph, products of conditional probabilities of a node's value, given its parent's values can be multiplied together, with unobserved values domain's summed together to produce a result.

Exact Inference

Introduction:

Processing through the formula described above, Exact Inference makes no approximations or assumptions. The result is calculated, as described above, with all values found in the random variable's distribution tables. Because of this need to work systematically through many variations of the network, Exact Inference can become costly with large networks.

Inference by Enumeration Algorithm:

This algorithm, as outlined in Figure 14.9 of AIMA, performs Exact Inference. My implementation of this procedure, ask(), takes inputs of a Bayesian Network, query variable, and assignments of evidence variables. This method works by initializing a distribution (mapping of domain values to representational probabilities for a given variable), and looping through each possible domain value of the query variable. For each domain value of the query variable, I produce a new assignment (mapping of random variables to their corresponding domain values) which is based on the evidence variables, with the addition of the query variable being assigned to its iterated domain value. With this unique assignment, I perform the Enumerate_All() method, and add the returned probability to the distribution created. Once each possible domain value of the query variable has been iterated over, the resulting distribution is normalized (meaning the probabilities are proportioned to sum to 1), and the distribution is returned.

Enumerate_All() is based on pseudocode from Figure 14.9 of AIMA. This is a recursive method which iterates through a network, stringing together products and summations of conditional probabilities at each node. Enumerate_All() receives as input the Bayesian Network being utilized, a topologically sorted list of variables in the network, and the evidence variables of the query along with a description of the query variable. This topological order is important due to the fact that this method takes the first variable off of the list to analyze and perform calculations on. After taking the variable from the list, it is then observed whether or not the current variable is given a value and is "evidence". If so, there is a recursive call made to the Enumerate_All() method with the rest of the list, and the result of that procedure is multiplied by the value found from the distribution table of the current node which tells the probability that the current variable is the value assigned to it. This product is then returned. If the variable isn't assigned yet, it processes through a for loop, being assigned to each possible value within its domain and evaluations of that probability are calculated in a similar manner as to if it had been assigned already. Each newly calculated result is summed together and returned.

Approximate Inference

Rejection Sampling & Likelihood Weighting

Introduction:

When working with large networks, exact calculation of inference becomes nearly impossible to do. Thankfully, methods have been developed to accurately estimate the conditional probabilities which exact inference provides us. Rejection Sampling and Likelihood Weighting are two examples of Monte Carlo problems. These algorithms both work by generating a variant number of random samples, testing those samples against the network at hand, and producing probabilistic conclusions in a timely manner.

Rejection Sampling:

This approximation algorithm, as defined in AIMA chapter 14, works by generating a number of sample assignments for the network at hand. As stated earlier, an assignment is simply a mapping of random variables in a bayesian network to their assigned value. After each sample is created, it then is tested against the evidence variables provided to the algorithm. If this sample is not compatible with the evidence variable assignment, meaning that random variables take disjoint values, the sample is ignored. Otherwise, it is then checked to see which value the query variable is representing in that set. A running tally is maintained of all the instances where a domain value of the query variable is found within a sample. Once the desired number of samples have been created, the distribution (a mapping of domain values to probabilities) is normalized and returned. Normalization simply means to add up all the evaluations across a distribution, and then divide each by the sum achieved. This always works out to create a distribution summing to 1.

The method rejectsamp() performs this algorithm in my project. I track the instances of a sample containing a certain query variable's domain value with the integer array counts. To generate a random sample of the variables across the network provided, I call the function GeneratePriorSample(). This method loops topologically through all random variables. With each variable, it then loops through each possible domain value, creates a new assignment object based on the previous assignment developed, and adds in the looped random variable assigned to its looped domain value. With this, the probability of that assignment in the network is calculated and added to a distribution set. Once all domain values have been accounted for, the distribution created is normalized and a random value from 0 to 1 is then calculated. I then loop through my distribution and determine if my value is less than that distribution set's probability plus my running total of percentages seen. If it is less, I assign the random variable to that domain value. Otherwise, I continue through the loop checking the domain values. Because the probabilities add up to 1, using the random value allows me to proportionally act on the propensity that the variable is a certain value.

