Anytime Algorithms for Learning Sum Product Networks and Sum Product Max Networks

by

Swaraj Pawar

(Under the Direction of Prashant Doshi)

Abstract

Although some research focuses on reducing the network sizes for Sum Product Networks and Sum Product Max Networks, the current structure learning algorithms have no control over the network sizes or the learning times. There exist some applications where the computation time is critical rather than the model performance. Anytime algorithms provide an approach to trade-off the computation time with the quality of the models. In this work, we introduce anytime algorithms for learning the SPNs and SPMNs. These algorithms return multiple models such that the initial approximate models need less learning time and are small in size. But by allowing more nodes and computation time, the performance of the networks improves over time. We evaluate the anytime algorithms over a testbed and demonstrate that the performance of the SPNs in terms of the log-likelihood and the SPMNs as given by the average rewards improves and reflect the performance profiles as expected for an anytime algorithm.

Index words: [Anytime Algorithms, Probabilistic Graphical Models, Decision Making, Machine Learning]

Anytime Algorithms for Learning Sum Product Networks and Sum Product Max Networks

by

Swaraj Pawar

B.E., Savitribai Phule Pune University, India, 2019

A Thesis Submitted to the Graduate Faculty of the
University of Georgia in Partial Fulfillment of the Requirements for the Degree

Master of Science

Athens, Georgia

2021

Anytime Algorithms for Learning Sum Product Networks and Sum Product Max Networks

by

Swaraj Pawar

Major Professor: Prashant Doshi

Committee: Sheng Li
Ninghao Liu

Electronic Version Approved:

Ron Walcott
Dean of the Graduate School
The University of Georgia
December 2021

Dedication

Dedicated to my mother, father, and my brother

Acknowledgments

First and foremost, I would like to acknowledge and give my warmest regards to my major professor, Dr. Prashant Doshi. This research was possible due to his constant guidance and support throughout my Master's research. His expertise and insights during the weekly meeting discussions helped me overcome the obstacles and gain a deeper understanding of my research area. I also express my sincere gratitude to Dr. Sheng Li and Dr. Ninghao Liu for being a part of my advisory committee.

I thank Layton Hayes and Hari Teja Tatavarti for helping me in setting up the work environment and understand the background concepts. I also thank Daniel Redder for proof reading my thesis.

I am extremely grateful to my parents for always standing by my side and supporting me to fulfill my dreams and ambitions. Their constant love, care, struggle and sacrifices for providing me the best education cannot be expressed in words.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many tasks that are performed by the intelligent or automated systems require reasoning. They need to take into consideration the available information given to them and draw some conclusions from it. Such conclusions might consist of finding the probability of some fact being true given the evidence or what actions to take in the given scenario. For example, consider the use of such a reasoning system in the field of medical. Given the symptoms, test results and other characteristics of the patient, such systems would aid the medical professional to find the possibility of some underlying disease or would help in proposing a treatment plan that would benefit the patient. Such tasks involving complex distributions and dependencies between multiple factors need a mechanism for efficient representation and inference.

Probabilistic Graphical Models have the ability to compactly represent such complex distributions over a high dimensional space using graph-based representations. The nodes of the graph structures represent the variables, and the edges denote the probabilistic dependence between them. This makes the conditional dependence and independence clear and offer a factored distribution of the joint distribution that they represent. Such distribution is effective for inference, but the inference is often intractable. Also, the graph structures could be designed by hand or learnt directly using a data-driven approach.

Bayesian Networks have been the predominant and the most popular probabilistic graphical models. Bayesian Networks are directed acyclic graphs that represent the probability distribution by exploiting conditional independence properties of the distribution to allow a compact and natural representation (Koller

& Friedman, 2009). But their inference is generally intractable or exponential in the number of nodes. Other graphical model like Arithmetic Circuits (Darwiche, 2003) and Sum Product Networks (Poon & Domingos, 2011) have inference that is linear in the number of edges and nodes of the networks respectively maintaining tractability.

Sum Product Networks are Probabilistic Graphical Models that can be learned directly from the data. Multiple algorithms have been presented in the past for structure learning for the SPNs. Most of these works target an improvement in the log-likelihood that is given by the networks learned by the algorithms. These networks are appealing due to the linear inference complexity in the terms of the size of the network. They are based upon the network polynomials that gives a representation of the probability distribution of the Bayesian Networks in form of a polynomial. They represent these network polynomials compactly which otherwise would be exponential in the number of variables. But a major limitation of these SPNs is that the size of the networks is not bounded.

Similarly, Influence Diagrams (IDs) have been the state-of-the-art graphical models used for making decision in order to maximize the expected utility. These diagrams are a generalization of the Bayesian Networks using utility and decision variables. Not much research exists for learning the IDs directly from data. Also, the inference is np-hard.

Later, Sum Product Max Networks (Melibari et al., 2016) have been introduced for probabilistic decision making. These networks generalize the SPNs by adding max and utility nodes to the structure. The max nodes correspond to the decision variables and the utility nodes to the reward functions. The inference for these networks is tractable and linear in the number of nodes. However, in the case of complex or sequential domains, the structure learning algorithm for SPMNs fails to learn the networks in reasonable time. Recurrent SPMNs (Tatavarti et al., 2021) and State-based Recurrent SPMNs (Hayes et al., 2021) have been designed to model sequential data. But still the sizes of these networks remain unbounded.

In some of the applications requiring reasoning for probabilistic inference or decision making, the amount of time required for computation may be more critical than the quality of the results. Such

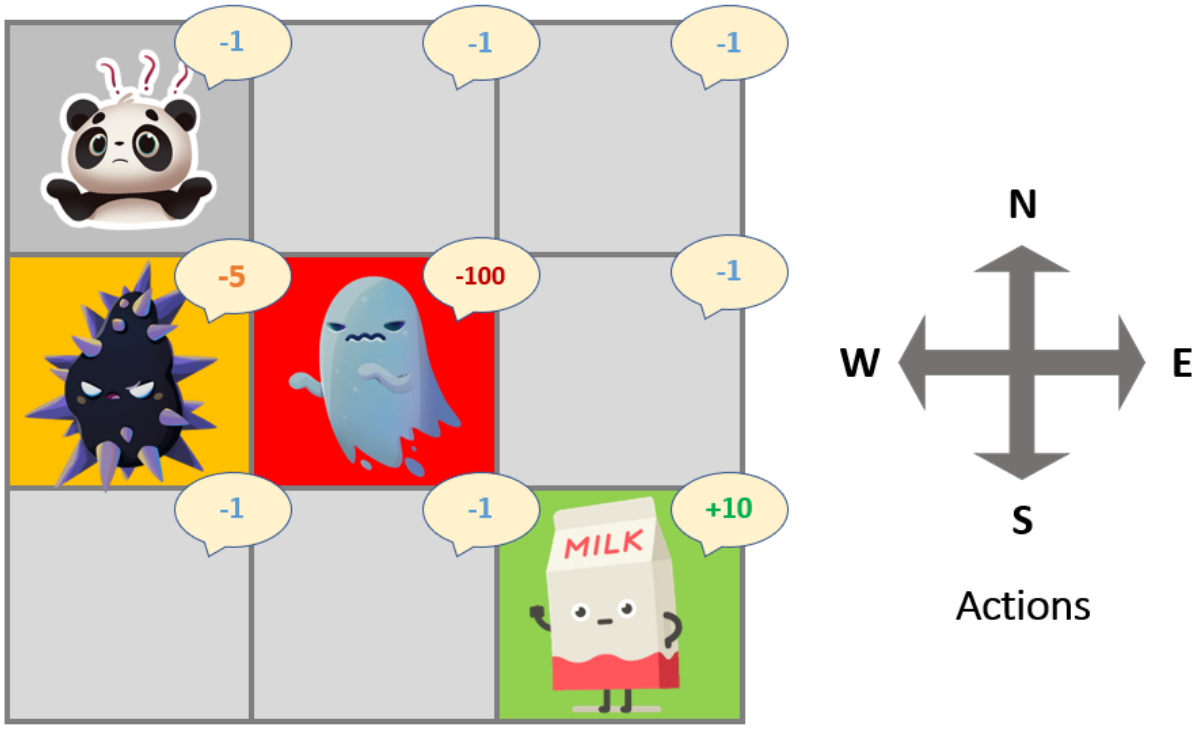


Figure 1.1: An example of a Navigation Grid Domain

intelligent systems can perform satisfactorily using approximate results computed in lesser amount of time. The longer times required for finding the optimal results degrade the overall utility of these systems. Such systems should have the capability to the trade the computation time with the quality of the results.

The structure learning algorithms for SPNs and SPMNs have no control over the network sizes or the complexity of the learnt networks. Some researchers have focused on simplifying the networks by limiting the data splits to two clusters and stopping the recursive learning algorithm early by having multivariate leaves (Vergari et al., 2015). Attempts have been made by (Di Mauro et al., 2017) to reduce the network size and the learning times by an alternate variable splitting method by sacrificing the performance of the networks. Although these efforts exist, there is a need of a mechanism that allows us to adjust the trade-off between the performance of the network and the network complexity or the structure learning time.

Anytime or flexible algorithms are the algorithms that showcase an increase in the quality of the results as the computation time increases (Zilberstein, 1996; Zilberstein & Russell, 1995). These algorithms can

(a) Initial Policy (b) Intermediate Policy (c) Final Policy

Figure 1.2: Improvement in the policy by an Anytime Technique.

return multiple approximate models in the initial phases and generate improved models as the algorithm progresses. In this thesis, we develop anytime approaches for learning the Sum Product Networks and the Sum Product Max Networks.

To better understand the concept of anytime algorithms in decision making, consider the navigation grid example as shown in the Figure 1.1. This toy domain consists of a 3×3 grid where task is to find the directions for the hungry baby panda to reach the milk location. The possible action at each step includes going to the *North*, *South*, *East* or *West* directions. The goal for the baby panda is to reach the green cell having milk to get a reward of 10. If the baby panda goes to a normal cell location, then the action drains its health by 1, that is a penalty of 1. If the panda steps into the spiky location, then it gets a penalty of 5. But if it goes to the red ghost cell, then it won't be able to escape that cell and receives a penalty of 100. Thus, a proper path is to be found by achieving maximum points.

An anytime algorithm to find a path in this domain would start with a bad policy and proceed to improve the average rewards gained. Consider that the algorithm first generates a model that gives a random policy allowing actions in any of the directions with equal probability as shown in figure 1.2a. Such a model would receive a very low reward since the probability of the panda going to the ghost location is very high. As the algorithm continues, consider that it generates an intermediate model that figures the relative location of the goal state to $south - east$ and takes only $south$ or $east$ action at each step as

seen in figure 1.2b. Such a policy would receive a higher reward since the probability of going to the ghost cell is reduced only to two possible policies: '*going south and then east*' or '*first going east and then south*'. At the end, the anytime algorithm would be able to figure out a proper policy giving optimal rewards by consuming enough resources. Using the policy shown in the figure 1.2c, the baby panda would always be able to reach the milk location by avoiding the danger cells.

As seen from the example, an anytime algorithm would improve the performance or the quality of the results as the algorithm progresses. Such an algorithm would be able to produce approximate results by utilizing less computation time as compared to the time required for finding the optimal results. Thus, the system wouldn't have to wait for long learning times and would be able to gain some better results while the anytime algorithm continues to focus on improving the results.

The work in this thesis focuses on addressing such types of reasoning problems by introducing anytime algorithms for learning SPNs and SPMNs that would initially generate approximate and less complex networks, and then add on to the network complexity for improving the performance.

1.1 Contribution

In this thesis, we present separate anytime or flexible algorithms for structure learning of Sum Product Networks and Sum Product Max Networks. The specific contributions are as follows:

- We present a new anytime algorithm, AnytimeSPN, for learning the structures of Sum Product Network with increasing performance. The AnytimeSPN algorithm makes use of a modified LearnSPN algorithm, LearnSPN*, that generates the networks flexibly at each iteration. The generated SPNs have increasing log-likelihood as the computation time and the network complexity increases.

- We also introduce an anytime algorithm called AnytimeSPMN that offers flexible structure learning for the Sum Product Max Networks such that the average rewards obtained on simulating the policies given by the networks and the log-likelihood of the networks increases over time. This

algorithm generates the structures using the presented LearnSPMN* algorithm, which is an extended version of the LearnSPMN algorithm to have a control over the network structure.

- Some modifications to the structure of the decision nodes of the SPMNs are presented in this thesis in order to incorporate flexibility.

- We redefine validity for SPMNs and further prove that the network structures returned by the anytime algorithms at each iteration satisfy the conditions that are required for a valid network.

- On a well-defined testbed of binary datasets for SPNs and decision-making datasets for SPMN, we evaluate our anytime algorithm to obtain the performance profiles for each dataset. We show an increase in the log-likelihood for the AnytimeSPN algorithm and an improvement in the average rewards and the log-likelihood for the AnytimeSPMN. We also showcase the increasing network complexities and computation time for the algorithms.

- To have a better understanding of the performance profiles of the anytime algorithms, we compare the performances with the baselines given by the LearnSPN and the LearnSPMN algorithms, and also with an upper bound or an optimal value for the results to show that the performance of these anytime algorithms converge towards an optimal value.

1.2 Organization of Thesis

The remaining thesis is organized as follows:

- In the Chapter 2, *"Background"*, we discuss important concepts and preliminaries that help in building a better understanding of the thesis. In Section 2.1, we discuss network polynomial and their evaluation. Then in the Sections 2.2 and 2.3, we elaborate on Sum Product Networks and Sum Product Max Networks along with their properties, the notion of validity and their structure learning algorithms. Here we present a new definition for SPMN validity. The Section 2.4 then explains

the anytime algorithms and their performance profiles. The Section 2.5 explains the working of the X-means clustering algorithm.

- The "Related Work" chapter, Chapter 3, discuss some important research work done in the field of SPNs and SPMNs that relates to the thesis work.

- The Chapter 4 *"Anytime SPN"* focuses on presenting an anytime algorithm for the Sum Product Networks. The Section 4.1 presents an algorithm which induces changes in the controlling parameters for the SPNs and also discusses the convergence criteria for the algorithm. A modified LearnSPN algorithm for generating flexible SPNs given the parameters is introduced in the Section 4.2. We then show the effects of the anytime algorithm on the SPN structure in the Section 4.3. We also discuss the validity for each SPN structure returned by the algorithm in the Section 4.4.

