Assignment 2

Written by Zitong Li, z5189688 and Alind Vats, z5248806 for COMP9418.

Representation

Class Declaration

In order to represent Bayesian networks, we thought of what a class declaration of the network would consist of. We thought that 3 dictionaries— representing the network, probability values and outcome spaces — would be sufficient to capture necessary information of a Bayesian network. These are the class members:

net (dict)

This is a dictionary that represents a node and all its children. The key refers to the name of a node and the value is a list of nodes that are its children.

factors (dict)

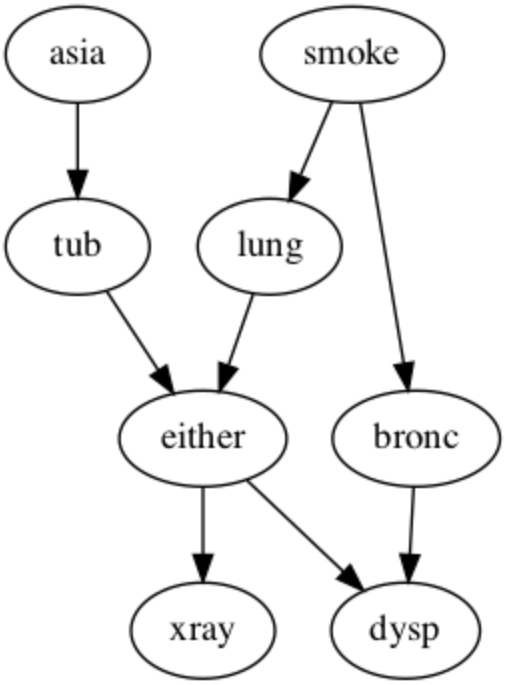
This is a dictionary that contains a node as the key, and its domain and the joint probability values of the different combinations of values/outcomes the node and its parents can take on.

outcomeSpace (dict)

This dictionary contains a node as the key and a tuple of all its possible outcomes as its value.

Class Functionalities

Insert and Remove Node: The insert function takes in the class, name of the node and outcome space and adds the name to net and outcome space to outcomeSpace. It also checks if the node being inserted already exists, and if it does, the function doesn't allow insertion.
Connect and Disconnect Edge: This takes in the name of the parent and child node as its arguments, checks if they exist in net, and connects them if they're not already connected. The disconnect function also takes the name of parent and child nodes as arguments and checks the existence of the parent and the child node in net, and deletes the edge by removing the child from the list of children nodes of a parent node.



Specify probabilities of node: This function takes in a node, an array specifying the probability values of the different outcomes corresponding to a node and optionally an array where the user can specify a list of parents. It computes the domain by adding the nodes in the parent list and the node specified and computes the joint probability table and maps it to a set of outcomes the domain nodes can take on.

Saving Bayesian network to file: This function takes in a file name and writes the outcome space of the different nodes and the domain and probability values to the specified file.

Load Bayesian network from file: This function takes in a filename, opens and reads in the outcome space of the different nodes and the domain and probability values from the specified file.

Pruning and pre-processing techniques for inference

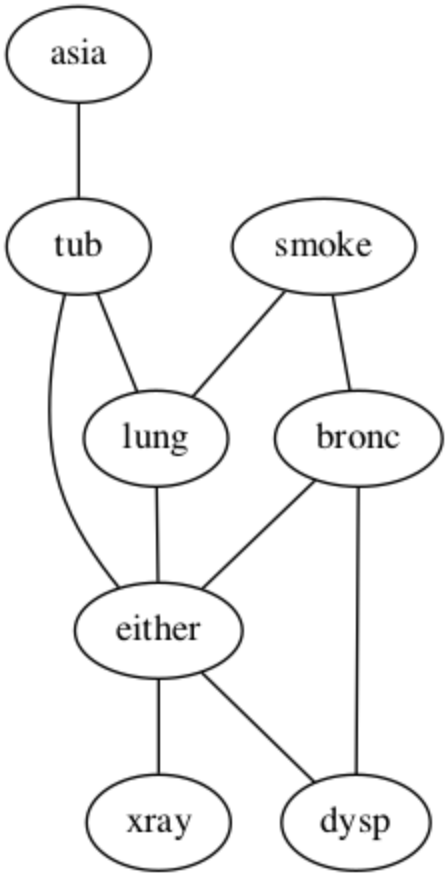
The prune function takes in two inputs-- query, a list of strings representing the query variables and evidence, which is a dictionary where the key is the node and the value is the outcome of the node. This function iteratively prunes out necessary nodes and edges in different loops and returns the new graph.