Likelihood Weighting:

This algorithm is also defined in AIMA chapter 14. I implement this algorithm as likelihood(), with inputs of a Bayesian Network, query variable, assignment of evidence variables, and an integer determining the number of samples to be generated. Much like the previous algorithm, likelihood weighting iterates through samples being generated. Unlike rejection sampling though, samples which do not conform to the evidence are not generated. WeightedSample() performs the procedure of creating these samples. Alongside each sample produced, is a weight. That weight describes the likelihood at which that sample would exist, and has an impact on the distribution created when weights are added together, rather than making standard +1 increments for a value seen. Once all the weights representing values seen across all samples for a given query variable are added together, the distribution which holds the mapping of domain values to sums is normalized and returned.

WeightedSample() creates a sample fitting the constraints passed in, as well as a weight to represent the likelihood of the sample existing. This data is bundled into an object, called a WeightedSample, which is returned. Processing through all random variables of the network in a topological order, assigned and unassigned variables are separated. Those which are already assigned, are placed in the sample with the given value they represent. The weight is then multiplied by the probability that the random variable would represent that value. Unassigned variables are looped through all possible domain values, determining the probability that each are represented. Ultimately, those values found are normalized and a value for the random variable is chosen the same way as in rejection sampling, with a random value.

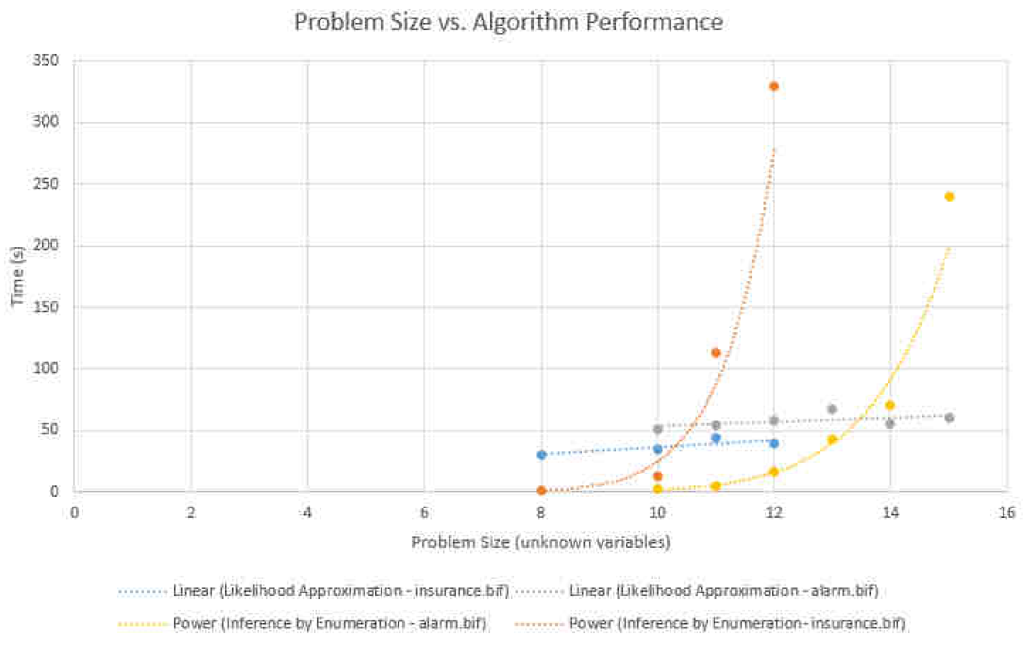
Network Analysis

I have performed tests on all the Bayesian Networks provided in the examples folder

Networks Utilized:

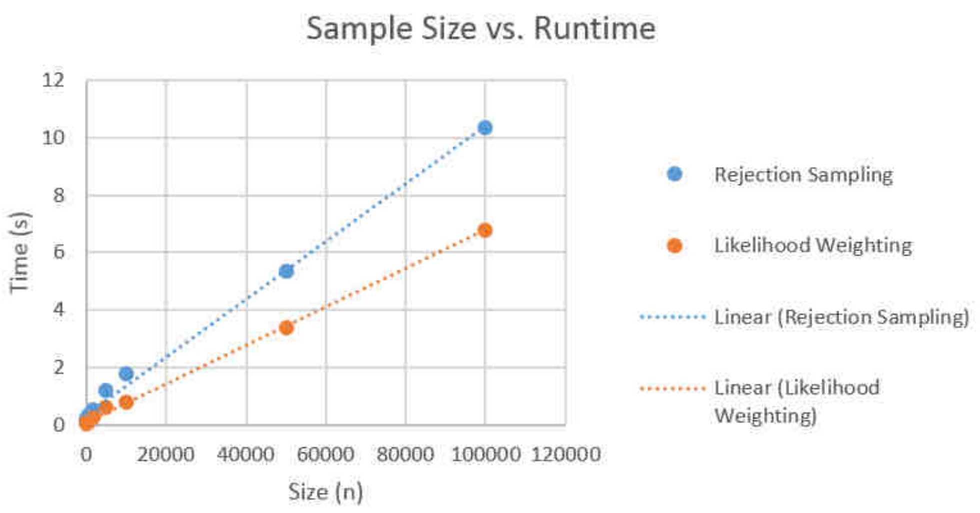
Results shown represent queries to the networks of aima-alarm.xml, alarm.bif, and insurance.bif.

Comparison of Algorithm Performances:

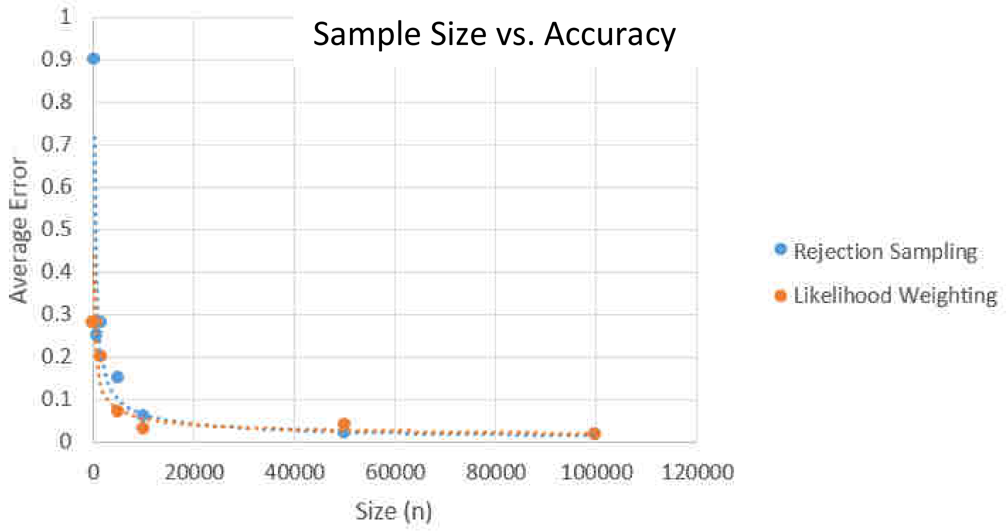


The intractability of Inference by Enumeration (Exact Inference) can be seen quite plainly in the graph above. As the problem size increases, the time it takes to determine a distribution with that algorithm increases exponentially. On the other hand, Likelihood Approximation processes in linear time, allowing large problem spaces to be solved more efficiently. I did not include results from Rejection Sampling because it was too rare to find samples which were consistent with the evidence. This is a weakness of Rejection Sampling, which forces it to only be applicable in smaller networks.

Relationship between Sample Size and Performance:



This is another graph showing the linear relationship between sample size and the amount of time for an approximation to be solved.



This shows the inverse relationship between sample size and the accuracy of the results. These data points are modeled off of tests from aima-alarm.xml with the query P (B | J = true, M= true). For small sample sizes, the accuracy was wildly off. This can be seen with the tests of sample size equal to 100. As sample sizes increase incrementally, performance is initially vastly improved. From there, the improvement between sizes diminishes until it will ultimately converge to zero.

Conclusion:

From the tests above it can be easily deduced that exact inference works with smaller networks, but fails miserably as the network size grows (runtime grows exponentially). Approximations work very well with larger networks, but the right sample size is needed to balance the costs of runtime with the benefits of accuracy. I found Rejection Sampling to be much less valuable to work with than Likelihood weighting, as its accuracy was typically lower for the same provided sample size, and it suffers the fate of rejecting the majority, if not all samples, with large networks.

References:

I used the AIMA textbook, as well as sample Bayesian networks as illustrated in the examples folder for this project.