- The *"Anytime SPMN"* chapter, Chapter 5, introduces an anytime approach for generating Sum Product Max Networks for decision making. The Section 5.1 first explains the changes made to the structure of networks. In the Section 5.2, we present the anytime algorithm for SPMN that returns better structures that are learned using the modified LearnSPN algorithm given in the Section 5.3. We illustrate the improvement in the SPMN structure by the anytime algorithm in the Section 5.4. Further in the Section 5.5, we discuss that each generated SPMN is valid. And finally, in the Section 5.6, we discuss the policy evaluation for the modified SPMN structure.

- In the Chapter 6, *"Experiments"*, we discuss the experimental results for the anytime algorithms. In the Section 6.1, we discuss the datasets used for evaluating the AnytimeSPN and the AnytimeSPMN algorithm. The Section 6.2 presents the performance profiles given by the AnytimeSPN algorithm for the datasets. Similarly, the Section 6.3 showcases the performance profiles and changes in the network over the duration of the AnytimeSPMN algorithm.

- Finally, we conclude the thesis in the Chapter 7 in which we summarize our work and discuss some future directions to this work.

Chapter 2

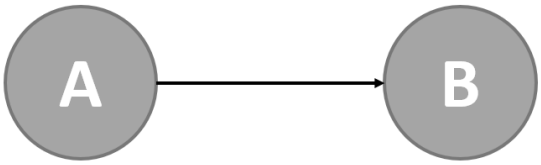
Background

In this chapter, we discuss the background foundation concepts and preliminaries that are necessary for a proper understanding of the work that is described in the later chapters of this thesis. In the Section 2.1, we first explain the network polynomial and some theorems related to it. Then in the Section 2.2, we define the Sum Product Networks and some properties associated with a valid SPN. Further we also describe the structure learning algorithm LearnSPN. We similarly define the Sum Product Max Networks along with their properties and the structure learning algorithm in the Section 2.3. To understand the notion of the anytime algorithms and their performance profiles, we explain them in the Section 2.4. We also describe the working of the X-means algorithm in the Section 2.5.

2.1 Network Polynomials

A network polynomial (Darwiche, 2003) gives us the probability distribution of a Bayesian network in form of a polynomial. Such a probability distribution is a multilinear function that takes the form of a multivariate polynomial where each variable has a degree 1.

In order to understand the network polynomial, let us consider a Bayesian Network N, having two random variables (A and B) as shown in the Figure 2.1. In this network, variable A is the parent of B. Such a Bayesian network N is a directed acyclic graph over the variables $\mathbf{X}$, having conditional probability



A	P(A)
True	$\theta_a = 0.7$
False	$\theta_{\bar{a}} = 0.3$

A	B	P(B|A)
True	True	$\theta_{b\|a} = 0.2$
True	False	$\theta_{\bar{b}\|a} = 0.8$
False	True	$\theta_{b\|\bar{a}} = 0.6$
False	False	$\theta_{\bar{b}\|\bar{a}} = 0.4$

Figure 2.1: A Bayesian Network with two variables

tables for each of the variables. The conditional probability value $\theta_{x|u}$ for the variable x having parent u is considered to be the network parameters and represent the probability $P(x|U)$.

The probability distribution for the Bayesian Network shown in the Figure 2.1 can be given as:

$$\begin{aligned} P(A,B) &= \sum_a \sum_b P(a,b) \\ &= P(a,b) + P(a,\bar{b}) + P(\bar{a},b) + P(\bar{a},\bar{b}) \\ &= P(a)P(b|a) + P(a)P(\bar{b}|a) + P(\bar{a})P(b|\bar{a}) + P(\bar{a})P(\bar{b}|\bar{a}) \end{aligned}$$

Since $\theta_{x|U}$ represents the probability $P(x|U)$, we can re-write the equation as:

$$P(A,B) = \theta_a\theta_{b|a} + \theta_a\theta_{\bar{b}|a} + \theta_{\bar{a}}\theta_{b|\bar{a}} + \theta_{\bar{a}}\theta_{\bar{b}|\bar{a}} \tag{2.1}$$

To compute the probability distributions given a set of evidence variables, evidence indicators λ_x for each of variable X are introduced so as to generalize the multivariate polynomial.

Definition 2.1.1 (Evidence Indicator) *For a variable X in the Bayesian network N given the evidence* **e**, *the evidence indicator variable λ_x is defined as:*

$$\lambda_x = \begin{cases} 1 & \text{, if } x \text{ is consistent with the evidence } \mathbf{e} \\ 0 & \text{, otherwise} \end{cases} \tag{2.2}$$

Thus, we can use the evidence indicators for generalizing the Equation 2.1 to get the network polynomial as:

$$f = \lambda_a \lambda_b \theta_a \theta_{b|a} + \lambda_a \lambda_{\bar{b}} \theta_a \theta_{\bar{b}|a} + \lambda_{\bar{a}} \lambda_b \theta_{\bar{a}} \theta_{b|\bar{a}} + \lambda_{\bar{a}} \lambda_{\bar{b}} \theta_{\bar{a}} \theta_{\bar{b}|\bar{a}} \tag{2.3}$$

Definition 2.1.2 (Network Polynomial) *If N is a Bayesian network over the variables* **X** *and U denotes the parents of the variable X in the network, then the network polynomial for the network N is given as:*

$$f = \sum_{\mathbf{x}} \prod_{x\mathbf{u} \sim \mathbf{x}} \lambda_x \theta_{x|\mathbf{u}} \tag{2.4}$$

In the Equation 2.4, the outer summation ranges over all possible instantiations **x** of the variables. For each of such instantiation **x**, the inner product ranges over all instantiations of families x**u** that are compatible with **x**. In order to compute the probability for an evidence **e**, that represents any instantiation of some variables **E** in the network, we can simply evaluate the network polynomial f for **e**.

Definition 2.1.3 *(Darwiche, 2003) The value of network polynomial f at evidence* **e**, *denoted by $f(\mathbf{e})$, is the result of replacing each evidence indicator λ_x in f with 1 if x is consistent with* **e**, *and with 0 otherwise.*

For example, consider the Equation 2.3 for the Figure 2.1. Let the evidence **e** be $\bar{a}b$. Then we can compute $f(\mathbf{e})$ by substituting $\lambda_a = 0$, $\lambda_{\bar{a}} = 1$ and $\lambda_b = 1$, $\lambda_{\bar{b}} = 0$. On evaluating the equation, we get $f(\mathbf{e}) = \theta_{\bar{a}} \theta_b$, which represents the probability $Pr(\bar{a}b)$.

Theorem 2.1.1 *(Darwiche, 2003) Let N be a Bayesian network representing probability distribution Pr and having polynomial f. For any evidence* **e**, *we have $f(\mathbf{e}) = Pr(\mathbf{e})$*

The network polynomials and their partial derivative can be used to answer multiple probabilistic queries as given by (Darwiche, 2003). The polynomial is exponential in size and cannot be represented as a set of terms. But they can be represented efficiently using Arithmetic Circuits (ACs) that may sometimes even be linear in size. The Arithmetic Circuits can be defined a rooted directed acyclic graphs having arithmetic operators $(+, -, \times, /)$ as the internal nodes and numerical values of the variables as the leaf nodes. The size of this representation is given by the number of edges that it consists of. Evaluating the arithmetic circuits for probabilistic inference is tractable as it is linear in their size.

2.2 Sum Product Networks

The Sum Product Networks (SPNs) can be considered to be a restricted form of the Arithmetic Circuits where the internal nodes are only sum and product nodes. The notion of the network polynomials forms the basis of the SPNs. As stated in (Poon & Domingos, 2011), the network polynomial can be generalized using an unnormalized probability distribution $\Phi(x) \geq 0$. Considering $\Pi(x)$ to be product of indicators having a value of 1 in state x, we get the network polynomial as $f' = \sum_x \Phi(x)\Pi(x)$. The partition function Z is the normalizing factor that is obtained from the network polynomial when all the indicator values are set to 1. For any given evidence **e**, we can then compute its probability as $P(\mathbf{e}) = \frac{\Phi(\mathbf{e})}{Z}$.

The size of the network polynomial is exponential in the number of the variables. But we could represent and evaluate the network polynomial in linear space and time with the help of Sum Product Networks. The definition of the SPNs is given as:

Definition 2.2.1 (Sum Product Networks) *(Poon & Domingos, 2011) A sum product network (SPN) over variables $x_1, x_2, ..., x_d$ is a rooted directed acyclic graph whose leaves are the indicators $x_1, ..., x_d$ and $\bar{x_1}, ..., \bar{x_d}$ and whose internal nodes are sums and products. Each edge (i, j) emanating from a sum node i has a non-negative weight w_{ij}. The value of a product node is the product of the values of its children. The*

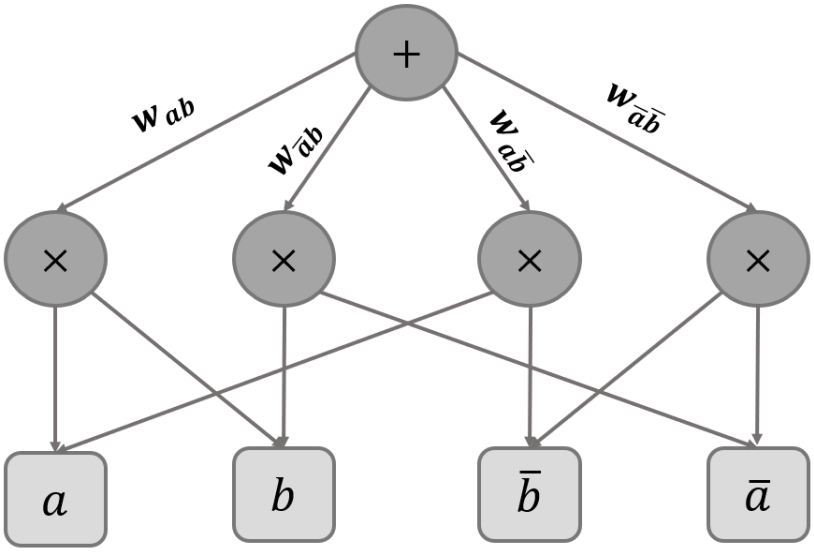


Figure 2.2: A Sum Product Network with two variables

value of a sum node is $\sum_{j \in Ch(i)} w_{ij} v_j$*, where* $Ch(i)$ *are the children of* i *and* v_j *is the value of the node j. The value of an SPN is the value of its root.*

The SPNs can be considered to be a function of their indicator variables as $S(x_1, ..., x_n, \bar{x_1}, ..., \bar{x_n})$. Consider the SPN having two variables that is shown in Figure 2.2. This SPN can be considered to be one of the possible representations for the network polynomial 2.3 that is given for the Bayesian network illustrated in Figure 2.1. In the case of Figure 2.2, the given SPN can be evaluated by $S(a, b, \bar{a}, \bar{b})$. If the complete state is $\bar{a}b$, then $S(x) = S(0, 1, 1, 0)$. If the partial evidence e is $a = 0$, then $S(e) = S(0, 1, 1, 1)$. Here the SPN can be evaluated as:

$$S(a, b, \bar{a}, \bar{b}) = w_{ab}ab + w_{a\bar{b}}a\bar{b} + w_{\bar{a}b}\bar{a}b + w_{\bar{a}\bar{b}}\bar{a}\bar{b} \tag{2.5}$$

The term $S(x)$ for an SPN represents that the indicators specify the complete state x. If a partial evidence e is given, it is abbreviated as $S(e)$. Similarly, when all the indicator variables are set to 1, it is represented as $S(*)$. The unnormalized probability distribution over $\mathbf{X}$ is defined by the values of

$S(x)$ for all $x \in \mathbf{X}$. So, under this distribution, the unnormalized probability of evidence e is $\Phi_S(e) = \sum_{x \sim e} S(X)$ and the partition function is $Z_S = \sum_{x \sim \mathbf{X}} S(X)$.

Definition 2.2.2 (Validity) *(Poon & Domingos, 2011) A sum product network S is valid iff $S(e) = \Phi_S(e)$ for all evidence e.*

Thus, we can say that an SPN is valid if it is always able to correctly compute the probability of any given evidence. So, if the SPN S is valid, then $S(*) = Z_S$. Such condition for a valid SPN is possible only if the expansion of the SPN is its network polynomial. By evaluating the SPN S in a bottom-up fashion, it can be expressed as the polynomial $\sum_k s_k \Pi_k(...)$, where the term $\pi_k(...)$ is a monomial over the indicator variables and $s_k \geq 0$ is the coefficient. That is, the monomials in the expansion are in one-to-one correspondence with the state x. This means that each of the monomials is non-zero in exactly one state and each state has exactly one monomial that is non-zero in the expansion. The detailed proof for the validity condition is given in (Poon & Domingos, 2011).

For a better understanding of valid SPNs, consider the SPN $S(a, b, \bar{a}, \bar{b})$ in Figure 2.2 having the expansion 2.5. Consider the partial evidence e to be $(a = 0)$. Thus $S(e) = S(0, 1, 1, 1)$ and from Equation 2.5, we get $S(e) = w_{\bar{a}b} + w_{\bar{a}\bar{b}}$. Now the evidence is consistent with the states $\bar{a}b$ and $\bar{a}\bar{b}$. Since $\Phi_S(e) = \sum_{x \sim e} S(x)$, we compute $\Phi_S(e) = S(0, 1, 1, 0) + S(0, 0, 1, 1)$. Since $S(0, 1, 1, 0) = w_{\bar{a}b}$ and $S(0, 0, 1, 1) = w_{\bar{a}\bar{b}}$, we have $\Phi_S(e) = w_{\bar{a}b} + w_{\bar{a}\bar{b}}$. Thus, we can conclude that since the given SPN structure is valid, we can observe that $S(e) = \Phi_S(e)$.

From the Definition 2.2.2 and the further discussion, it is clear that a sum product network is valid if its expansion gives its network polynomial. But to ensure this condition, some properties are provided for the SPNs that help to guarantee validity.