The update function helps update the value of a node with the specified value. It takes in a 2D dictionary factor, which contains the factors of a node, the node we want to update, the value we want to update the node with and outcomeSpace, which is a dictionary containing the outcome space of all nodes. The function returns the updated factor dictionary with the new domain and table.

The spread function helps figure out all reachable nodes from a source node. It takes in the graph, a dictionary, and a list of strings indicating the node where we start the spread. The function returns a list of strings, indicating the nodes that are reachable from the source input by the user.

The make_undirected function returns an undirected version of the graph-- it takes in a directed graph and returns its undirected equivalent. The transpose_graph function takes in a graph and returns its transpose.

To get an elimination order, we first need to convert our bayesian net into a moral graph and that is what function “moralize” do, here is the result after applying moralize to the graph.



There is a function that also retrieves the variable elimination order given a graph. There is a function that takes in the node to be eliminated and returns the fill-in edge count. There is a function that takes in a graph and returns its moralized equivalent. There is a function that connects all the nodes in a graph. There is also a function that shows the moralized graph.

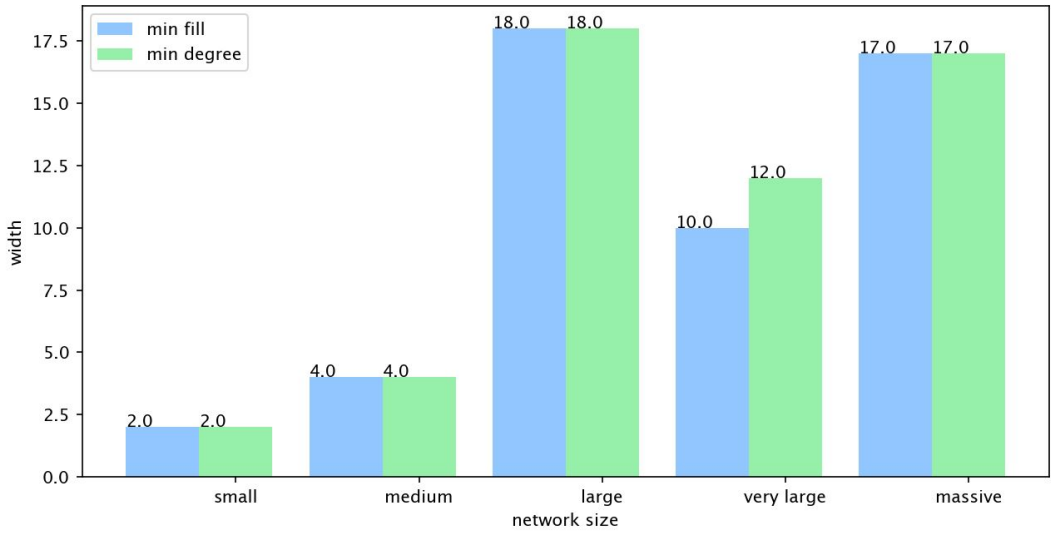
Elimination ordering Benchmark

There are two different heuristic used in elimination ordering: Min-degree and Min-fill.

1. Min-degree: Choose a variable with the fewest dependent variables (neighbors in the graph), if some nodes have the same min degree, than choose the one with less fill-in edges.
2. Min-fill: Choose vertices to minimize the size of the factor that will be added to the graph (add smallest number of fill-in edges), if some nodes have the same min fill-in edges, than choose the one with less degree.

