Name: Parashara Ramesh	**Student Id**: A0285647M	**NUSNET Id**: E1216292

Part 1 (Importance Sampling)

My implementation consists of the following steps:

1. **construct_graph_from_factors**: This function takes proposal_factors and evidence as input, and it outputs a NetworkX graph representing the Directed Graphical Model (DGM) along with the topological order of variables. If a factor contains observed variables, factor_evidence is applied to that factor. However, if the factor is univariate and the variable is observed, it is excluded when constructing the graph. This approach ensures that the resulting graph does not include any observed variables as nodes or edges.
2. **_sample_step:** This function is executed 'num_iterations' times and samples a state value for each variable in the proposal factors following the topological order. When sampling a state value for a specific variable, it focuses on a single slice of the factor where its parent nodes have predetermined states. This allows the use of factor_evidence for observed parent variables to extract the relevant slice. The probabilities from this slice are then used with np.random.choice to sample a state value for that variable. In this manner, samples are obtained for each variable in the proposal distribution.
3. **calculate_w_values_for_all_samples**: In this function, sampled states from the proposal distribution, along with the given evidence states, are used to calculate 'p' values from the target factors. Instead of finding the joint distribution, probabilities from each factor table are multiplied based on the specific states of each variable. Similarly, 'q' values are obtained from the proposal factors. Using both the ‘p’ and ‘q’ values the 'r' values are computed. These 'r' values are then normalized to derive 'w' values across all iterations.
4. **get_probs_for_each_query_node_state_configuration:** This function constructs a dictionary where the keys represent unique state configurations from the proposal graph variables, and the values represent cumulative weights for those configurations. Ultimately, these values sum up to 1.
5. **create_out_factor_from_state_probs:** This function uses the dictionary obtained in the previous step to construct an output factor with values corresponding to the state configurations' order, as determined by the helper function get_all_assignments.

Part 2 (Gibbs Sampling)

My implementation consists of the following steps:

1. **construct_factor_graph**: This function takes as the input the nodes and edges given to construct a networkX graph which represents the DGM
2. For each node’s factor **factor_evidence** is first applied with the given evidence and then finds its markov blanket using a function **‘markov_blanket’**.Then I marginalize all other variables present in this factor which do not correspond to its markov blanket to get the conditional probability for that node given its markov blanket
3. A loop of burn_in + num_of_iterations is run to sample values for each variable that many times, but the only difference is that we throw away the sampled states obtained from the burn_in period
4. **_sample_step:** In this step, gibbs sampling approach is used to sample states for each node for a particular timestep. For each node_i, we utilize the states of nodes from node_1 to node_i-1 obtained in the current timestep, while the states of nodes from node_i+1 to node_N are taken from 'input_sample_states,' which correspond to the previous timestep's samples. The actual sampling is done using **np.random.choice** on the probabilities associated with a particular slice corresponding to using **factor_evidence** by providing the states of all the other nodes as evidence.
5. **calculate_conditional_prob:** In this function, I tally the occurrences of state configurations obtained in all timesteps. Dividing this count by the total count of all configurations provides the probability for each specific state configuration. This approach is akin to a frequentist method, counting configuration occurrences and normalizing. These probabilities are then used to construct the output factor. The order of state configurations is determined with the "get_all_assignments" helper function, and the output factor's values are populated using the probability dictionary.