Definition 2.2.3 (Scope) *(Poon & Domingos, 2011) The scope of a node is union of the scopes of its children, where the scope of a leaf node is the set of random variables whose distribution it holds.*

Definition 2.2.4 (Completeness) *(Poon & Domingos, 2011) An SPN is complete iff each child of a sum node has the same scope.*

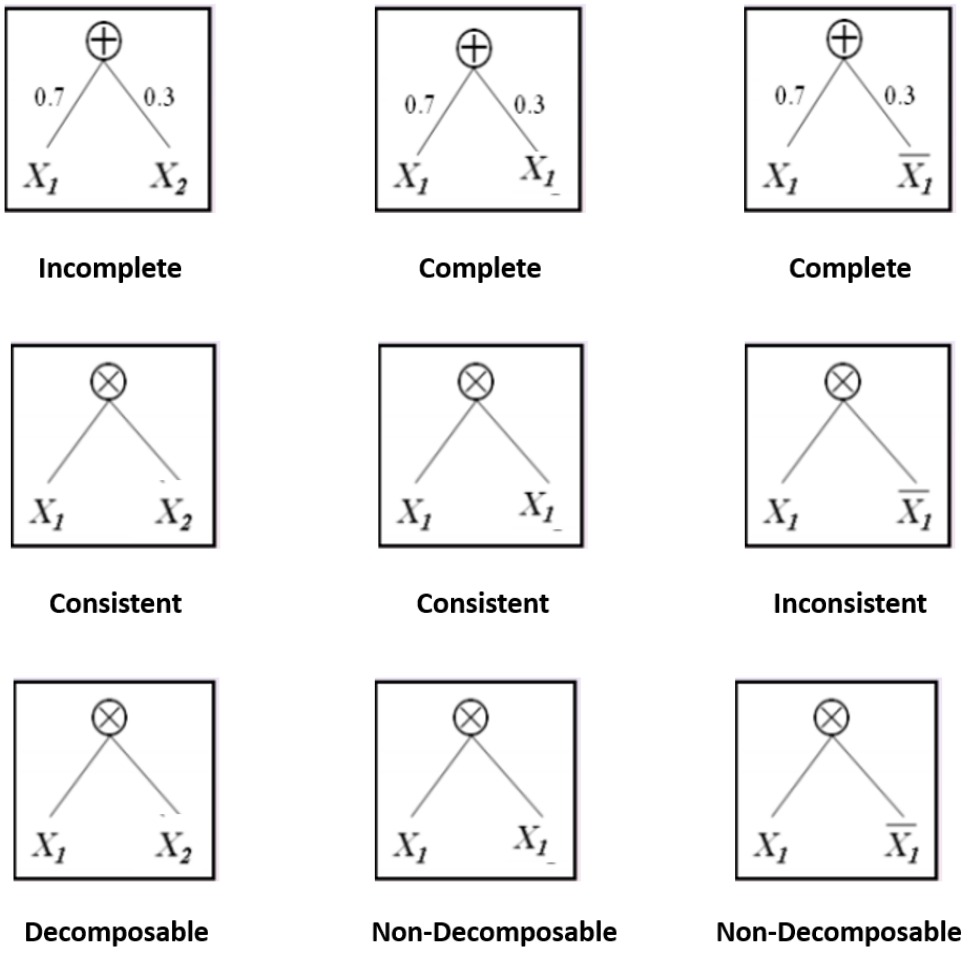


Figure 2.3: Examples illustrating the properties of the SPNs

Definition 2.2.5 (Consistency) *(Poon & Domingos, 2011) An SPN is consistent iff no variable appears negated in one child of the product node and appears non-negated in another.*

Definition 2.2.6 (Decomposability) *(Poon & Domingos, 2011) An SPN is decomposable iff no variable appears in the scopes of more than one child of a product node.*

The Figure 2.3 illustrates various examples of SPNs that portray the satisfaction of the properties that are defined above. Using these properties, we can ensure the validity of any SPN structure.

Theorem 2.2.1 (SPN validity) *(Poon & Domingos, 2011) An SPN is valid if it is sum-complete and consistent / decomposable.*

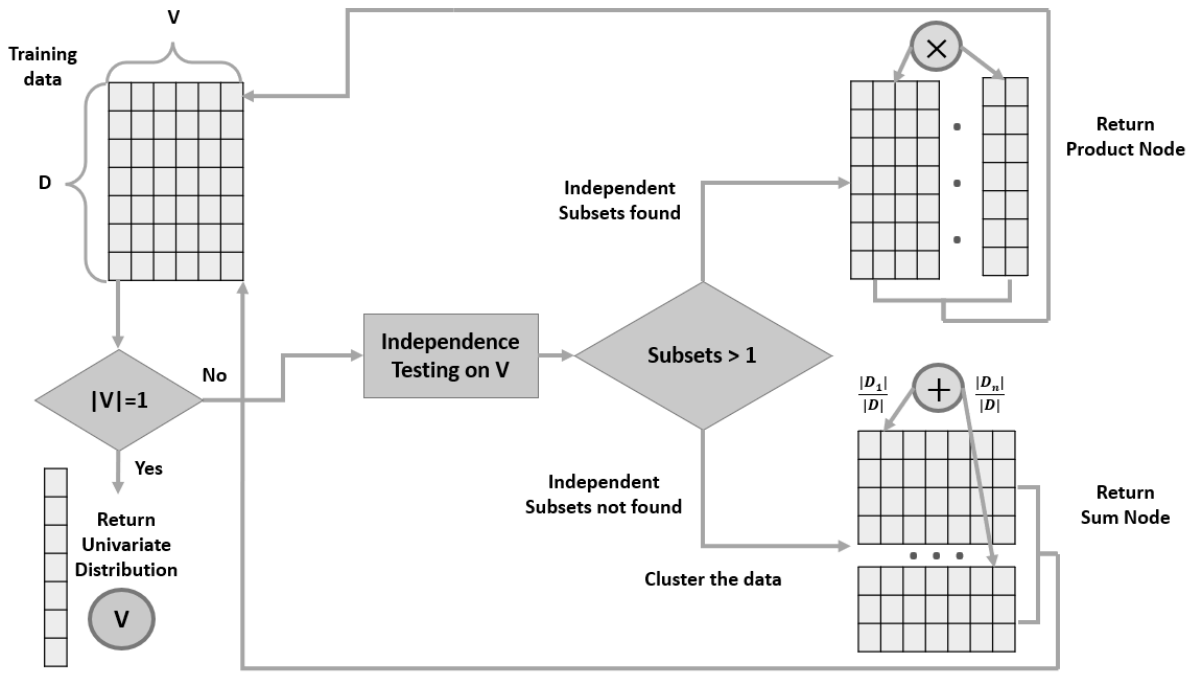


Figure 2.4: An illustration of the LearnSPN algorithm

It is important to note that the properties completeness and consistency / decomposability are sufficient to guarantee validity, but they are not necessary for validity. For example, an SPN that represents the polynomial $S(x_1, x_2, \bar{x_1}, \bar{x_2}) = \frac{1}{2}x_1x_2\bar{x_2} + \frac{1}{2}x_1$ satisfies the condition 2.2.2 for validity, but is incomplete and inconsistent. But these properties are essential for a stronger property that every sub-SPN is valid.

2.2.1 Structure Learning

We have discussed the properties that are sufficient for a valid SPN or necessary for every sub-SPN structure to be valid. Also, as the SPN is able to efficiently represent the network polynomial, it offers tractable inference. But it is important to understand how the structure of these SPNs are designed so that the validity and tractability of the networks are maintained. One of the earlier methods was to design these networks by hand and then try to adjust the network weights over the training dataset. This approach would require deep understanding of the domain and might prove to be a complicated task for complex domains. To resolve this issue, a structure learning algorithm was presented by (Gens & Domingos,

Algorithm 1: LearnSPN

Input: D: Dataset, V: Variables
Output: learned SPN structure

```
if |V| = 1 then
    return smoothed univariate distribution over V
else
    Partition variables V into independent subsets V_j
    if success then
        return ∏_j LearnSPN(D_j, V_j)
    else
        partition D into clusters D_j of similar instances
        return Σ_j (|D_j|/|D|) × LearnSPN(D_j, V)
```

2013) that is able to learn the network structures directly from the data. This generic structure learning algorithm, known as LearnSPN, is presented in Algorithm 1.

The LearnSPN algorithm is a recursive algorithm as illustrated in Figure 2.4 that returns a valid SPN at each call. The recursive definition of SPNs is given as:

Definition 2.2.7 (SPNs) *(Gens & Domingos, 2013) A sum product network is defined as:*

1. *A tractable univariate distribution is an SPN*
2. *A product of SPNs with disjoint scope is an SPN*
3. *A weighted sum of SPNs with the same scope is an SPN, given all weights are positive*
4. *Nothing else is an SPN*

The LearnSPN algorithm takes as input a dataest D having a set of variables V, the algorithm first checks if the set V contains only a single variable. If this is the case, then the algorithm returns a smoothed univariate distribution over that variable. This algorithm repeatedly performs splits on the dataset using independence testing and clustering until it reaches the base case of a single variable in the data. First the algorithm applies independence testing on the variables V and tries to split them into

approximately independent subsets. If it is successfully able the partition the variable set, then a newly created product node is returned whose children are the sub-SPNs that are learned using each of these correlated subsets of variables that are independent of variables in other groups. Hence each of these groups have a disjoint scope that ensures decomposability. If the independence testing fails to find any such partitions, then the algorithm partitions the dataset into clusters having similar instances. In this case, a sum node is returned whose edges correspond to the sub-SPNs learned from the respective clusters. Each out-going edge of a sum node corresponding to a cluster is a weighted edge whose weight is given by the proportion of instances within that cluster ($\frac{NumberOfInstanceInTheCluster}{TotalInstancesInTheDataset}$). Since all the clusters have the same scope, the resultant SPN is complete. Thus, recursive application of independence testing and clustering operations guarantee a valid SPN. An additional base case can be used for the algorithm where the dataset is naïve factorized on reaching a required minimum number of instances. Till date, multiple structure learning algorithms have been built upon this generic LearnSPN algorithm.

2.3 Sum Product Max Networks

The Sum Product Networks, that are probabilistic graphical models, have been extended to Sum Product Max Network by (Melibari et al., 2016) to incorporate decision making capability in them. As evident from the name, the SPMNs include max nodes for decision variables in addition to the sum and product nodes, and also include utility nodes to represent the utility functions.

Definition 2.3.1 (Sum Product Max Networks) *(Melibari et al., 2016) An SPMN over decision variables $D_1, ..., D_m$, random variables $X_1, ..., X_n$, and utility functions $U_1, ..., U_k$ is a rooted directed acyclic graph. Its leaves are either binary indicators of the random variables or utility nodes that hold constant values. An internal node of SPMN is either a sum, product, or max node. Each max node corresponds to one of the decision variables and eachoutgoing edge from a max node is labeled with one of the possible values of the corresponding decision variable. Values of a max node i is $\max_{j \in Children(i)} v_j$, where $Children(i)$ is*

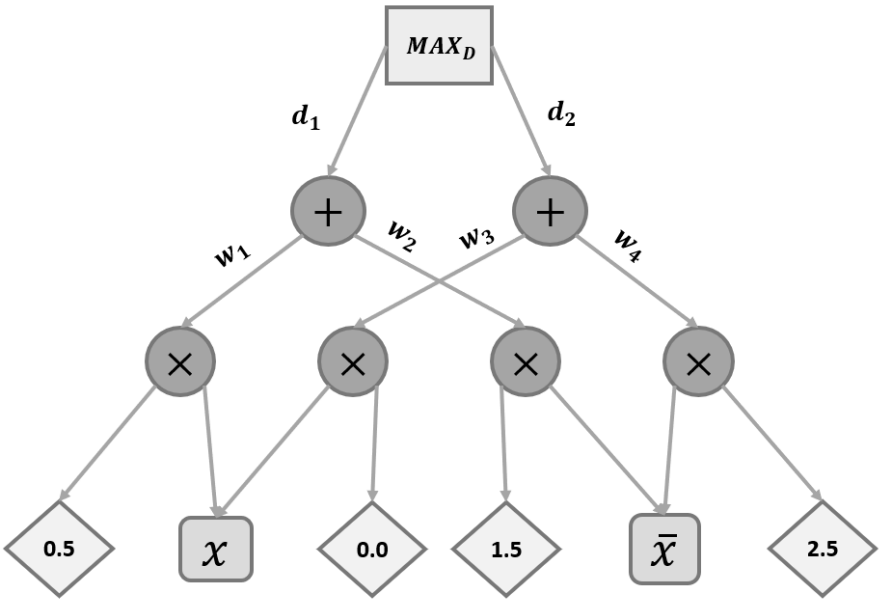


Figure 2.5: A Sum Product Max Network

the set of children of i, and v_j is the value of the subgraph rooted at child j. The sum and product nodes are defined as in the SPN.

Before discussing the validity for SPMN, we must understand the concepts of information sets and partial orders (Koller & Friedman, 2009). The information sets $I_0, I_1, ...I_m$ are subsets of random variables such that the variables in the information set I_{i-1} are observed before the decision D_i is taken. Such sets could be possibly empty and the variables in the set I_m need not be observed before any decision variable. The order in which the variables are observed and the decisions are taken is the partial order $P^{\prec}$ and is given as $I_0 \prec D_1 \prec I_1 \prec ... \prec D_m \prec I_m$. This order is partial in the sense that the variables in an information set can be observed in any order. Also, given the partial order, it is important to understand how the decision making problems are expressed mathematically.

2.3.1 Sum-Max-Sum Rule

The mathematical expression that is used for the evaluation of a decision making problem by utilizing the partial order is termed as the Sum-Max-Sum rule (Koller & Friedman, 2009). Consider the partial order

$P^{\prec} = I_0 \prec D_1 \prec I_1 \prec ... \prec D_m \prec I_m$. Let $\mathbf{X_i}$ denote the set of random variables that are included in the information set I_i. Thus, the maximum expected utility (MEU) is given by the Sum-Max-Sum rule as:

$$MEU(P^{\prec}) = \sum_{\mathbf{X_0}} P(\mathbf{X_0}) max_{D_1} \sum_{\mathbf{X_1}} P(\mathbf{X_1}|\mathbf{X_0}, D_1) max_{D_2} ... \\ max_{D_m} \sum_{\mathbf{X_m}} P(\mathbf{X_m}|\mathbf{X_0}, ..., \mathbf{X_{m-1}}, D_1, ..., D_m) U(\mathbf{X_0}, ..., \mathbf{X_m}, D_1, ..., D_m)$$

In this rule we alternate between summing over the variables in the information set and maximizing over the decision variable that requires the information set to compute the MEU. This equation can be further simplified as:

$$MEU(P^{\prec}) = \sum_{\mathbf{X_0}} max_{D_1} \sum_{\mathbf{X_1}} max_{D_2} ... \sum_{\mathbf{X_{m-1}}} max_{D_m} \\ \sum_{\mathbf{X_m}} P(\mathbf{X_0}, ..., \mathbf{X_m}|D_1, ..., D_m) U(\mathbf{X_0}, ..., \mathbf{X_m}, D_1, ..., D_m)$$

Let σ be the one of the possible policies, that is a sequence of decisions to be taken, for the decision making problem. Let X be the set of all random variables in the scope given by $P^{\prec}$, and $\Phi(X, \sigma)$ be the unnormalized joint distribution for X given σ. Then the above Sum-Max-Sum rule can be rewritten as:

$$MEU(P^{\prec}) = max_{\sigma} \sum_{x \sim X} \Phi(x, \sigma) U(x, \sigma) \tag{2.6}$$

Thus, the optimal policy σ^* can be obtained as $\sigma^* = max_{\sigma} \sum_{x \sim X} \Phi(x, \sigma) U(x, \sigma)$. The expansion of a sum product max network yields an expression that is identical to the Sum-Max-Sum rule. Hence, for all the possible decision policies, the SPMN must return the policy that maximizes the expected utility by following an optimal policy.