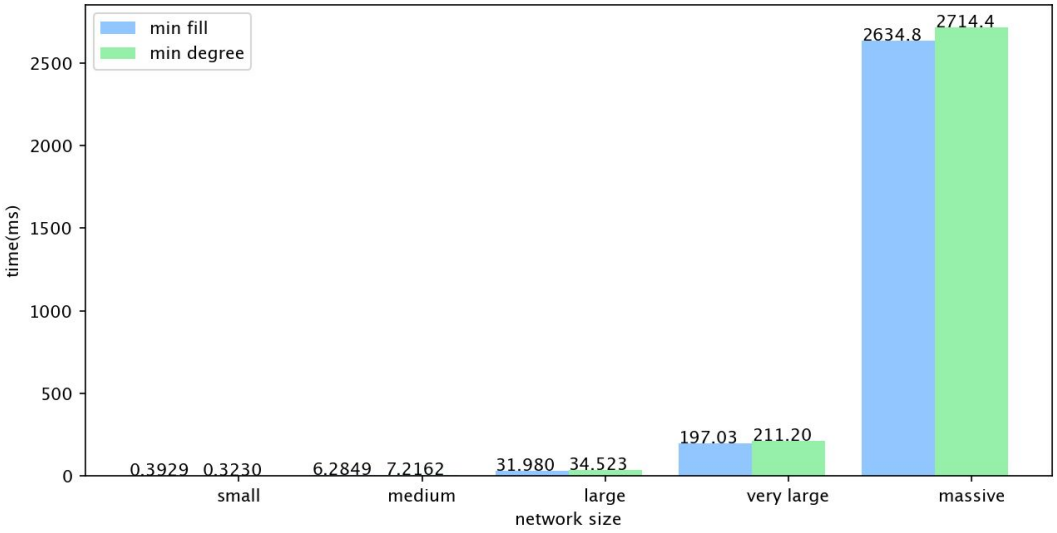
Both heuristic are tested on five different scale networks, here are the results:

1. The width of the eliminaiton order



From the result, most networks have the same width of elimination order except for the very large one, where min-fill heuristic is slightly better.

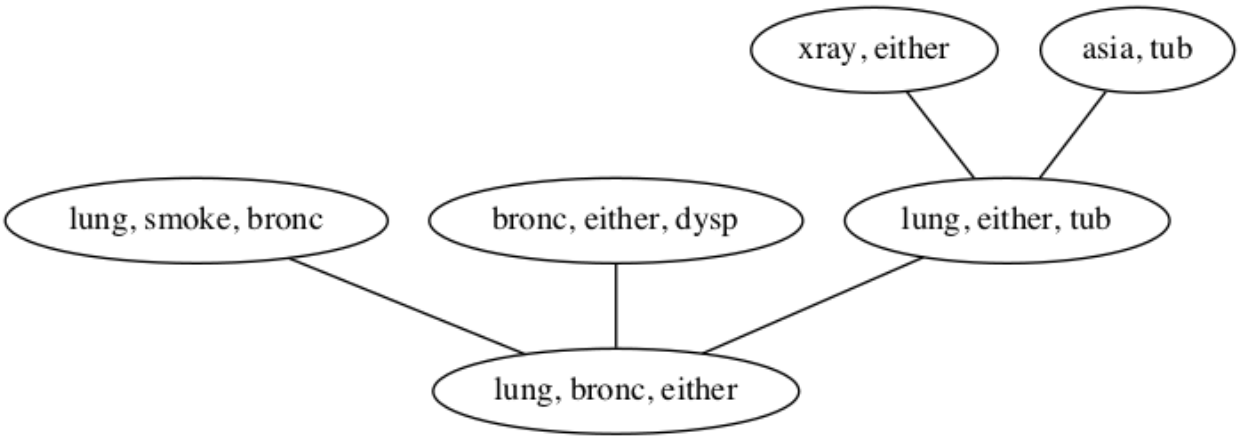
2. The time cost



Time cost highly depends on the scale of the network. In general, these two hearistic of ordering have very close performance.

Exact inference

There is a function, to_jointree, that takes in a graph and returns its jointree equivalent. The function takes in a moralized graph and a variable elimination order. The first step in the function is to create appropriate clusters. Nodes are iteratively appended to clusters, the underlying data structure of which is a list of sets. After construction of appropriate clusters, a JoinTree object is returned.



There is a JoinTree class used to represent jointrees. Its constructor takes in clusters, which is a dictionary representing the cluster, factors, which is a dictionary representing the factors, and outcomeSpace, which is a dictionary representing the outcome space. The class also consists of messages, a dictionary of messages contained. This class also contains several functions that assist in the representation, creation and manipulation of jointrees.

There is a function that shows the jointree; there is one that adds a node to a cluster, which is a set. There is also a need to merge two existing clusters, which creates a new cluster containing the merge of the two clusters input into the function and removes the two individual clusters from the clusters list. Add_cluster is similar but instead of merging, it adds an existing cluster to another.

