COMP9418 Report

zID 5202067

For most of this assignment, I did not do much of error checking. It is expected of the user to provide the correct input; otherwise there would be an exception and/or errors.

Task 1: Representation

For task 1, I used BIF file format to build the Bayesian network. The code I made would parse the texts and look for certain keywords to create the nodes and probabilities. The representation of the Bayesian network graph is stored as a list of nodes. The node itself keeps track of the nodes that points to it and the nodes that points from it. These nodes are also stored as neighbors, in order for it to be used to form clusters. The filled edge later on would also be stored as neighbors, but not in_nodes or out_nodes. There are various housekeeping functions that is used to manipulate the graph, such as deleting nodes, removing edges, etc. The removal and addition of edge itself doesn't change the probability inside the node. What changes it is a different function that uses variable elimination. When changing the variable, the number of nodes pointing to it in addition to the node itself becomes the query to the graph. Once the query is answered, through grouping and division, we can get the new probability for the given node.

In addition to basic functions, I create a function as well to make a morality graph out of the network. This is done through looking at each nodes' parents and assign them as neighbor if they are not already. This does not affect the output file or network in any way since the output file looks at the variable in_nodes and out_nodes, which are explicitly declared connection between parent nodes and their children.

The difficulty that I have is in creating a variable elimination function. Since it is a bit confusing for me at first because the eliminated variable causes the probability to be weird. It turns out that I eliminated the variable prematurely and it ended up making the data nonsensical. Other than variable elimination, task 1 seems rather straightforward in that it is parsing texts to create a graph.

Task 2: Pruning and Pre-processing Data

For pruning the nodes, I would prune the nodes before creating the moral graph. I decided not to prune the edges because it would not affect the running time significantly in my opinion and it would disrupt some functions. In other words, in order to make sure no error could've happened due to the missing edge, I decided to forego it entirely, instead cutting the time by simply not counting the given evidence later instead. The node pruning removes unrelated nodes and leaf nodes to simplify the query.

The resulting graph is shown to be much smaller and therefore much more manageable to run, especially when the original graphs contain more nodes. The effect shows exponential effectiveness, in that larger graphs would feel the impact of pruning more relatively compared to smaller nodes. This is because the problem could've been solved under a smaller graph, making traversal towards the entire graph unnecessary and straining in terms of complexity.

Table 2.1 Running time of random query in seconds using jointree cluster

Dataset	No Pruning	W/ Pruning
Asia.bif	0.055130958557128906	0.05285239219665527
Alarm.bif	0.26468324661254883	0.07280707359313965
Hailfinder.bif	0.25208353996276855	0.12523913383483887
Munin.bif	Memory Error	Memory Error

This graph shows that the pruning affects the query. Obviously, the shape of the graph is a huge factor in how long the query will run. The memory error might be because the huge number of nodes in the join tree. While it worked for most query, the largest query wouldn't work in the join tree graph.

An issue I encounter when pruning is if the query requests parent nodes that have no connection to each other through morality graph. This would cause the pruned nodes to have no edge at all, which results in no graph since they are simply individual nodes. While it is not addressed in my implementation, the solution is quite simple in that the resulting marginal would simply be the two probabilities joined.

For testing purposes, I created 3 heuristics: the min-degree, min-fill, and min-fill with min-degree tiebreaker. Logically, it seems that querying through jointrees and variable elimination should have the same result, given that the cluster and order of elimination is correlated. The join tree caused memory error given the large size, so I decided to use variable elimination instead.

Table 2.2 Running time of random query in seconds using variable elimination

Dataset	Min-fill w/ Min-degree	Min-degree	Min-fill
Asia.bif	0.048872947692871094	0.05086326599121094	0.04986381530761719
Alarm.bif	0.19647002220153809	0.20248174667358398	0.21542835235595703
Andes.bif	7.611605882644653	11.459443092346191	5.439654588699341
Hailfinder.bif	0.33809804916381836	0.3660154342651367	0.33809518814086914
Munin.bif	42.18917107582092	29.545294046401978	42.644965171813965

It shows that the different heuristics were not necessarily better than the other. It also looks like the two min-fill heuristics have similar outcome, which is consistent with how it works, being that the second condition is merely a tiebreaker. It seems that the better heuristic is highly dependent on what the query is and what the graph looks like.

These tasks are rather straightforward in my opinion. The pruning node is simply looking for nodes to be deleted that doesn't affect the query. The problem with combining join tree with this is that the join-tree that is already too big wouldn't help in a lot since building the nodes itself already caused the error.

Task 3: Exact Inference

For the tree, I created a dictionary of nodes. It stores the tuple of name of the nodes as key and the nodes as value. The tree has functions to print the probability of individual clusters and the image of the graph. The graph has two edges for each nodes because it represents the node going back and forth for messages. The four operation functions for the nodes is also stored in the tree. Each operation is mostly not error checked; it expects the user to give the correct clusters and values.