2.3.2 SPMN Validity and Properties

The term $S(x)$ for an SPMN S represents that the indicators specify the complete state x. If a partial evidence e is given, it is abbreviated as $S(e)$. The expected utility value for the given policy σ and evidence e is given as $EU(e, \sigma) = \sum_{x \sim e} \Phi(x, \sigma) U(x, \sigma)$, that is a marginal of expected utility of all the states that are consistent with the evidence. We redefine validity as:

Definition 2.3.2 (SPMN validity) *A sum product max network S is valid iff $S(e) = \sum_{x \sim e} S(x)$ for all evidence e.*

Corollary 2.3.1 *A sum product max network S is valid iff the evaluation of the network for a given evidence e maximizes the expected utility for that evidence, that is $S(e) = max_\sigma EU(e, \sigma)$*

Proof: The expansion of an SPMN S can be given using the expression $max_\sigma \sum_k s_{k,\sigma} \Pi_k(...) U(k, \sigma)$, where $\Pi_k(...)$ is the monomial over the indicator variables and $s_{k,\sigma} \geq 0$ is its coefficient. For evaluating SPMN to get the expansion, the indicator variables that are consistent with the evidence are set to 1 and the remaining to 0. A bottom-up pass is performed by applying the operators at each node on the values of its children. Thus, an SPMN is valid if it is able to correctly compute the MEU for any given evidence. For this purpose, the states x and the monomials given by expansion must be in one-to-one correspondence. Thus, each monomial is non-zero in exactly one state (condition 1), and each state has exactly one non-zero monomial in it (condition 2). So, from condition 2, $S(x)$ would be equal to $max_\sigma s_{x,\sigma} U(x, \sigma)$ corresponding to the monomial that is non-zero. Therefore, $\sum_{x \sim e} S(x) = \sum_{x \sim e} max_\sigma s_{x,\sigma} U(x, \sigma) = max_\sigma \sum_k s_{k,\sigma} U(k, \sigma) n_k(e)$, where $n_k(e)$ is the number of states x consistent with e for which $\Pi_k(x) = 1$. But $n_k(e) = 1$ from condition 1 if the state x for which $\Pi_k(x) = 1$ is consistent with the evidence and $n_k(e) = 0$ in all other cases. Thus, we get $\sum_{x \sim e} S(x) = max_\sigma \sum_{k:\Pi_k(e)=1} s_{k,\sigma} U(k, \sigma) = S(e)$ which is the condition for validity. The coefficients $s_{x,\sigma}$ represent the probability distribution $\Phi(x, \sigma)$. As stated previously, $\sum_{x \sim e} S(x) = \sum_{x \sim e} max_\sigma s_{x,\sigma} U(x, \sigma) = max_\sigma \sum_{x \sim e} \Phi(x, \sigma) U(x, \sigma) = max_\sigma EU(e, \sigma)$. Thus, we also have $S(e) = max_\sigma EU(e, \sigma)$ that shows that the expansion of the network maximizes the expected utility.

The maximum expected utility is computed by performing a bottom-up pass after setting the indicator variables for the given evidence. The policy given by the network is found by performing a top-down trace by choosing the edges that maximize the decision nodes. Similar to the discussion in the Section 2.2 for SPNs, we discuss a few properties for SPMNs to ensure its validity. This guarantees that the evaluation of the SPMN gives its Maximum Expected Utility value.

Definition 2.3.3 (Sum-completeness) *(Melibari et al., 2016) An SPMN is sum-complete iff each child of a sum node has the same scope.*

Definition 2.3.4 (Decomposability) *(Melibari et al., 2016) An SMPN is decomposable iff no variable appears in the scopes of more than one child of a product node.*

Definition 2.3.5 (Max-complete) *(Melibari et al., 2016) An SPMN is max-complete iff each child of a max node has the same scope.*

Definition 2.3.6 (Max-uniqueness) *(Melibari et al., 2016) An SPMN is max-unique iff each max node corresponding to a decision variable appears at most once in every path from the root to the leaves.*

Theorem 2.3.2 *(Melibari et al., 2016) An SPMN is valid if it is sum-complete, decomposable, max-complete, and max-unique.*

2.3.3 Structure Learning

A structure learning for Sum Product Max Networks, known as LearnSPMN, was introduced by (Melibari et al., 2016) that learns the SPMN structures given the dataset and the partial order. Here we discuss the modified version of the LearnSPMN algorithm (Tatavarti, 2020) in Algorithm 2, that leads to an improvement in the results as compared to the original algorithm. The working of the LearnSPMN algorithm is illustrated in the Figure 2.6.

The LearnSPMN algorithm takes as input a dataset D having variables V and partial order $P^{\prec}$. Initially, the current information set for this recursive algorithm is the first set in the partial order. The

Algorithm 2: LEARNSPMN

Input: D: Dataset, V: Variables, i: Information set index, $P^{\prec}$: Partial order
Output: learned SPMN

if $|V| = 1$ **then**
  if variable $v \in V$ is utility **then**
    $u \leftarrow$ estimate $Pr(V = True)$ from D
    return utility node with value u
  else
    return smoothed univariate distribution over V
if $V \cap P[i] = \phi$ **then**
  $i \leftarrow i + 1$
$V_R \leftarrow P^{\prec}[i+1] \cup P^{\prec}[i+2] \cup P^{\prec}[i+3]...$
if $P^{\prec}[i]$ is a decision variable **then**
  for $v \in$ decision values of$P[i]$ **do**
    $D_v \leftarrow$ subset of D where $P^{\prec}[i] = v$
  return MAX_v **LearnSPMN(**$D_v, V_R, i+1, P^{\prec}$**)**
else
  Partition variables V into independent subsets S
  Merge together subsets having variables $\in V_R$
  if $|S| > 1$ **then**
    return $\prod_j$ **LearnSPMN(**$D_j, V_j, i, P^{\prec}$**)**
  else
    partition D into clusters D_j of similar instances based on the values in P[i]
    return $\sum_j \frac{|D_j|}{|D|} \times$ **LearnSPMN(**$D_j, V, i, P^{\prec}$**)**

algorithm checks if the variable set V contains only one variable. In this case, if that single variable is a utility value, then the algorithm returns a utility node with an estimated value of $Pr(V = True)$. Otherwise, it returns a smoothed univariate distribution for that variable. If the variable in the current information set $P^{\prec}[i]$ is a decision variable, then the algorithm returns a max node having sub-SPNs as its children that were learnt from the data slices corresponding to each possible decision value for that decision variable.

If the current information set consists of random variables, an independence test is performed on the entire variable set V to form independent subsets. Then the subsets having variables in the future information sets are merged together. Now if there are more than one subset present, then a product

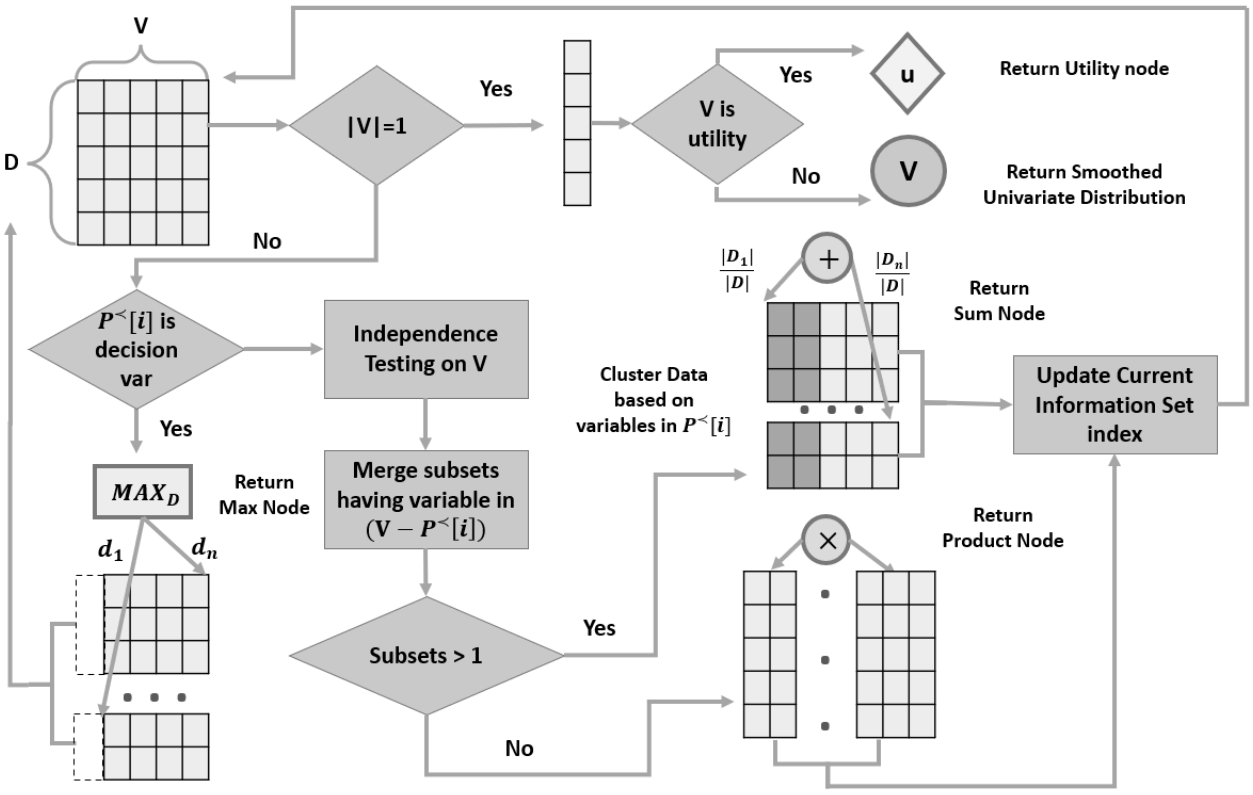


Figure 2.6: An illustration of the LearnSPMN algorithm

node is returned. The children of this product node are learnt from the data corresponding to the subset partitions. If no such subsets exist, then the dataset is split into clusters of similar instances by considering the variables in the current information set. The SPNs learnt from these clusters form the children of a sum node whereas the proportion of the clusters are the edge weights.

2.4 Anytime Algorithms

In many reasoning applications, finding the optimal or the exact solution might not be necessary. The inherent complexity of such intelligent reasoning systems makes it undesirable or infeasible to provide results in a reasonable amount of time. In such systems, the time taken to compute the results might be more critical than the quality of the results. Thus, computing approximate results in the given time is

sufficient in these cases. So, to allow a trade-off between the computation time and the performance of the system, anytime algorithms were introduced.

The term 'Anytime Algorithms' was first introduced by (Dean & Boddy, 1988) in the context of their work on time-dependent planning. Similarly, 'Flexible Computing' was presented by (Horvitz, 1987) to solve time-critical decision problems. But these approaches are built upon the idea that the quality of the results improve gradually as the computation time increases. This ensures a trade-off between the output quality and the resource consumption.

As mentioned in (Zilberstein, 1996), following are some of the desirable properties that an anytime algorithm must be based on:

- *Measurable Quality*: We should be able to precisely determine the quality of the approximate result.
- *Recognizable Quantity*: We should be able to determine the quality of the result easily at run time.
- *Monotonicity*: The quality of the result should be a non-decreasing function of time and the input quality. This can be guaranteed by simply returning the best result so far rather than the last generated result when the quality is recognizable.
- *Consistency*: The quality of the result should be correlated with the computation time and input quality. The output quality might not be deterministic, but the variance in the quality must be narrowed.
- *Diminishing Returns*: The improvements in the output quality are larger at the early stages, but it diminishes over time.
- *Interruptibility*: The algorithm can be stopped anytime and provide some result.
- *Preemptability*: The algorithm can be interrupted and resumed with less overhead.