The remove function removes a cluster from the list of clusters. There is a function that joins two factors and returns a new factor which is a combination of the two factors that the function takes in. First, we need to determine the domain of the new factor. It will be union of the domain of the two input factors. We also eliminate the repetitions. The product iterator is used to create all the combinations of values as mentioned in the outcomeSpace.

The marginalize function takes in a factor to be marginalized and a variable to be summed out. The function returns a new factor with the updated domain. We iterate over all possible outcomes in the outcome space. The updated factor is then returned.

The queryCluster function takes in a list of query variables and a node in the elimination order whose cluster contain the query variables. The function returns the factor with the marginal of the query variables.

The getMessages function consolidates all the messages and returns a dictionary of all messages. The input to the function is the root node. A depth-first search is done for each neighbor of a node. Push and pull functions are implemented to help us with this. The pull function takes in the root node and the node we came on from the search and returns a factor with a message from that node to the root node. This function utilizes depth-first search and recursion to do so. The push function does the same but with outgoing messages. It also takes in the root node and the previously visited node and utilizes depth-first search and recursion to return a factor with a message from previous node to root.

Approximate inference - Gibbs sampling

The Gibbs sampling function takes in the number of samples, number of chains, list of query variables and a dictionary of query evidence. The function returns a list of dictionaries, where each one is a sample that contains the node as its key and the node's value as its value. Get_acceptance computes the acceptance probability of a sample. The node waiting to be assigned, previous node and the current value waiting to be assigned. It returns the acceptance probability.

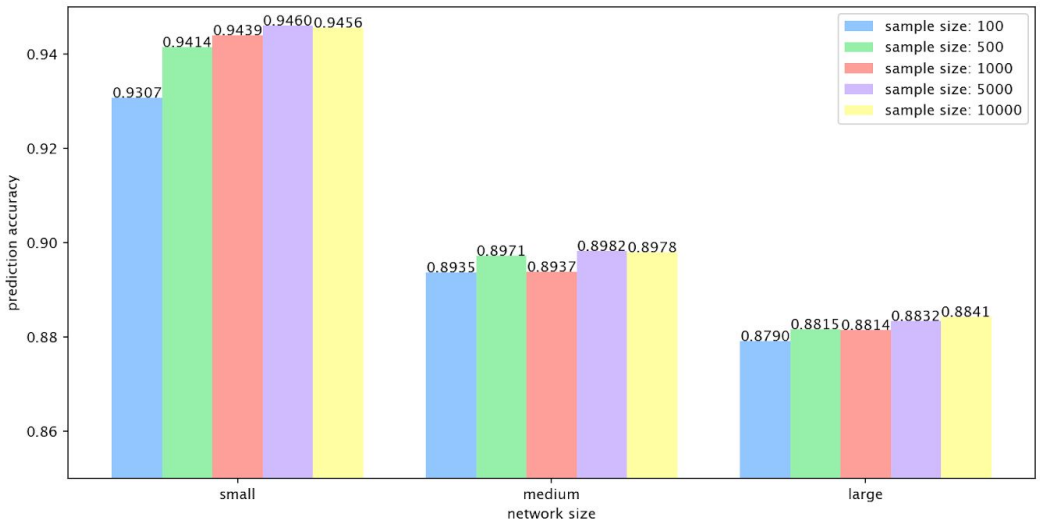
The burn_in function takes in the number of chains, the number of samples used to test if chains are mixed and evidences of the query and generates chains and keeps sampling until mixed. It returns a list of mixed chains. Sample_once samples a value from the outcome space of a node when a node is passed into the function.

Gibbs sampling accuracy Benchmark

Due to the large scale of networks in "very large", "massive", we can't get an exact inference in a feasible time so we only compare the network in "small", "medium" and "large".

For the network in each category, select one variable as query each time get the approximate inference by sampling(with different sample size) and exact inference by using jointree, compare them and compute an accuracy score and do it for every variable in the network so that we can get a mean accuracy for this network.

By choosing 5 different sample size in range of 100 to 10000, we summarize a bar chart as followed. The y-axis is the accuracy of prediction based on samples and x-axis is network size.



From the bar chart we can make following conclusion:

1. When the network size is fixed, increasing the sample size will increase the accuracy.
2. When the sample size is fixed, the larger scale the network is, the less accurate the prediction will be.