A
Project Report
on

An Independent Study of Practical Issues in Uncertainty Management

In partial fulfillment of the requirements
for the award of the degree
Of

MASTER OF COMPUTER SCIENCE

By
Harshvardhan Rajpurohit
200485741

Department of Graduate studies
University of Regina
3737 Wascana Pkwy,
Regina, SK S4S 0A2

August, 2024

Abstract

This project focuses on developing a dynamic algorithm that transforms a Bayesian network into an Arithmetic Circuit. The approach involves parsing a Bayesian Interchange Format (BIF) file to extract the network structure and parameters. An elimination order is calculated, and with it, variables are removed one by one from the Bayesian network, creating potentials and storing the information in the process. The conversion process integrates indicator and probability nodes to construct the Arithmetic Circuit, enabling efficient probabilistic inference. This transformation uses the compact representation of Sum-Product Networks to enhance computational efficiency and scalability. The Arithmetic Circuit preserves the probabilistic relationships and allows quick, precise inference, making it appropriate for use in probabilistic reasoning.

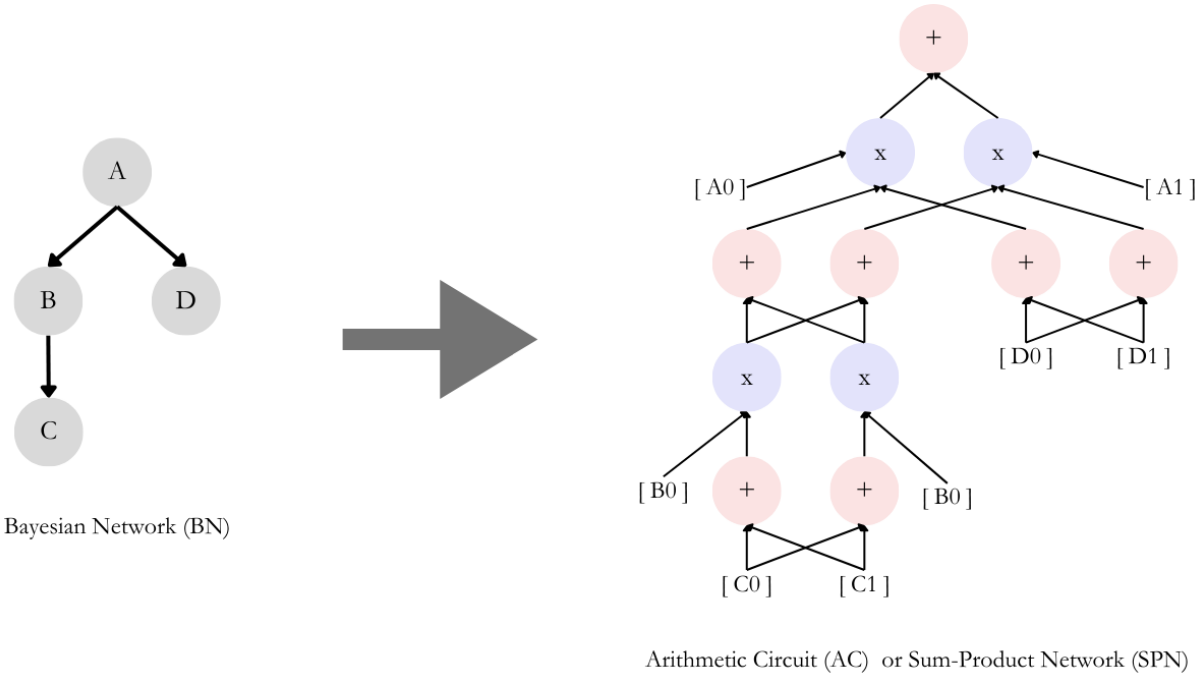

Figure 1: Conversion from BN to AC

Contents

Table of Figures

Glossary

Below are the commonly used terms in the report.

1. *Dictionary*: A data structure in Python that stores key-value pairs. Each key is unique and acts as an identifier for its associated value, enabling efficient data retrieval. Dictionaries are used to organize and manage data by mapping keys to their corresponding values.
2. *List*: A data structure in Python that holds an ordered collection of items, which can be of different data types. Lists are mutable, allowing for dynamic modification of their content. They are commonly used to store sequences of data elements and support operations like indexing, slicing, and iteration.
3. *Node*: A fundamental unit in data structures such as linked lists, trees, and graphs. It typically contains data and may include references to other nodes. In the context of Bayesian networks and arithmetic circuits, a node represents a variable or a value and can have connections to other nodes, reflecting relationships or dependencies.
4. *Marginal*: It refers to the process of summing out or integrating over certain variables to focus on the remaining variables in probabilistic models. In Bayesian networks, marginalization is used to compute the probability distribution of a subset of variables by eliminating others, providing insights into specific aspects of the model.
5. *Bucket*: A collection of items grouped together based on certain criteria, often used in algorithms to organize and manage data. In the context of Bayesian networks and variable elimination, buckets are used to group variables and their associated potentials, facilitating efficient computation and inference.
6. *PGMPY*: Probabilistic Graphical Models using Python is an open-source Python library for working with probabilistic graphical models, such as Bayesian networks and Markov networks. It provides tools for creating, manipulating, and performing inference on these models, making it a valuable resource for researchers and practitioners in the field of probabilistic modeling and machine learning.