A metric for the result quality is important to consider for constructing an anytime algorithm. Different types of algorithms tend to approach the results in completely different way. The following metrics

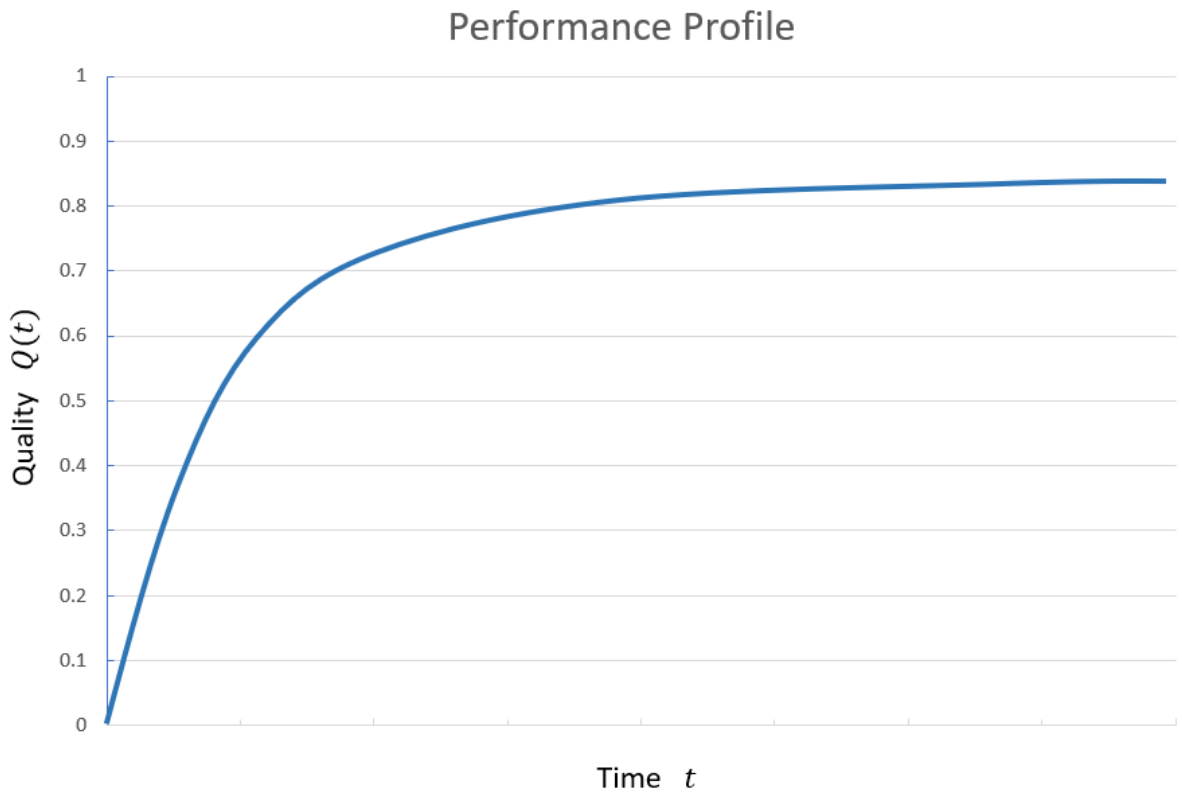


Figure 2.7: Expected performance profile for an Anytime Algorithm

have been specified in (Zilberstein, 1996; Zilberstein & Russell, 1995) for evaluating the performance of the anytime algorithms:

- *Certainty*: It is a measure of the degree of certainty that the results is correct.
- *Accuracy*: It measures how close the approximate result is to the actual answer.
- *Specificity*: It is a measure of the level of detail of the result. In this case, the result is always correct, but the level of detail increases.

The behavior of an anytime algorithm is characterized by performance profiles (PPs) to allow for an efficient meta-level control of the algorithm. They can be defined as:

Definition 2.4.1 (Performance Profile) *(Zilberstein, 1996) A performance profile of an anytime algorithm, $Q(t)$, denotes the expected output quality with execution time t.*

Definition 2.4.2 (Conditional Performance Profile) *(Zilberstein & Russell, 1995) The conditional performance profile (CPP) of an anytime algorithm A is a function $CPP_A : Q_{in} \times R^+ \longrightarrow Q_{out}$ that maps input quality and computation time to the expected quality of the results.*

The improvement in the performance over the time is summarized quantitatively using PPs. They generally describe the expected output quality as a function of run time. A conditional performance profile can capture the dependency of output quality in the execution time as well as the input quality. The expected performance profiles are typically considered to be monotonically increasing functions of the time as shown in the Figure 2.7.

2.5 X-MEANS Clustering Algorithm

The well-known and simple to use clustering algorithm, the K-MEANS algorithm, has a major drawback. For this algorithm, the number of clusters (K) to be found in the data are needed to be pre-specified. Knowing the value of K is not always possible as it would require expert knowledge to understand the data domain and know the optimal number of clusters that are present in the data. Thus, a clustering algorithm that is able to discover the number of clusters present in the data is needed.

The X-MEANS algorithm, introduced by (Pelleg & Moore, 2000), serves this purpose. This algorithm is able to determine the optimal number of clusters using measures like Bayesian Information Criterion (BIC) (Schwarz, 1978). The algorithm tries to search the number of clusters to optimize BIC. The Bayesian Information Criterion or the Schwarz Information Criterion is a criterion used to select the best model from a given finite set of models. The model having a greater BIC score is preferred.

Definition 2.5.1 (Bayesian Information Criterion) *The BIC score, in (Schwarz, 1978), is formally defined as:*

$$BIC(M_j) = log\text{-}likelihood(D_j) - \frac{p_j}{2} log(R) \tag{2.7}$$

Where M_j is one of the models available for selection, D_j is the data for that model, p_j is the number of parameters in M_j and R is the number of data instances.

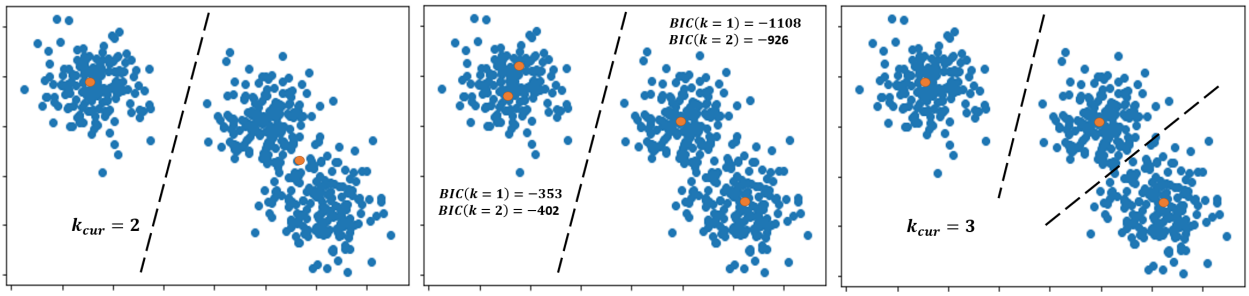
Algorithm 3: X-MEANS

Input: D: Dataset
Output: Clusters C

$k_{cur} = 2$
while True **do**
    Cluster set $C \leftarrow$ K-MEANS(D, k_{cur})
    $k' = k_{cur}$
    for c in C **do**
        $bic_{prev} \leftarrow \text{BIC}(\{c\})$
        $C' \leftarrow$ K-MEANS$(c, 2)$
        $bic_{new} \leftarrow \text{BIC}(C')$
        if $bic_{new} > bic_{prev}$ **then**
            $k' = k' + 1$
    if $k' > k_{cur}$ **then**
        $k_{cur} = k'$
    else
        return C

The algorithm 3 describes the functioning of the X-MEANS algorithm. The algorithm starts by considering the lowest possible number of clusters (k_{cur}) in the data, that is two clusters. This algorithm works in two phases: *Improve Parameters* and *Improve Structure*. The *Improve Parameters* phases runs the K-MEANS algorithm for the current number of clusters until convergence. The *Improve Structure* phase determines whether new centroids should appear and where should they appear. In this phase, the current BIC score is estimated for each cluster that is currently found. Then for each of these clusters, the K-MEANS algorithm further tries to split the cluster into two parts. If the BIC score for that newly found split is greater than the one for that original cluster, then the new centroids are considered and the value of k_{cur} is incremented. Once all the clusters are checked for improvement, if the value of k_{cur} has increased, then the algorithm proceeds with the *Improve Parameters* phase again with the newly found k. The algorithm stops if the value of k_{cur} doesn't increase further.

The X-MEANS algorithm is illustrated in the Figure 2.8. The figure 2.8a shows that the algorithm has currently found two clusters for the value of $k_{cur} = 2$. Then for each of these two clusters, the K-MEANS algorithm tries to split them in two parts. The new and old BIC scores for each cluster are then



(a) (b) (c)

Figure 2.8: Illustration of the X-MEANS algorithm. Initially, the algorithm starts with two clusters as shown in the sub-figure (a). The sub-figure (b) shows comparison of the current BIC score with the new ones for each cluster. The updated clusters are shown in sub-figure (c).

computed. As seen in the figure 2.8b, there is an improvement in the BIC score for the right-side cluster, but the score for the left cluster degrades. Thus, the value of k_{cur} is increased to 3 and the new clusters are found in the data.

Chapter 3

Related Work

In this chapter, we discuss some previously done research work in the field of SPNs and SPMNs that relate to our work and also distinguish our thesis work from the currently existing work.

Since the inception of the Sum Product Networks by (Poon & Domingos, 2011), multiple algorithms have been presented for learning their structure. This includes the cluster architecture algorithm given by (Dennis & Ventura, 2012), the LearnSPN algorithm (Gens & Domingos, 2013) that learns tree SPNs with leaves as univariate distributions, SPNs using discriminative learning by (Gens & Domingos, 2012), Indirect-Direct SPNs (ID-SPNs) by (Rooshenas & Lowd, 2014) having leaves as Arithmetic Circuits representing multivariate distributions, a bottom-up greedy approach by (Peharz et al., 2013), SPN-SVDs (Adel et al., 2015) based on rank-one sub-matrices, etc. Most of these works have focused on designing structure learning algorithms that would increase the log-likelihood of the learned SPNs as compared to the previously existing algorithms. There isn't much work that emphasizes on controlling the sizes of the learned networks or reducing the learning times.

A greedy structure search strategy, SearchSPN (Dennis & Ventura, 2015) tries to convert the tree SPNs into graph SPN for the purpose of increasing the log-likelihood. It proposes the SearchSPN algorithm that acts as a post-processing step on the tree structured SPNs given by the LearnSPN algorithm for converting the SPNs to a graph network. This approach improves the log-likelihood over the input network. But it also leads to an increase in the number of nodes in the structure along with the

total computation time due to the further processing. Our approach doesn't perform any post-processing on the SPNs, either to increase the log-likelihood or to decrease the network sizes, but the anytime approach shows an improvement in the log-likelihood of the networks learnt in less time as compared to the LearnSPN algorithm.

A few modifications have been suggested by (Vergari et al., 2015) to improve or simplify the LearnSPN algorithm. This work suggests binary splits for all the sum nodes in the network for allowing deeper SPN structures to be learned. This approach shows some improvement over the LearnSPN log-likelihood. For yielding simpler SPNs with fewer number of edges, they suggest the use of Chow Liu Trees (Chow & Liu, 1968) as the leaves instead of performing a naïve factorization for allowing the recursive LearnSPN algorithm to stop if the number of instances in the dataset falls below a threshold value. Although the log-likelihood is improved, this approach would lead to an increase in run time due to the learning of CLTs. Another suggestion is to make use of bagging where multiple sub-SPNs are learned using bootstrapped samples. Although the work fails to present the impact of this approach on the network sizes, intuitively this approach would lead to an increase in the size. The SPFlow library (Molina et al., 2019) implements the LearnSPN algorithm utilizing the first suggestion by using the K-means algorithm with two clusters for the sum nodes. This algorithm forms the baseline for our work.

An alternative approach for variable splitting is proposed by (Di Mauro et al., 2017). They present a method, Random Greedy Variable Splitting (RGVS), for the variable splitting operation. For the RGVS method, they randomly pick a subset from the scope variables V having the size $\sqrt{|V|}$. The independence testing uses a greedy G-test method. They limit the possible number of independent subsets to only two groups which might not always be the case. Also, all the other remaining variables from V are assigned to one of the groups selected at random. This method shows reduction in the learning time as well as a drop in the performance of the model. But it doesn't display a gradual trend in the log-likelihood when the number of variables in the random subset that are considered for splitting are varied. Our approach draws inspiration from this method for selecting a subset from V for the variable splitting task but allows

multiple independent subsets to be formed using a non-greedy method and avoids allocating the remaining variables randomly to one of the subsets.

To obtain a more expressive and accurate model, two alternate adaptive instance clustering methods have been given by (Liu & Luo, 2019). They present the method 'MSH' based on Mean Shift (Fukunaga & Hostetler, 1975) and another method 'DBSH' based on DBSCAN (Ester et al., 1996) having lower complexity. This optimized clustering avoids the number of clusters to be pre-specified. They impose a threshold on the number of clusters, which if crossed set the number of clusters to two. Our approach uses the X-MEANS algorithm that does not need the number of clusters to be pre-specified and limits the number of clusters to the maximum allowed value.

The algorithm LEARNSPMN is introduced by (Melibari et al., 2016) for learning the Sum Product Max Networks that are used for decision making given its domain data and the partial order. Extensions of SPMNs for sequential domains, RSPMNs (Tatavarti et al., 2021) and S-RSPMNs (Hayes et al., 2021), use a template network that reduces the network size as compared to the SPMNs. They further showcase that the size of these learned networks still remains unbounded and express a need for having a flexible control on the network sizes. Our approach tries to have a control on the size and the learning time of the networks in relation to their performance.

Chapter 4

Anytime SPN

The LearnSPN algorithm is widely used for learning the structures of the SPNs from the data. But it has no control over the size or the performance of the networks. For a given dataset, the implementation of this algorithm in SPFlow (Molina et al., 2019) would always generate the same network as the default number of clusters to be formed for a sum node is always two (Vergari et al., 2015) and all the scope variables are used for variable splitting for the product nodes. Such an algorithm would take a long time and learn complex networks for large datasets. Also due to the algorithm not being flexible, it might learn complex networks even for simpler domains. Other algorithms that were designed to reduce the size of such networks act as a post-processing step for this algorithm. This would increase the overall run time for learning the network.

In this chapter we introduce an anytime algorithm for the SPNs. This anytime approach for learning SPNs would allow us to have a control over the size and the performance of the learned networks. In the initial phases of the algorithm, this approach generates simpler networks, having smaller size and an approximate model of the data in lesser run times. As the algorithm progresses, the size of the networks increases along with their performance.

The Section 4.1 present the AnytimeSPN algorithm that returns improved SPNs at each iteration. A modified LearnSPN algorithm is presented in the Section 4.2 that learns the networks flexibly. The

Section 4.3 shows the updates to the structure of the SPNs by the AnytimeSPN algorithm. Finally, in the Section 4.4, we show that the networks generated at each iteration is valid.

4.1 The AnytimeSPN Algorithm

Algorithm 4: AnytimeSPN

Input: D: Dataset, V: Variables
Output: Series of Learned SPN

```
Set maximum cluster limit k ← 2
Set # variables for independence testing n ← √|V|
Set the first operator curOp ← sum
do
    spn ← LearnSPN*(D, V, curOp)
    Increment the parameters k and n
until log-likelihood converges
```

The Algorithm 4 shows us the anytime technique, AnytimeSPN, that gives us better SPN structures at each iteration. This algorithm requires two inputs. It takes the dataset D of the domain whose probabilistic graphical model is to be learned, along with a list of the random variables V that gives us the scope of the domain as the inputs. The algorithm then outputs a series of sum product networks such that their performance, estimated by the log-likelihood, keeps on improving as the algorithms progresses and the networks become more complex.