The jointree creation starts with creating a list of clusters. The clusters are taken from neighbors. Since neighbors in a morality graph denotes a family, it makes sense to use this as a way to take in clusters. The clusters are created through the elimination order by eliminating nodes until the last node is removed. Afterwards, the list of nodes is connected to edges based on the lectures. Instead of having a separator as shown in the lecture, I simply store the separator as keys in the edge. To ensure that the resulting graph is a tree, I only add an edge if there is no path to that node already, ensuring that the resulting edge won't cause a cycle. This works because if there is a cycle from cluster A to cluster B, then there is a series of clusters from A that can connect to B while maintaining the running property of the jointree.

The jointree operations is stored in the jointree graph. When adding a cluster, it is simply adding a leaf node in the graph. Deleting a node would remove a leaf node in the graph. Adding a variable simply checks surrounding nodes for a cluster with that variable before creating a new cluster with the combined variables and then adding edges to the cluster. Merging cluster works the same way except the 2nd node is deleted as a result of the merging.

When querying, the message is not computed entirely during run-time. It will only generate message if it is not yet created and the query requests it. This is to ensure that if a query is not run, a message does not need to be computed. A root node for the query is selected by finding a cluster that contains the query entirely, otherwise, it will select one at random. The evidence is used to remove irrelevant data.

The query through jointree and variable elimination has exactly the same result. This is expected because jointree is similar to variable elimination. However, when running multiple query with similar path, the jointree is more efficient. Given that the number is not computed every time the query is requested, it will run faster after subsequent similarly structured queries. On the other hand, the join tree structure takes a lot of memory. For example, each individual cluster stores dictionaries for edges and messages as well as a dataframe of probability for that particular cluster. Each tree contains a dictionary of clusters. The resulting representation causes particularly huge dataset to result in memory error.

When using different heuristics, the resulting tree would be different. The clusters formed would be similar in the most parts while some might change because the order of removal is different. The connecting edges between nodes would also look different since the order of clusters would be different as well. Again, different queries work better with different heuristics. The general rule of thumb is that using min-fill with min-degree heuristics would give better results most of the time compared to the other heuristics.

When using pruning, the resulting tree would be smaller. When the dataset is large, pruning would significantly reduce the running time it takes to construct the tree and the message propagation from the root nodes throughout the graph. However, the smaller tree can have adverse effect as well. Some queries might not be able to be answered from the resulting smaller tree from pruning the network. This means that saving the messages becomes a bit redundant given the unreliability of the smaller tree. Because of this, while it could be faster at computing smaller number of queries given a tree, when there is a huge number of queries using the same tree, pruning might not give as much advantage in the long run.

Task 4 Iterative Join Graph Propagation

The join graph, which is a bethe cluster graph is constructed by first constructing the clusters. I used the same method of constructing clusters to the jointree algorithm. I also create individual attributes as clusters, with their name as key instead of tuples. Each attributes is connected to the associated clusters.

I also keep track of the edges in the cluster graph instead of in each individual nodes. During IJGP loop, it will loop through these edges and send the message accordingly. The first element in the edge is always the individual attribute and the second element is always the cluster. I made it this way because the message that the cluster send to message would need to be grouped and using the key to group is convenient. The message sent is also different since the individual attribute always send an empty probability table, since the factor of it is 1.

The algorithm always converges on the datasets. This might happen because the graph could make a tree. Since join tree is a form belief propagation as well, it seems that the iterative join graph would also converge given the structure. Considering the convergence and similarity to the join tree, the resulting marginal probabilities are also the same to the join tree counterpart.

On large datasets, again the algorithm doesn't seem feasible. It would take exponentially longer given the number of clusters and the number of attributes. Since it takes a lot of iteration before the message actually converges, the time it needs to solve a query isn't viable.

On certain datasets where a tree structure might not happen, there isn't a guarantee that the message would converge. Especially on convoluted graphs, there is no way to ensure the convergence on polynomial time. When this happens, I think a timeout should happen and the result after that timeout should give a good approximation of the marginal probabilities.

In contrast to jointree, the Bethe cluster wouldn't be affected as much when using different heuristics. The only thing that would change when using different heuristics would be the clusters. Despite the change in the clusters' contents, the message would converge at the same time anyway since the update would happen in each iteration in a similar fashion regardless of heuristics.

Pruning, on the other hand, would help Bethe cluster exponentially as well. Through pruning, the resulting factors would be significantly less. This would create less clusters and less attributes in general. Despite the lesser number of clusters, the number of iteration it would take before converging shouldn't be too different compared to the clusters before pruning. However, because of the less edges overall, the time it takes to complete an iteration is significantly reduced.