Introduction

The project aims to be an independent study of practical issues in uncertainty management, by developing a python program that performs the transformation of a Bayesian Network (BN) to its equivalent Arithmetic Circuit (AC). Bayesian networks are powerful tools for representing and reasoning about uncertain knowledge, but their computational demands can be high. Arithmetic Circuits, on the other hand, offer a more efficient way to handle these computations. By transforming them into Arithmetic Circuits, we aim to achieve more efficient probabilistic inference.

This project explores the fundamental concepts of both Bayesian networks and Arithmetic Circuits. The journey begins with understanding the Bayesian Interchange Format (BIF) files, which store the structure and parameters of Bayesian networks. The core of the project is to convert these networks into Arithmetic Circuits, integrating indicator and probability nodes to facilitate fast and accurate probabilistic inference. The project aims to utilize several python libraries to establish a smooth conversion process that upholds the statistical dependencies of the original Bayesian network, allowing for quicker and more scalable inference. This code is designed to work consistently across different scales of Bayesian networks, whether it's a simple network with just two nodes, the well-known Asia Bayesian Network, or even larger and more complex networks. This adaptability ensures that the software can be widely applied, thus making it a valuable asset for the domains of machine learning and probabilistic reasoning.

In the upcoming sections of the report, we will delve into comprehensive explanations of the functionalities of each of these components in the project. Code Structure overview explains the overall working of the project, and elaborates about the functions that guide all of the process for the conversion of a BN to an AC. The next section dives into a deeper understanding and explanation of the functionalities implemented in the project that aids in the conversion process. Libraries like NetworkX and PGMPY are put to use in a wide variety of tasks such as marginalizing nodes, determining elimination orders, extracting information from BIF (Bayesian Interchange Format) files, thus simplifying many of the complex tasks involved in working with probabilistic graphical models. Apart from the conversion, this section also explains the usage of Graphviz library to visually represent the resulting AC. Lastly, additions to the current system and its features are discussed in the conclusion section, explaining in-depth about the future improvements to the project.

Project Methodology and Overview

Development Approach

For this project, we adopted the CLEAN architecture to ensure a modular and maintainable codebase. CLEAN architecture divides the system into distinct layers with clear responsibilities, enhancing organization, scalability, and testability. Our approach involved separating the core business logic related to Bayesian Networks (BN) and Arithmetic Circuits (AC) from application-specific rules and data handling. By organizing the code into multiple files, each serving a specific purpose, we maintained the individuality and clarity of each component. This modular structure allowed for efficient management and development, with each layer being independent yet seamlessly integrated. The use of CLEAN architecture facilitated a scalable and maintainable system, where new features could be added with minimal disruption, and testing could be conducted at various levels to ensure reliability and robustness.

Code Structure Overview

The Python code for converting a Bayesian network into an Arithmetic Circuit is organized into several key files and components. Each file and component are designed to handle a specific part of the transformation process, ensuring modularity and ease of understanding. The code is organized into five separate files, and each file is named based on its specific function and purpose within the project. The code follows the principles of CLEAN architecture, which divides the functionality into different parts. This approach not only makes the code easier to read and understand but also aids in debugging and tracing the flow of execution. The function names are designed to give a brief overview of their specific actions. By dividing the code into various functions, we can easily reuse them based on their specific purposes and how they are utilized in the overall process.

To execute the process, the *main.py* file is crucial as it contains the main function which calls various functions from imported files. One important function within this main file is "`create_sum_product_network`". This particular function takes the Bayesian network (BN) as input, which has been converted to a specific format, and then proceeds to call different functions to convert the BN network into the corresponding arithmetic circuit.

The *BN_functions.py* file incorporates the PGMPY and NetworkX libraries and encompasses several crucial functions. One of the key functions is the "`read_bn_file`" which is designed to extract the details of the Bayesian network from the BIF file. Another essential function is "`marginalize`," which computes a marginalized distribution across the parent variables of the input distribution. The "`eliminate_node`" function is responsible for removing a node from the buckets created in previous iterations. Additionally, the

"`evaluate_arithmetic_circuit`" function determines and returns the final value by assigning appropriate values to the indicator nodes based on the input evidence.