The AnytimeSPN algorithms consists of two parameters: the maximum cluster limit k and the number of variables used for independence testing n. The value of k denotes the upper limit on the possible number of clusters that could be formed at the sum node. This gives us a control over the branching at the sum node and leads to an increase in the complexity of the networks over time. The other parameter n gives us the size of the subset of the scope variables that are considered for independence testing for forming the product nodes. This leads to a reduction in the run time and generates approximate networks in the initial iterations. This parameter is further elaborated in the Section 4.2.

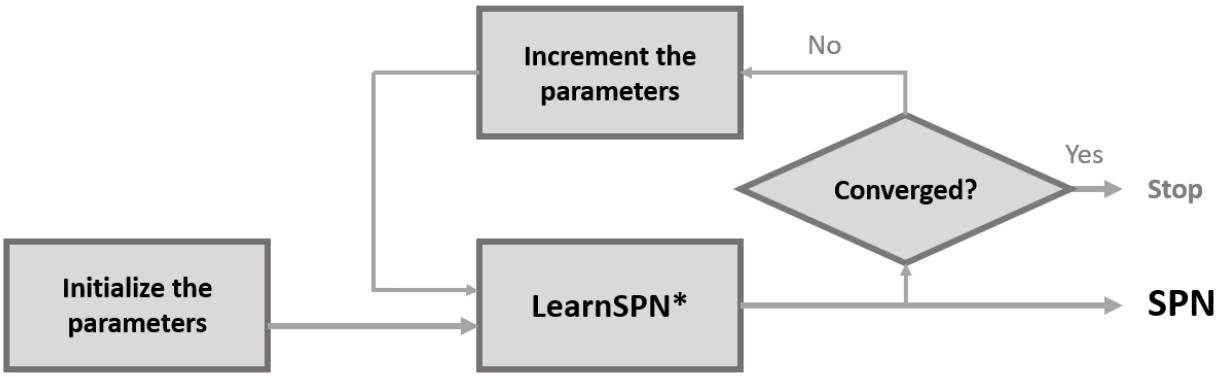


Figure 4.1: An illustration of the AnytimeSPN algorithm

The initial value of k is set to the value 2 in order to limit it to least the possible branching at the sum node. The value of n is set to the root of the number of variables present in the scope. It is set to the value $\sqrt{|V|}$ initially. The initial value of n is chosen to be $\sqrt{|V|}$ since it leads to a linear complexity for the RGVS method (Di Mauro et al., 2017). Otherwise, the complexity remains sub-quadratic. Choosing a very small fraction for n might lead to larger networks, since only a few or no variables are considered independent of others and might require variable splitting a large number of times. This in turn would increase the run time. Thus, we use the value suggested by (Di Mauro et al., 2017).

At each iteration of the algorithm, a sum product network is generated using the LearnSPN* algorithm that would be discussed in the Section 4.2. The current operation for the LearnSPN* algorithm is set to sum so that the root of the generated networks is always a sum node. The parameters k and n are incremented at each iteration after learning the SPN structure. The value of the parameter k is incremented by 1 at each step. To ensure that the increase in the performance is higher in the initial iterations, the increments for the parameter n is distributed over the first 20 iterations. This process continues until the value of n increments to the upper bound of $|V|$ and the performance (log-likelihood) of the learnt networks comes to a convergence. The algorithm terminates when the standard deviation in the log-likelihood of the past three iterations falls below the value 10^{-3}. The high-level functioning of the AnytimeSPN algorithm is illustrated in the Figure 4.1.

4.2 Modified LearnSPN for the Anytime Approach

Algorithm 5: LearnSPN*

Input: D: Dataset, V: Variables, $curOp$: current operation
Parameters: k: maximum cluster limit, n: # variables for independence testing, m: minimum number of instances to allow an operation
Output: learned SPN structure

```
if |V| = 1 then
    return smoothed univariate distribution over V
if |D| < m then
    return naïveFactorization(D, V)
if curOP = prod then
    Partition variables V[: n] into independent subsets V_j
    Distribute the remaining variables V[n :] evenly among the subsets V_j
    return Π_j × LearnSPN*(D_j, V_j, sum)
else
    partition D into clusters D_j of similar instances such that j ≤ k
    return Σ_j (|D_j|/|D|) × LearnSPN*(D_j, V, prod)
```

The sum product networks at each iteration of the AnytimeSPN algorithm are generated using the Algorithm 5. The LearnSPN* algorithm follows the outline of the LearnSPN algorithm along with the addition of the parameters k and n that are updated at each iteration of the AnytimeSPN algorithm. Along with these parameters, the algorithm also takes as input the domain dataset D, the list of scope variables V and a current operation indicator $curOp$ that indicates whether the current operation for the recursive call is to form a sum node or a product node. These two nodes are generated alternately from the root node to the leaf nodes of the network. An additional parameter m is used that gives the minimum number of instances that are in the dataset to perform an operation. The algorithm returns the SPN structure that is learned from the data by adhering to the given values of the parameters k and n.

The algorithm first checks if only one variable remains in the scope of the dataset. If this is the case, then the algorithm returns a smoothed univariate distribution over that variable to form a leaf node. If the number of instances in the dataset D falls below as threshold value of m, then the variables in V are naïve-factorized. To naïve-factorize the variables, a product node is returned whose children are the

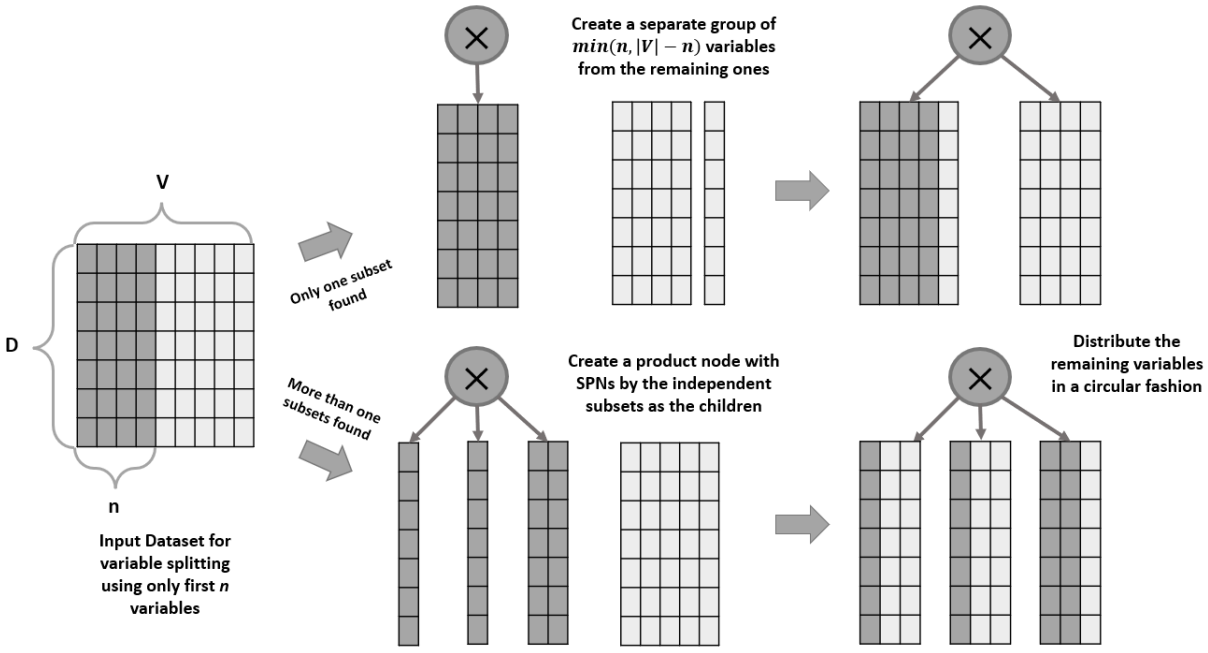


Figure 4.2: An illustration of the variable splitting for the LearnSPN* algorithm

smoothed univariate distributions for each of the variables in the set V. Otherwise, the algorithm performs independence testing as explained in the section 4.2.1, if the $curOp$ is *prod*. Else if the $curOp$ is *sum*, then clustering as given in the section 4.2.2 is performed.

4.2.1 Variable Splitting for LearnSPN*

The algorithm LearnSPN* makes use of the Randomized Dependence Coefficient (RDC) method for the independence testing task to generate the product nodes in the networks. The parameter n gives the number of scope variables from $|V|$ that would be used for the variable splitting operation using independence testing between the variables. An overview of the flexible variable splitting method that we use for our algorithm is given in the Figure 4.2.

In this approach, only the first n variables from the scope variables V are used for the independence testing. We denote this selected set as $V[:n]$ and the set of remaining variables as $V[n:]$. Then the set $V[:n]$ is partitioned into independent subsets V_j of variables. In the case if only one such subset is found, another subset is created using $min(n, |V| - n)$ number of variables from the set $V[n:]$. Now if any

variables are remaining in the set $V[n\ :]$, i.e. $V[n\ :] > 0$, then those variables are evenly distributed amongst the variable subsets in a circular fashion, starting from the first one. If the value of the parameter n exceeds the value of $|V|$, then n is set to a default value of $|V|$. Finally, the sub-trees learned from each of the subsets are assigned as the children of a product node. We avoid selecting the n variable randomly from the scope, because it may cause the log-likelihood to drop due to the random nature.

This approach would improve on the RGVS method (Di Mauro et al., 2017), given the fact that we do not restrict the number of children for a product node to two. Also, we do not assign all the remaining variables to one of the subsets selected at random which might lead to sudden changes in the performance of the AnytimeSPN algorithm over the duration.

4.2.2 Clustering for LearnSPN*

The children for the sum nodes are found by clustering the given dataset into groups of similar instances. The LearnSPN* algorithm provides a control over the possible number of branches at the sum nodes of the network.

The maximum number of clusters at any sum node in the network is restricted to an upper limit that is given by the parameter k. These clusters are formed by a slightly modified version of the X-means clustering algorithm that was discussed in the Section 2.5. In addition to the data D being clustered, this new algorithm termed as X-meansWithLimit also takes as input the parameter k that restricts the partitioning to k clusters. The X-meansWithLimit algorithm is given by the algorithm 6.

For a given dataset, the X-meansWithLimit algorithm initially tries to partition the dataset into two clusters. If clusters are found, in the next iteration it further tries to partition the data in each of the clusters of the previous iteration again into two clusters. If the BIC score of the new model including a split within the cluster is better than that of the previous model where no split is considered, then the number of clusters found is incremented accordingly. After evaluating each cluster from the previous iteration, if the number of clusters newly found equals or exceeds the limit k, then k clusters from the dataset are returned. Otherwise, if the number of the newly found clusters exceeds the number of clusters

Algorithm 6: X-MEANSWITHLIMIT

Input: D: Dataset k: Upper Limit on the Number of Clusters
Output: Clusters C

```
k_cur = 2
while True do
    Cluster set C ← K-MEANS(D, k_cur)
    k' = k_cur
    for c in C do
        bic_prev ← BIC({c})
        C' ← K-MEANS(c, 2)
        bic_new ← BIC(C')
        if bic_new > bic_prev then
            k' = k' + 1
    if k' ≥ k then
        Cluster set C ← K-MEANS(D, k)
        return C
    if k' > k_cur then
        k_cur = k'
    else
        return C
```

found in the previous iteration, then the process is repeated again. Else the algorithm is terminated and the clusters from the previous iteration are returned.

Thus, such a process of clustering is allowed to continue while the number of clusters found is less than k. The sub-SPN structures learned from each of the clusters form the children of the sum node.

4.3 Understanding ANYTIMESPN with an Example

To gain a better understanding of how the ANYTIMESPN algorithm affects the structure of the Sum Product Networks, consider an example of the *NLTCS* dataset. The National Long Term Care Survey (*NLTCS*) data consist of 16 binary variables. This is a detailed longitudinal survey data that makes observations of health and functional status of adults that are of age 65 years or older. It measures a person's ability to perform various daily living activities.

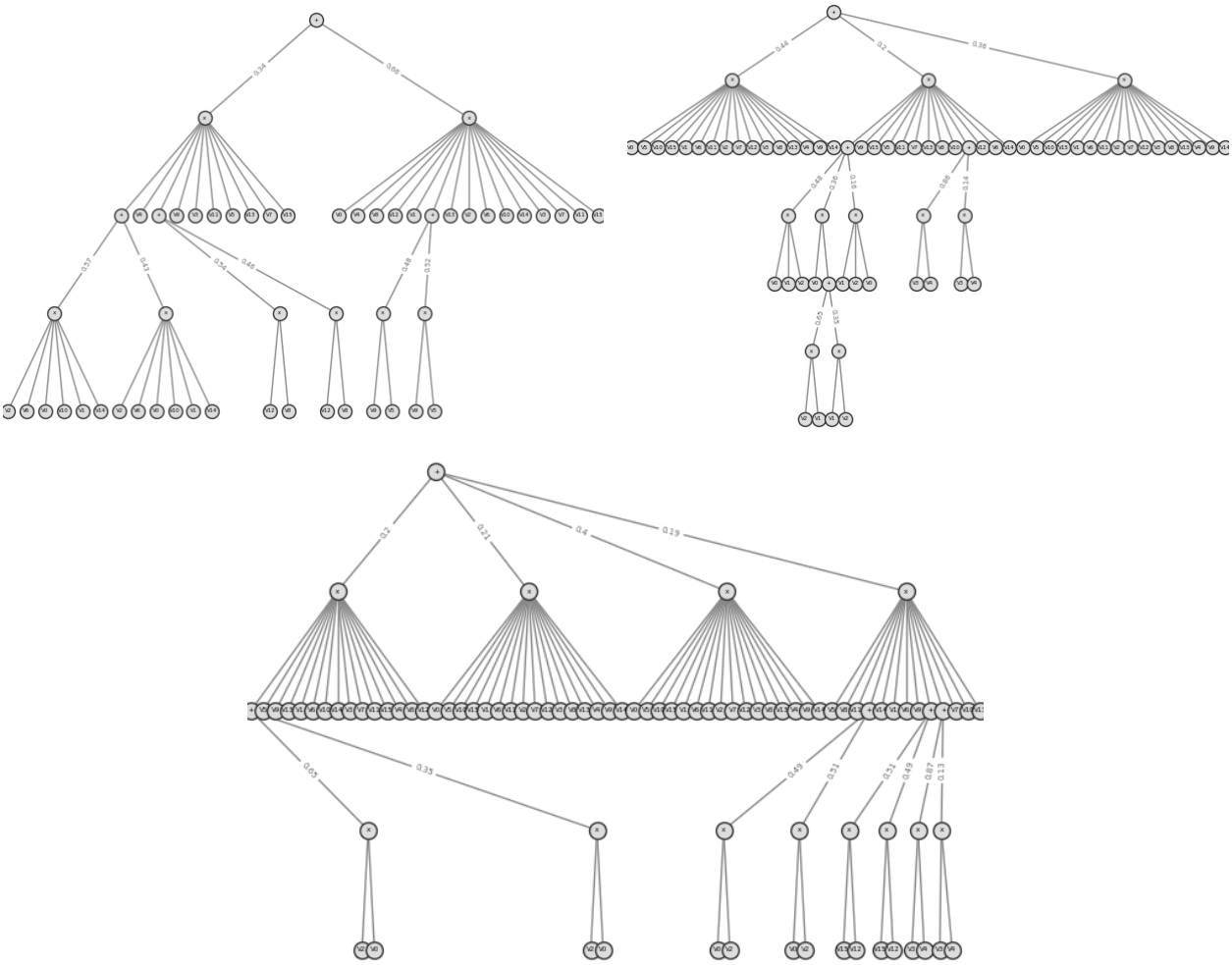


Figure 4.3: Sum Product Network structures given by the AnytimeSPN algorithm for the *NLTCS* domain. The structures from the first three iterations are shown.