The file named *structure_functions.py* includes a set of functions that handle the modification of the arithmetic circuit's structure. One of the key functions in this file is "`create_node`" which is tasked with generating a dictionary of a node that essentially represent an individual node within the arithmetic circuit. Additionally, this function calculates various statistics related to the nodes, such as the counts of sum and product nodes, the total number of nodes, and other necessary metrics.

In order to facilitate the conversion of keys in a dictionary, flattening multi-dimensional arrays, or converting list items to a specific type, a *helper.py* file has been created. This file contains the necessary helper functions to streamline these conversion tasks.

Following the transformation of the Bayesian Network (BN) into an Arithmetic Circuit (AC), the file *plot_graph.py* is utilized. This file includes the function "`plot_graphviz`" which plays a key role in visualizing the arithmetic circuit by leveraging the Graphviz library.

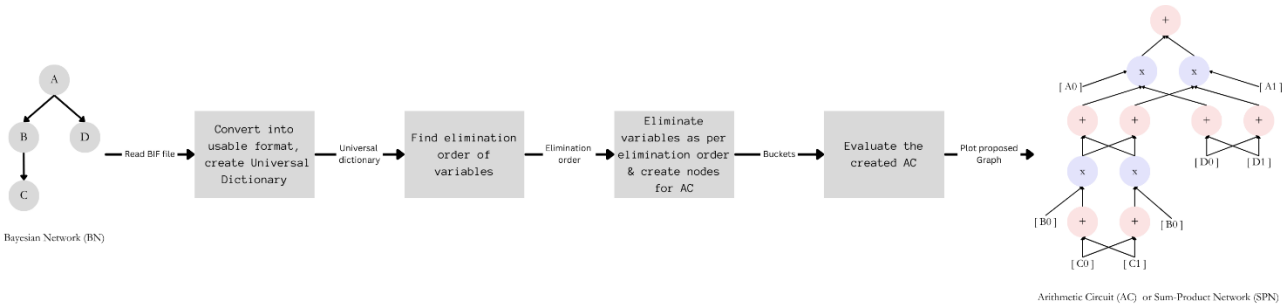

Figure 2: Process of conversion from BN to AC

Brief Overview of Code Components

In this upcoming section, you'll find comprehensive descriptions of each step involved in the conversion of a Bayesian Network (BN) to its corresponding Arithmetic Circuit (AC). There are 5 major steps that guide the entire conversion process, beginning with file reading up until culminating in inference on the newly constructed arithmetic circuit.

In an arithmetic circuit, there are four distinct types of nodes: sum, product, indicator, and probability nodes. This circuit is essentially a rooted directed acyclic graph (DAG) where all the leaf nodes are classified as either indicator or probability nodes, while the internal nodes are exclusively sum or product nodes. To facilitate implementation, a standardized structure for representing an individual node within the circuit was established.

I. Reading and parsing the BIF File

A BIF (Bayesian Interchange Format) file is a file format used for representing Bayesian networks (BN). BIF files store the structure of the Bayesian network and the conditional probability tables (CPTs) associated with each node. The first step in the process of conversion is reading and parsing this BIF file using the PGMPY library. The BIFReader class is a specialized tool designed to facilitate the reading and loading of BN from files in the BIF format.

The `read_bn_file` function found in the *bn_functions.py* file utilizes the BIFReader class to parse and retrieve the Bayesian Network (BN). Following this step, the next significant task is to validate the BN model. This is achieved through the `check_bn_model` function, which is responsible for verifying the correctness of the BN model and will raise an exception if any issues are found. It's also worth noting that the output format of the `read_bn_function` function is currently different and requires some refining to ensure it meets the necessary standards.

In order to convert this into a useful format, another function namely `get_bn_graph_nodes`, is implemented in the same file. This function converts the input BN into a dictionary format where the keys are variables of the BN and the values are dictionaries. These dictionaries follow a consistent structure, where the keys are states, values, and parents, and the values are obtained from parsing the BIF file, providing the respective value for the corresponding keys.

In the conversion process, a universal dictionary representing the BN is established. This dictionary contains keys that correspond to the variables of the BN. The values associated with these keys are dictionaries themselves, following a structure where the keys represent all possible values of the variables in the BN, and the corresponding values represent the probabilities. These formats are show in the image below:

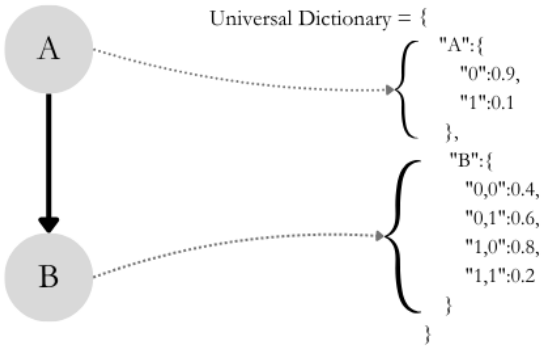

Figure 3: Structure of a Universal Dictionary

II. Finding Elimination Order and Topological Order of Nodes

Variable elimination is a fundamental algorithm used for exact inference in Bayesian networks. It systematically removes variables from the network to compute marginal probabilities, which are the probabilities of interest for specific variables while considering all possible values of other variables. The elimination order in this algorithm for BNs is crucial as it dictates the sequence in which variables are removed from the network. The choice of elimination order can significantly impact the computational efficiency and memory usage of the algorithm.

The VariableElimination class in the PGMPY library is a powerful tool for implementing the variable elimination algorithm, which is used for exact inference in Bayesian Networks (BNs). This class offers a range of methods for computing marginal probabilities, conditional probabilities, and MAP (Maximum A Posteriori) queries in an efficient manner. To leverage this class effectively, a function called `find_elimination_order` has been included in the *bn_functions.py* file. This function is designed to utilize various ordering techniques to determine the best elimination order for a given BN. Specifically, the function can generate elimination orders using four different techniques: Min-neighbours, Min-weight, Min-fill, and Weighted-min-fill. These techniques may produce distinct or similar elimination orders, each with its own advantages. Moreover, the function also computes the reverse topological ordering of nodes in the BN using the NetworkX library, which offers a function to topologically sort the nodes of the given model. This provides a total of five elimination orders for use in the conversion process.

III. Building the Arithmetic Circuit

Before we delve into the logic of converting from a BN to an AC, it's important to grasp the representation of a node in the final graph. The image below displays the keys of the dictionary representing a node.


```
{
  "type": " ",
  "value": 0,
  "references": [ ],
  "node": " ",
  "variable_value": " ",
}
```


Figure 4: Structure of a node represented in Arithmetic Circuit

The dictionary has 5 different keys and each of their explanation is given below:

- **Type**: The value associated with this key indicates what type of node the dictionary represents. The possible values are sum, product, indicator, and value. The "Value" type highlights that the current node has a probability value associated with it and is to be used accordingly.
- **Value**: The value associated with this key indicates the probability value for the node. Only "Value" type nodes will contain a numerical value associated with this key, for every other node, its value would just be None.
- **Node**: This key indicates the variable in the BN for which this current node has been created.
- **Variable_value**: This key indicates the specific value or state of the variable associated with the node. This value is crucial for defining the context in which the node operates.
- **References**: This key contains a list of dictionaries, each representing other nodes that this current node is directly connected to or depends on. These referenced nodes are the direct child nodes of the current nodes. Since "value" and "indicator" type nodes do not have any children, the value of this key for such nodes would be an empty list.

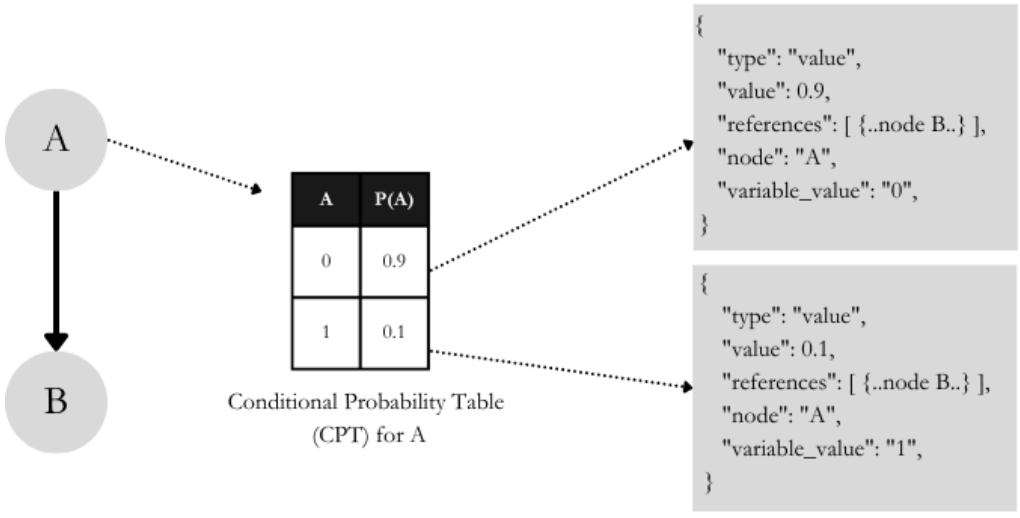

Figure 5: Representation of different value nodes of a variable from Bayesian Network

In the context of variable elimination in BN, "buckets" are a conceptual and organizational tool used to manage intermediate computations efficiently. They help in systematically eliminating variables and combining factors to compute marginal probabilities or conditional probabilities. This project incorporates the use of such buckets to implement the variable elimination algorithm and thus convert the BN to an AC by removing nodes from

the BN while simultaneously adding their respective representations and connections in the AC. The project follows a certain pattern for these buckets, for ease of usage as well as representation. The pattern can be depicted by the contents of the image below:

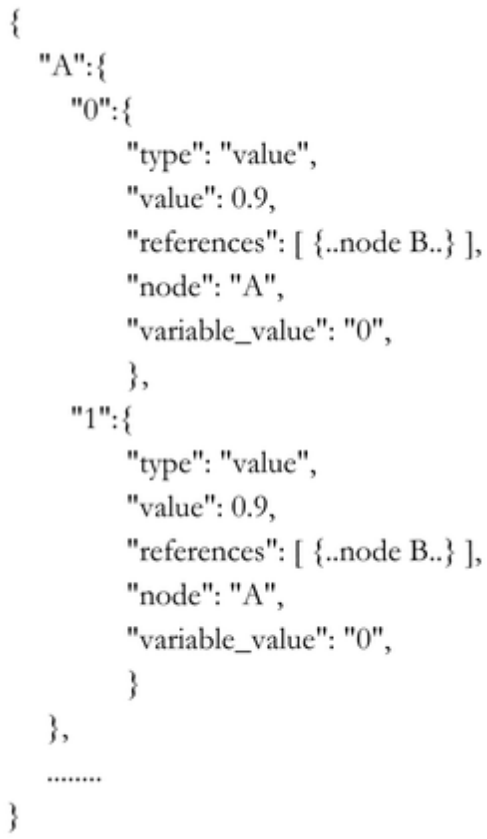

Figure 6: Structure of a buckets

Briefly, the conversion process can be understood as eliminating every node, as per the elimination order, by creating its level 1 product nodes using the indicator and value nodes and then eliminating this current node by calculating the marginalized distribution over every other variable involved with the current node and buckets containing this current node. Once eliminated, the buckets are updated with the resultant distribution. After every node is eliminated, the final dictionary obtained in the buckets would be the AC representation of the given BN.

The conversion process involves following the reverse topological order of nodes for their elimination. Within the *bn_functions.py* file, there is a function called `create_product_nodes` which is responsible for generating the first-level product nodes connecting the "Indicator" type nodes to their corresponding "value" type nodes. This function also determines whether the current node is a leaf node. If it is a leaf, the function calls the `marginalize` function to return a marginalized distribution over the parent

variables of the current node. If the current node is not a leaf, the function returns a representation of the current node's distribution with the level 1 product nodes. After examining the distribution that was returned, the buckets are then modified according to the pseudo code provided below:

Initialize buckets variable with empty dictionary
for each node in **elimination_order**:
- Create product nodes for the current node.
- Retrieve **marginalized nodes**.

- *If* **no marginalized nodes**:
  - Calculate potentials.
  - Eliminate current node and update buckets.
else:
  - for each **marginalized node**:
    - If node is not in buckets, add it.
    - If node is in buckets, create and update product nodes for associated buckets.

Figure 7: Pseudo code for populating buckets

The approach used in this conversion involves the selection of a node from the BN according to a predetermined elimination order. The selected node is then eliminated, creating and maintaining these buckets until all the nodes in the specified order have been removed. The elimination of the current node is contingent upon the type of node. If the current node is a leaf node, simply computing the marginal over its parent variables will suffice to eliminate it from the BN, as leaf variables only appear in their own CPT throughout the BN. However, when the node being eliminated is not a leaf node, a two-step process is followed to ensure its successful elimination.

In the *bn_functions.py* file, the process for eliminating a non-leaf node is carried out within the `eliminate_node` function. The first step of this process involves creating a potential over all the variables that are linked to the current node. This encompasses all the buckets associated with the current node, as well as the previous representation of product nodes for the current node. Since this cannot be completed in a single step, potentials for every bucket containing the current node are computed. Below is the pseudo code for the creation of the potential:

Initialize a new dictionary for calculation of buckets, **new_buckets**
for each **selected node** dictionary from current bucket:
- find index of current node in its key, say **index1**
- *for* each item of **current node** dictionary:
  - find index of current node in its key, say index2

  - *If* both the bucket have **same keys**:
    - *If* bucket not created in new_buckets, add it.
    - Create and add product node to **new_buckets**.

  - *If* value at **index1** matches value at **index2** :
    - *If* bucket not created in **new_buckets**, add it.
    - Remove value from current node key at **index2**.
    - Form **new_bucket_node_key** for the potential created.
    - *If* not a duplicate, create and add product node to **new_buckets**.

Return the updated **new_buckets**

Figure 8: Pseudo code for calculating potential

After calculating the potential for the bucket, the next crucial step involves eliminating the current node from this potential. In the context of a BN, node elimination refers to summing out the current variable, which essentially entails summing over the other variables and obtaining a distribution that excludes the current variable. The function `remove_node_from_potential` from the same file offers the necessary code to achieve this outcome, and its pseudo code is detailed below:

Set **delete_bucket_item** flag to True.
Deduce bucket name excluding the current node that is being removed, say variables_other_than_node
If no variables are left:
- set **delete_bucket_item** to False.