The Figure 4.3 shows the changes in the learned structures of the SPNs for the *NLTCS* dataset for the first three iterations of the AnytimeSPN algorithm. As seen, the first network is restricted to only two branches at the sum nodes of the network allowing an approximate representation of the probability distribution. In the next iterations, the allowed upper limit on the number of branches can be seen to be increased to 3 and then 4. These structures show an improvement in the representation of the distribution over the previous iteration. This is understood by the increase in the number of network parameters, that is the number of the edge weights at the sum nodes. Also, the number of nodes in the networks and hence the network complexity is seen to be increasing.

4.4 Validity of the Generated SPNs

Theorem 4.4.1 *The SPNs returned at each iteration of the* AnytimeSPN *algorithm are valid.*

The AnytimeSPN algorithm makes use of the LearnSPN* algorithm that utilizes the two parameters k and n. We need to show that for any combination of the parameters k and n in the given ranges, the SPN remains complete and decomposable, thus ensuring validity. So, we prove by induction from leaves to the root of the network that the SPNs at each iteration are valid.

Base Case:

Validity is trivially true in the case of the leaf nodes since they only represent the univariate distributions and are not affect by the parameters of the algorithm.

Induction Hypothesis:

Let n^0 be an internal node having children $n^1, ..., n^l$. Following the notations in (Poon & Domingos, 2011), the scope of n^0 is given as V^0, a state of V^0 as x^0, the expansion of the sub-network rooted at n^0 as S^0, and the unnormalized probability of x^0 under S^0 as $\Phi^0(x^0)$. The same notations apply for the other nodes as well. Thus, by induction hypothesis, the expansion of SPN rooted at n^l is $S^l = \sum_{x^l} \Phi^l(x^l)\Pi(x^l)$.

Induction Step:

If the node n^0 is the product node in the case of naïve-factorization, then its children $n^1, ..., n^l$ consists of leaf nodes representing univariate distributions of each distinct variable in V. Thus, $V^i \cap V^j = \phi$, where V^i and V^j are the scopes of distinct children of n^0. This ensures decomposability of the node. Also, $S^0 = (\sum_{x^1} \Phi^1(x^1)\Pi(x^1)) \times ... \times (\sum_{x^l} \Phi^l(x^l)\Pi(x^l))$ which is its network polynomial.

Consider the sub-SPN rooted at the sum node n^0. This node could have 2 to k number of children. All these clusters would definitely share the same scope since they are discovered within the same dataset which would ensure completeness irrespective of the parameter k. Thus, if it has k children, then the expansion $S^0 = w_{01} \sum_{x^1} \Phi^1(x^1)\Pi(x^1) + ... + w_{0k} \sum_{x^k} \Phi^k(x^k)\Pi(x^k)$. If the scopes of all the children are same, $V^0 = V^1 = ... = V^k$, then a one-to-one correspondence is established between the monomials

of the expansions S^0 and the states of V^0. Therefore, $S^0 = \sum_{x^0}(w_{01}\Phi^1(x^0) + ... + w_{0k}\Phi^k(x^0))\Pi(x^0)$. This represents the network polynomial and maintains validity.

Now consider the case of n^0 being a product node. Let the independence testing performed on the subset of first n variables in V^0 find l independent subsets having scopes $V^1, ..., V^l$ where $l \leq n$. Since the subsets are independent of each other, $V^i \cap V^j = \phi$, for all distinct subset pairs V^i and V^j. The distribution of remaining $|V| - n$ variables ensures that the subsets are still disjoint and the product node n^0 having children of scopes $V^1, ..., V^l$ is decomposable. Thus, the expansion of sub-SPN rooted at n^0, $S^0 = (\sum_{x^1} \Phi^1(x^1)\Pi(x^1)) \times ... \times (\sum_{x^l} \Phi^l(x^l)\Pi(x^l))$, is its network polynomial since $V^i \cap V^j = \phi$ for all distinct subset pairs V^i and V^j.

Thus, the introduction of the parameters k and n to the structure learning algorithm for SPNs do not interfere with the completeness and decomposability of the internal nodes irrespective of the parameter values. Since the network remains complete and decomposable at each iteration of the AnytimeSPN algorithm, the algorithm always generates a valid network structure.

Chapter 5

Anytime SPMN

The LearnSPMN algorithm, for learning the structures of the Sum Product Max Networks from the data, was introduced by (Melibari et al., 2016). But this algorithm has no bound or control over the size or the performance of the learnt networks. The implementation of the LearnSPMN algorithm is provided in the *spmn* branch of the *https://github.com/SwarajPawar/SPFlow.git* repository. Considering the suggestion by (Vergari et al., 2015) for SPNs, this implementation always creates two clusters for the sum nodes to be formed and involves all the scope variables for variable splitting for the product nodes.

This algorithm would take a long time and would learn complex networks for large datasets. Since the algorithm is not flexible, it might learn much complex networks even for simple domains. The size of the networks increases exponentially for sequential domains. The RSPMNs (Tatavarti et al., 2021) and S-RSPMNs (Hayes et al., 2021) introduce a recurrent template network for perfectly observable and partially observable environments respectively. But still the size of these template networks remains unbounded.

In this chapter we introduce an anytime algorithm for learning the SPMNs. This anytime approach for SPMNs would allow us to have a control over the size of the learned networks by trading the performance of the networks. This approach would generate simpler networks in the initial iterations, that have smaller size and approximately model the data in lesser computation times. As the algorithm progresses, the performance of the networks increases by adding on to the network complexity.

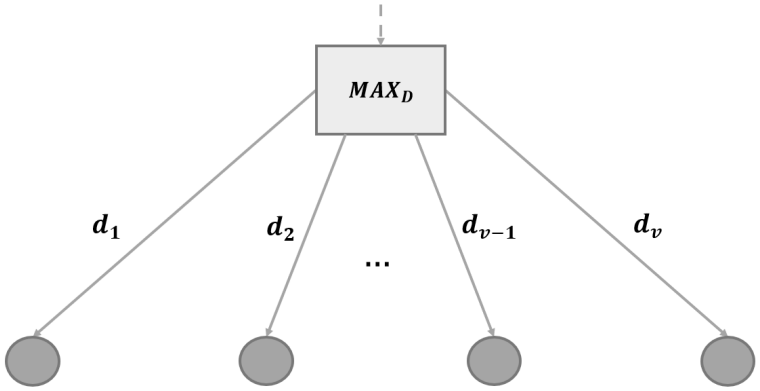
Initially, we introduce a few modifications to the structure of the SPMNs in the Section 5.1. The Section 5.2 presents the AnytimeSPMN algorithm that returns better networks at each iteration. The modified LearnSPMN algorithm used by the AnytimeSPMN algorithm is then presented in the Section 5.3 that helps to learn the networks flexibly. The Section 5.4 illustrates the changes in the SPMN structure by the AnytimeSPMN algorithm with an example. In the Section 5.5, we show that the SPMNs generated at every iteration are valid. In the last Section 5.6, we explain the evaluation of the policy from the generated networks.

5.1 Modifications to the SPMN Structure

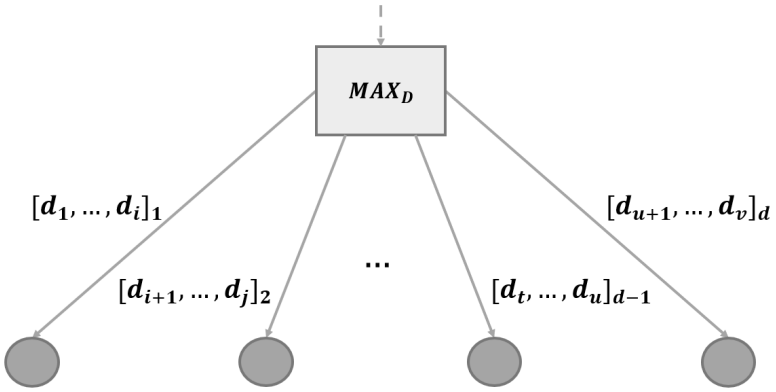
According to the definition of SPMNs, Definition 2.3.1, each max node in the network corresponds to one of the decision variables and each outgoing edge from a max node is labeled with one of the possible values of the corresponding decision variable. Thus, the number of branches for a particular max node is fixed and is equal to the number of decision values for the decision variable that the node represents. Having a control over the branching factor at nodes is necessary as it allows us to have a control over the size of the SPMN structure.

Thus, we introduce a modification to the structure of the max nodes and limit the possible number of branches to d. The parameter d gives the number of children that a decision node can have. But the value of d needs to be less than or equal to the possible number of decision values v. Due to this modification, the decision values are distributed amongst the d groups. Now in this case, the outgoing edges from the max nodes are labeled with one of the groups of the decision values rather than labelling them with a single decision value. This modification to the structure of the network has been illustrated in the Figure 5.1. To evenly distribute the decision values among the d groups, the distribution is performed in a circular fashion.

If the child node corresponding to the group $d_i, ..., d_j$ yields the maximum value, then it means that any decision value from that group at random can be considered for the policy. Hence now we do not



(a) The original SPMN structure having a separate branch for each decision value and need v branches



(b) A modified decision node where branches a group of decision values and need less than v branches

Figure 5.1: Modification to the SPMN decision node structure

obtain a policy having a definite decision value for each of the decision variables, but we get an approximate stochastic policy that gives a set of possible decision values for a decision variable.

This modification, thus, yields approximate models giving approximate policies at the initial iterations of the anytime algorithm. As the value of the parameter d is increased, the number of decision values per group are gradually reduced. Finally, when the value of d reaches the maximum possible value of v, each of the groups has only one possible decision value left. Thus, the models become less approximate and return better policies over the duration, which is the aim of an anytime technique. When the parameter d reaches v, the model returns an optimal policy.

Algorithm 7: AnytimeSPMN

Input: D: Dataset, V: Variables, $P^{\prec}$: Partial order

Output: Series of Learned SPMN

Set maximum cluster limit $k = 2$
Set maximum decision node branches $d = 2$
Set # variables for independence testing $n = \sqrt{|V|}$
Set maximum depth after decision nodes $d_{max} = 1$
do
    $spmn \leftarrow$ **LearnSPMN*(**$D, V, 0, null$**)**
    Increment the parameters k, n, d and $depthmax$
until log-likelihood converges

5.2 The AnytimeSPMN Algorithm

Similar to the algorithm 4 used for generating SPNs, the AnytimeSPMN algorithm (Algorithm 7) gives the overall procedure for learning SPMNs using the anytime technique at each iteration. A dataset D consisting of randomly generated tuples $\langle X_0, D_1, X_1, ..., D_m, X_m, U \rangle$, following the partial order $P^{\prec} = I_0 \prec D_1 \prec I_1 \prec ... \prec D_m \prec I_m$, is given as an input to the algorithm, where I is the information set, D is the decision variable and U is the utility value. The algorithm also takes as input the list of variables V present in the scope of the dataset domain along with the partial order $P^{\prec}$ of the variables. This algorithm outputs a series of learnt SPMNs such that there is an improvement in the performance at each iteration as the network complexity increases.

Additionally, the AnytimeSPMN algorithm consists of the four controlling parameters: k, d, n and d_{max}. The parameters k represents the upper limit on the maximum number of clusters allowed while creating a sum node and the parameter n represents the number of variables that are used for the independence testing operation for creating the product nodes. These two parameters have the same semantics as explained previously for the algorithm 4 that was introduced for the SPNs. A new parameter d gives the maximum number of outgoing edges allowed for each decision node present in the SPMN. Each of these edges could have a group of decision values assigned to it as discussed in the Section 5.3. The decision value for that decision variable is selected randomly from the set of values linked to the edge

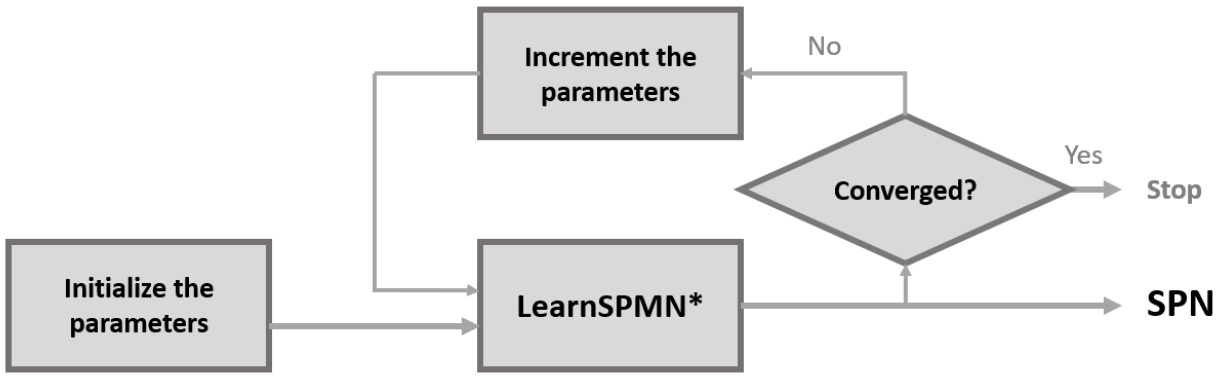


Figure 5.2: An illustration of the AnytimeSPMN algorithm

that gives the maximum value during evaluation. The value of the parameter d_{max} gives the maximum possible depth allowed from the point where no decision variables appear in the scope of the branch. This allows us to control the depth and hence the size of the network from that point. Truncating the depth of the networks from the chosen point avoids interfering with the max-uniqueness of the network.