Compute the marginal distribution for variables_other_than_node using **marginalize** function
If **delete_bucket_item** is *True*, delete the old bucket from new_buckets.
If a bucket with the new name exists:
- Merge contents using **calculate_potential** function
- Return the result.

else:
- Update **new_buckets** with the marginal distribution.
- Return the updated **new_buckets**.

Figure 9: Pseudo code for removing variable from potential

The following steps are applied to each node in the elimination order until the sole remaining node in the bucket is the root node of the Bayesian network (BN). The data in the buckets are adjusted and updated to represent the equivalent arithmetic circuit (AC) of the input BN. Once this conversion process is complete, the resulting AC can be utilized for inference

after undergoing a verification process to ensure its accuracy. Subsequently, the next phase involves evaluating this generated AC and visually representing it.

IV. Evaluation of Arithmetic circuit and inferencing

To verify the correctness of the proposed arithmetic circuit, the following method was employed: every indicator value for all the variables was turned on, signifying the need to sum out every variable up to the root node. Each indicator node was assigned a value of 1, enabling every sum and product node to receive a value for computation. This approach aimed to aggregate all the Conditional Probability Tables (CPTs) from the Bayesian Network (BN) to form a Joint Probability Distribution (JPD) encompassing all the variables. By multiplying all the CPTs together and summing out all the variables, the resultant value would consistently be 1, signifying the creation of a JPD. Hence, turning every indicator node to the ON state equates to the process of multiplying and summing out every variable from this amalgamation of diverse probability distributions.

The `evaluate_arithmetic_circuit` function, which is defined in the *bn_functions.py* file, is a recursive function responsible for assigning values to the indicator nodes and performing computations using the arithmetic circuit (AC) structure. When the function encounters a node of the "sum" or "product" type, it calls itself with updated reference values for the specific node. If the current node is of type "value," the function simply returns the contents of the "value" key.

In the case of an "indicator" node, the function examines the evidence provided for inference and assigns appropriate values. If no evidence is provided, it assigns a TRUE value to every indicator node, indicating that they are turned on. However, if evidence is provided, the function compares the evidence contents with the values in the "node" and "variable_value" keys of the nodes. As it traverses through the AC representation, when it encounters a node matching the provided evidence, it turns the indicator node ON for that specific "node" and "variable_value" combination, while turning off the other indicators for the same node. Any indicator nodes that do not have evidence variables are kept ON. This allows the proposed AC to be verified and queried with specific evidence for accurate inference. The pseudo code for the same function has been provided below:

Determine the node type:
If **value** node, return its value.
If **sum** node, evaluate and sum all child nodes.
If **product** node, evaluate and multiply all child nodes.
If **indicator** nodes:
- *If* no **evidence**, return 1.
- *for* each evidence provided:
  - *If* current item matches **variable** and **value**, set to 1.
  - *If* current item matches **variable** but **not value**, set to 0.
  - *If* current item **not in evidence**, set to 1.

Return the value set.

Figure 10: Pseudo code for evaluating the proposed Arithmetic Circuit

V. Visualizing the Arithmetic Circuit

Graphviz is an open-source graph visualization software that provides tools for creating diagrams of abstract graphs and networks. It uses a simple text-based language called DOT to describe the structure of the graphs. The Graphviz library can generate various graphical outputs, including images in formats like PNG, PDF, and SVG.

As the AC forms a directed graph, the Digraph class from the Graphviz library appears to be the most suitable choice for visualization. The *plot_graph.py* file includes the `plot_graphviz` function, which contains the necessary code for visualizing the AC. Initially, the function initializes the Digraph for the AC data provided and creates a blank file with an SVG extension to plot the AC digraph.

The construction of the digraph involves traversing and plotting all the nodes in the graph first, followed by creating edges between the nodes based on the dictionary of other nodes in the "references" key of each node. The "references" key essentially holds all the children of that node, so a directed edge from the current node to all the nodes in the "references" list is deemed correct and appropriate.

Once all the nodes and edges are created, a DOT file is generated containing this information. The render function is then utilized to plot this information from the DOT file onto the initial blank image, thus providing you with the visual representation of the converted AC.

The images following this paragraph depicts the way the directed graph for the converted sum-product network appears to be. Distinct colors are utilized to distinguish between different types of nodes, making them easier to identify. The labels '+' and '*' are assigned to sum and product nodes, respectively, while indicator nodes have labels in the format of

"(Variable name)_indicator_(variable value)". Similarly, probability nodes have similar labels to indicator nodes, with the only difference being "(Variable name)_value_(variable value)". Sum nodes are denoted by a red background color, product nodes by a sky blue background color, indicator nodes by a lavender background color, and value nodes by a light purple background color. At the bottom of the graph, statistics for the number of sum, product, parameter, and total nodes are included. The given arithmetic circuit is for the Asia Bayesian Network and Survey Bayesian Network.

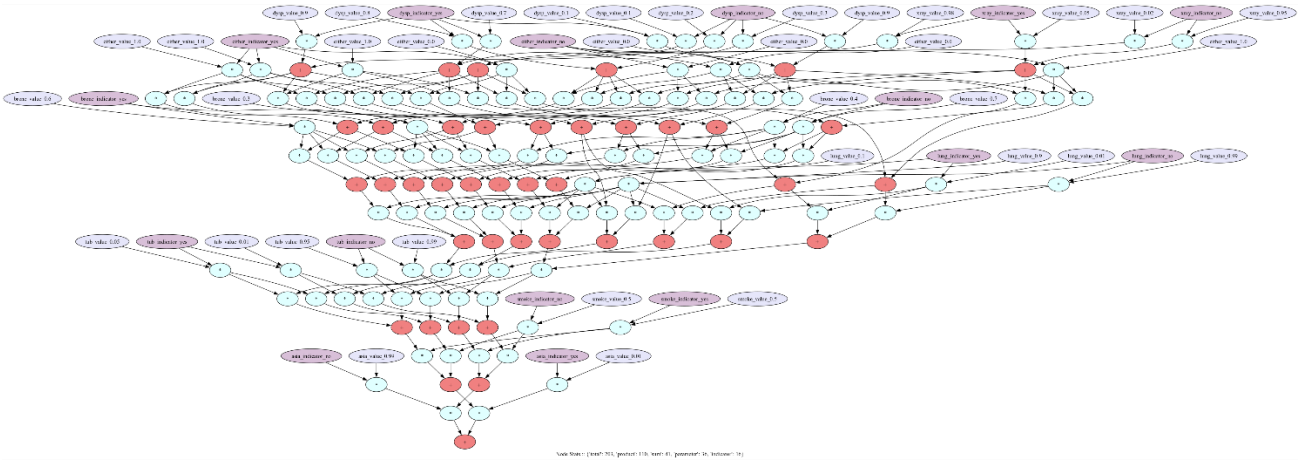

Figure 11: Arithmetic Circuit for Asia Bayesian Network

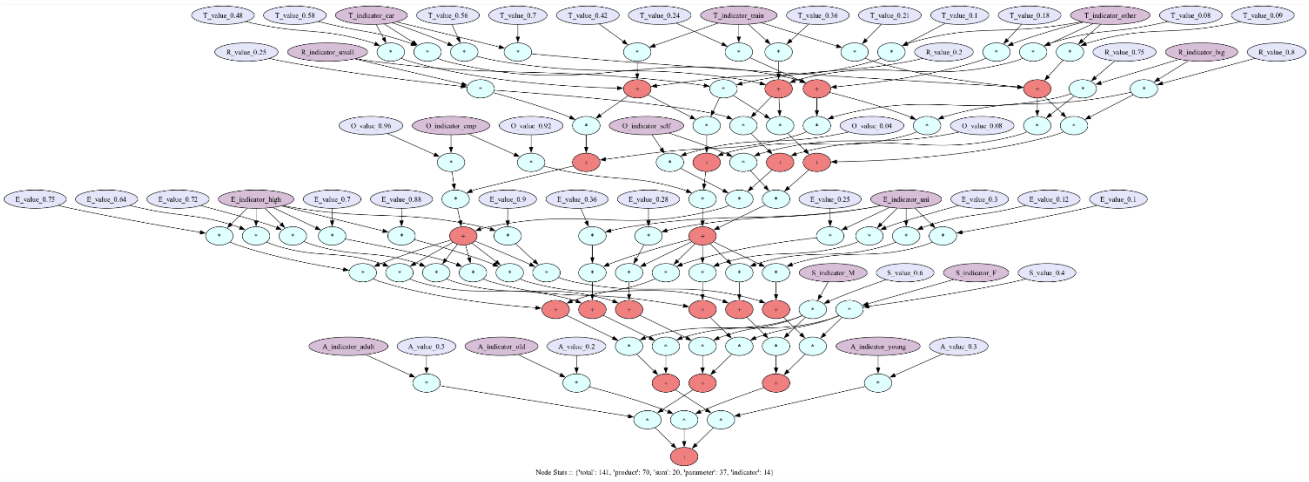

Figure 12: Arithmetic Circuit for Survey Bayesian Network

Conclusion and Future Work

In Conclusion, the project was made with significant strides in developing an innovative methodology for the conversion of Bayesian Networks (BN) into Arithmetic Circuits (AC). This involved a meticulous process encompassing the construction of product and sum nodes, the management of evidence through indicator nodes, and the efficient marginalization of variables using the variable elimination algorithm. Consequently, the resulting Arithmetic Circuits provide a robust framework for conducting probabilistic inference. These ACs offer the advantage of breaking down complex probabilistic models into more manageable components, thereby enhancing computational efficiency and scalability. In addition, we are considering refactoring and re-writing the project for improved efficiency, with a focus on minimizing the use of multiple loops as well as targeting larger BN networks. This overhaul aims to optimize the project's complexity, making it the most suitable for our approach. In addition to restructuring the code, there is significant potential for enhancing the project by carefully selecting the order of eliminations. This can lead to achieving optimal performance by automatically choosing the most effective elimination order.

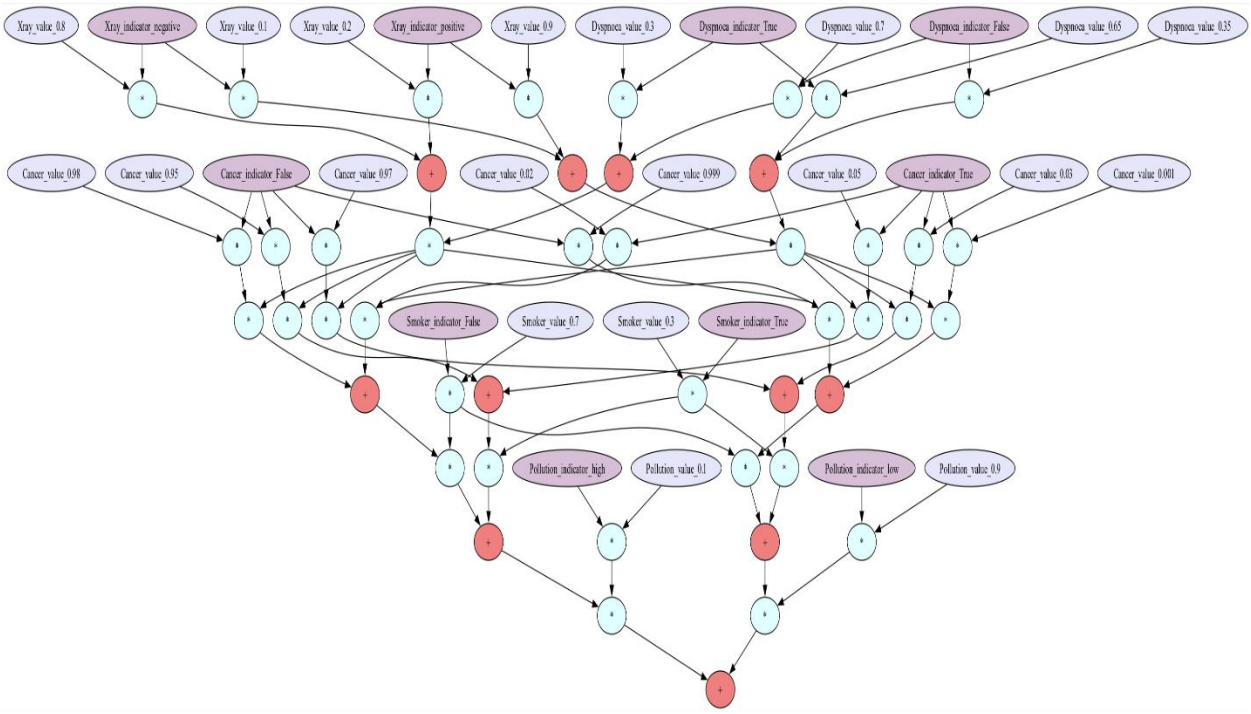

Figure 13: Plotted Arithmetic Circuit for Cancer Bayesian Network

References

- Y. Cao. (2010). "Study of the Bayesian networks," 2010 International Conference on E-Health Networking Digital Ecosystems and Technologies (EDT), Shenzhen, China. Retrieved from https://ieeexplore.ieee.org/document/5496612
- Ben-Gal, I. (2008). “Bayesian Networks”. In Encyclopedia of Statistics in Quality and Reliability (eds F. Ruggeri, R.S. Kenett and F.W. Faltin). Retrieved from https://onlinelibrary.wiley.com/doi/10.1002/9780470061572.eqr089
- Rooshenas, A. & Lowd, D. (2014). “Learning Sum-Product Networks with Direct and Indirect Variable Interactions”. 31st International Conference on Machine Learning. Retrieved from https://proceedings.mlr.press/v32/rooshenas14.html
- P. Wijayatunga. (2019). "Probabilistic Graphical Models and Their Inferences (Tutorial)". 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W), Umea, Sweden. Retrieved from https://ieeexplore.ieee.org/document/8791995