The algorithm first initializes these parameters to their lowest feasible values. The value of k is initialized to two clusters for the sum nodes and the parameter n is initialized to the value of $\sqrt{|V|}$ as discussed previously in the Section 4.1. The possible number of edges for the decision nodes is initially restricted to only two edges. The maximum possible depth d_{max} of the branches after the last decision node in the branch is limited to one. These parameters are given to the modified LearnSPN algorithm, termed as LearnSPMN* algorithm, to learn the SPMN structures at each iteration. After an SPMN is learned, the values of the parameters k, d, n and d_{max} are incremented. The value of n is incremented till it equals to $|V|$, while the value of d is incremented till it reaches the total number of decision values possible for the decision variables. The values for k and d_{max} are increased by 1 without any maximum limit. This process stops when the log-likelihoods of the generated networks reaches convergence after the value of n equals $|V|$. The algorithm is said to be converged when the standard deviation in the log-likelihood of the last three networks falls below the threshold of 10^{-3}. The overall functioning of the AnytimeSPMN algorithm is illustrated by the Figure 5.2.

5.3 Modified LearnSPMN for the Anytime Approach

Algorithm 8: LearnSPMN*

Input: D: Dataset, V: Variables, i: Information set index, d_{cur}: current depth after decision nodes
Parameters: $P^{\prec}$: Partial order, k: Maximum cluster limit,
d: Maximum decision node branches,
n: # variables for independence testing,
d_{max}: Maximum depth after decision nodes
Output: learned SPMN

if $|V| = 1$ **then**
  if variable $v \in V$ is utility **then**
    $u \leftarrow$ estimate $Pr(V = True)$ from D
    return utility node with value u
  else
    return smoothed univariate distribution over V
Update i and d_{cur}
if $d_{cur} \geq d_{max}$ **then**
  return $\Pi_j \times$ naïveFactorize(D, V)
$V_R \leftarrow P^{\prec}[i+1] \cup P^{\prec}[i+2] \cup P^{\prec}[i+3]$...
if $P^{\prec}[i]$ is a decision variable **then**
  $v_{groups} \leftarrow$ distribute decision values in d groups
  for $v_g \in v_{groups}$ **do**
    $D_{v_g} \leftarrow$ subset of D where $P^{\prec}[i] \in v_g$
  return MAX$_{v_g}$ **LearnSPMN*(**$D_{v_g}, V_R, i+1, d_{cur}$**)**
else
  Partition variables $V[:n]$ into independent subsets S
  Distribute the remaining variables $V[n:]$ among the subsets $V_j \in S$
  Merge together subsets having variables $\in V_R$
  if $|S| > 1$ **then**
    return $\Pi_j \times$ **LearnSPMN*(**D_j, V_j, i, d_{cur}**)**
  else
    partition D into clusters D_j of similar instances based on the values in P[i] such that $j \leq k$
    return $\Sigma_j \frac{|D_j|}{|D|} \times$ **LearnSPMN*(**D_j, V, i, d_{cur}**)**

The LearnSPMN* algorithm, given by the algorithm 8, that is used for learning the SPMN structures flexibly at each iteration is based upon the method given by the LearnSPMN algorithm. Along with the domain dataset D, the scope variables V and the partial order $P^{\prec}$, the algorithm requires the

information set index i as the input. This is the index of the current information set from the partial order that is to be processed. At the beginning, i is set to 0 to start from the first information set. Another parameter d_{cur} is used to keep a track of the depth from the point when decision variables vanish from the scope of data to the point where the recursive algorithm currently is. This parameter is initially passed as $null$ since decision variables initially exist in the scope V. Other controlling parameters required by the algorithm are k, d, n, and d_{max} whose values are provided by the algorithm 7. The LearnSPMN* algorithm returns the learnt SPMN structure by adhering to the parameter values that are provided to it.

The algorithm 8 initially checks whether the scope V of the dataset contains only a single variable. If this is the case, then the algorithm returns a utility node if the variable in V is a utility variable. Otherwise if the variable is a random variable, then the smoothed univariate distribution over that variable is returned. The information set index i is incremented if there are no variable from the current information set $P^{\prec}[i]$ in the scope V. After updating the current information set index, all the variables from the next information set onward are stored in V_R.

5.3.1 Controlling Depth in LearnSPMN*

The two parameters d_{max} and d_{cur} are used to control the depth of the branches while learning the network recursively. The depth is the SPMNs can be controlled after a particular point in the branches. If the current scope of variables V contains any decision variables within it, it is not possible to truncate the current branch since it would not a valid policy. This is because the path from the root of the network to the utility leaf nodes must contain all the decision variables on which that utility is dependent upon for deriving a valid policy from the network. Hence, a branch in the network can be truncated only if the decision variables no longer appear within the scope of that branch.

Initially the value of the parameter d_{cur} is $null$. This indicates that the decision variables exist within the scope V. If there are no decision variables in the scope V and parameter d_{cur} is still $null$, the current branch can be truncated from this point onward. Thus, when this point is reached, the value of the

parameter d_{cur} is set to 1. If the value of d_{cur} is already an integer value, it is updated to the next integer value to indicate an increment in the level of depth of the branch.

The parameter d_{max} gives us the maximum depth that is allowed from the point from where the branch can be safely truncated. If the value of d_{cur} exceeds the limit given by d_{max}, then the algorithm returns a product nodes having $|V|$ children created by naïve-factorizing the scope variables V. This gives an approximate distribution over the variables V once the maximum allowed depth is reached.

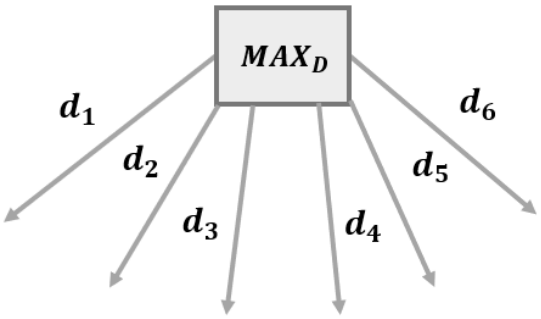
The maximum limit d_{max} is initially assigned the value and incremented after each iteration of the AnytimeSPMN algorithm. Since the allowed depth for the branches increases over time, it allows the LearnSPMN* algorithm to generate deeper and better models having more nodes and complexity. This reduces the learning time in the initial iterations and improves the performance in the later phases.

5.3.2 Decision Variable Splitting in LearnSPMN*

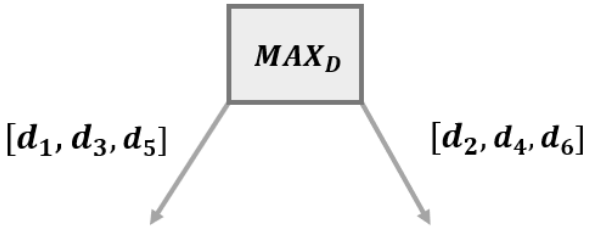
As discussed previously in the Section 5.1, the decision nodes in the network are modified to consist of a maximum of d children. Thus, the parameter d is used to control the branching of the max nodes that represent the decision variables.

The task of decision variable splitting is performed when the current information set in the partial order contains only a single decision variable $P^{\prec}[i]$. In such a case, all the distinct decision values corresponding to the variable $P^{\prec}[i]$ are discovered within the dataset D. Then d groups are created to hold the decision values. If the number of the decision values discovered within the data is less than the value of d, then the number of groups created equals the number of decision values found. After creating the groups, the discovered decision values are distributed among the groups. For each of the decision groups v_g, a subset D_{v_g} of dataset D where the value of the decision variable $P^{\prec}[i]$ belongs to that group is formed. The algorithm returns a new max node having at the most d children for the variable $P^{\prec}[i]$ whose children are the sub-SPMNs learnt from each of the created subsets D_{v_g} with the scope V_R.

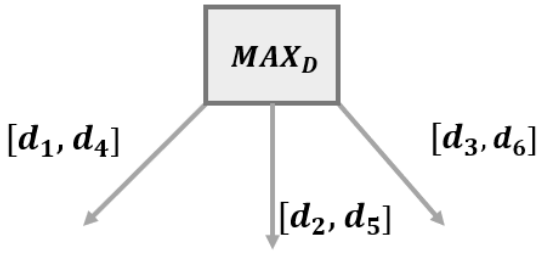
The distribution of the decision values among the d groups is done in a circular fashion for even distribution of the values. To understand how this distribution is performed, consider an example of



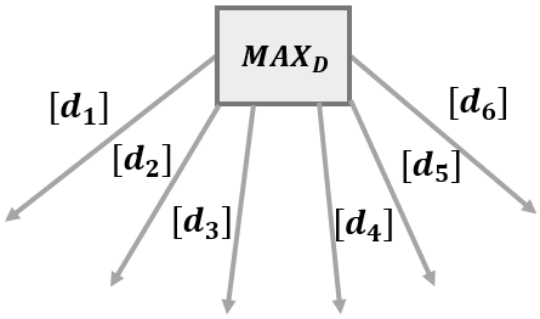
(a) The original SPMN structure for $v = 6$



(b) Modified decision node structure for $d = 2$



(c) Modified decision node structure for $d = 3$



(d) Modified decision node structure for $d = 6$

Figure 5.3: An example illustrating the changes in the grouping of the decision values as the controlling parameter d is varied

a decision variable D having the six decision values given as $\{1, 2, 3, 4, 5, 6\}$ as shown in the Figure 5.3. If the value of the parameter $d = 2$, then the distribution is done as $\{[1, 3, 4], [2, 4, 6]\}$. For $n = 3$, the grouping is $\{[1, 4], [2, 5], [3, 6]\}$. When the value of n is set to the maximum possible value of v, we would get 6 decision value groups as $\{[1], [2], [3], [4], [5], [6]\}$. Thus, when the value of the parameter d reaches the maximum possible value, the branching for the modified max node becomes similar to the branching in the original SPMN structure.

As evident from the example, the approximation level of the expected utility is reduced as the value of d is increased. Thus, the parameter d also controls the quality of the policy along with the network size. So, as the parameter d is increased, the maximum utility obtained over the groups of the decision variables also increases till it reaches the optimal MEU value.

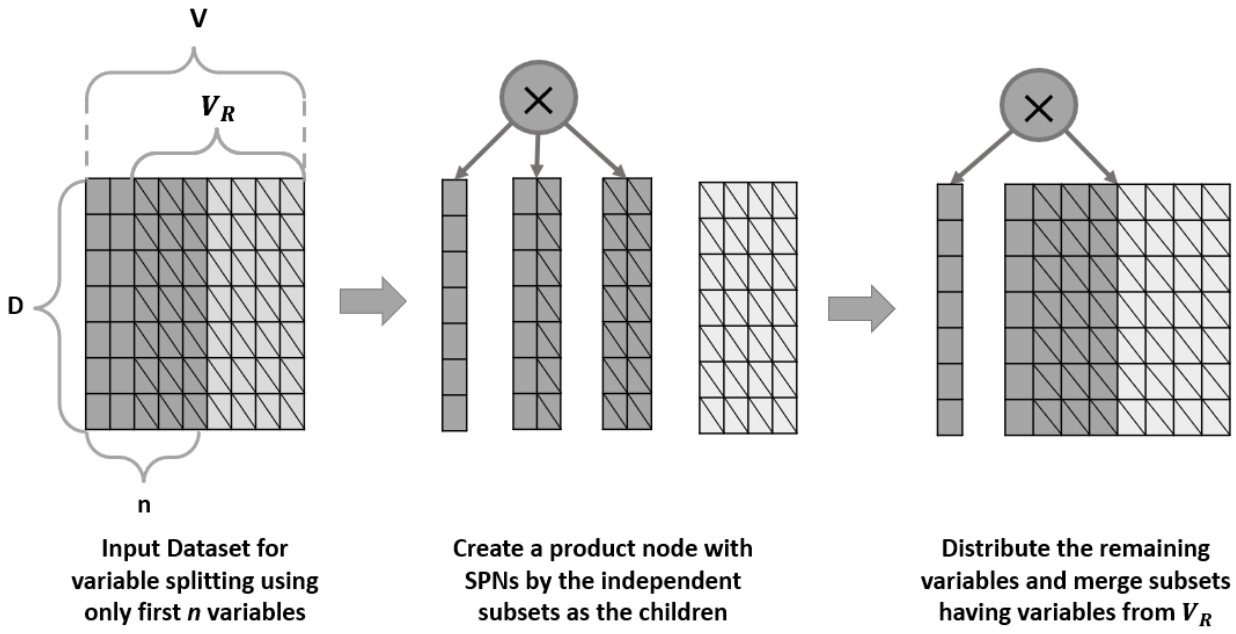


Figure 5.4: Illustration of the variable splitting operation for the LearnSPMN* algorithm

5.3.3 Independence Testing for LearnSPMN*

If the current information set $P^{\prec}[i]$ doesn't consist of a decision variable, then variable splitting operation is performed for creating a possible product node. As explained previously for the LearnSPN* algorithm in the Section 4.2.1, the parameter n is similarly used for the LearnSPMN* algorithm as well to reduce the computation time required for independence testing.

The variable splitting operation for the LearnSPMN algorithm also uses the first n variables from the scope V for independence testing. This set of n variables, denoted as $V[:n]$, are partition into the independent subsets $V_j \in S$. The remaining $|V| - n$ scope variables, $V[n:]$ are distributed among the subsets S in such a way that none of the variables $V[n:]$ that are also in the set V_R are assigned to the subsets that are disjoint from V_R. In other words, if a variable in $V[n:]$ is present in V_R, it is only assigned to the subsets V_j having variables from V_R, that is $V_j \leftarrow V_j \cup \{v\}, v \in V[n:]$, if $v \in V_R$ and $V_j \cap V_R \neq \phi$. After distributing the variables in V_R, all the subsets V_j having variables that are present in V_R are merged together to form a single subset. Now if there are more than one subset that are still

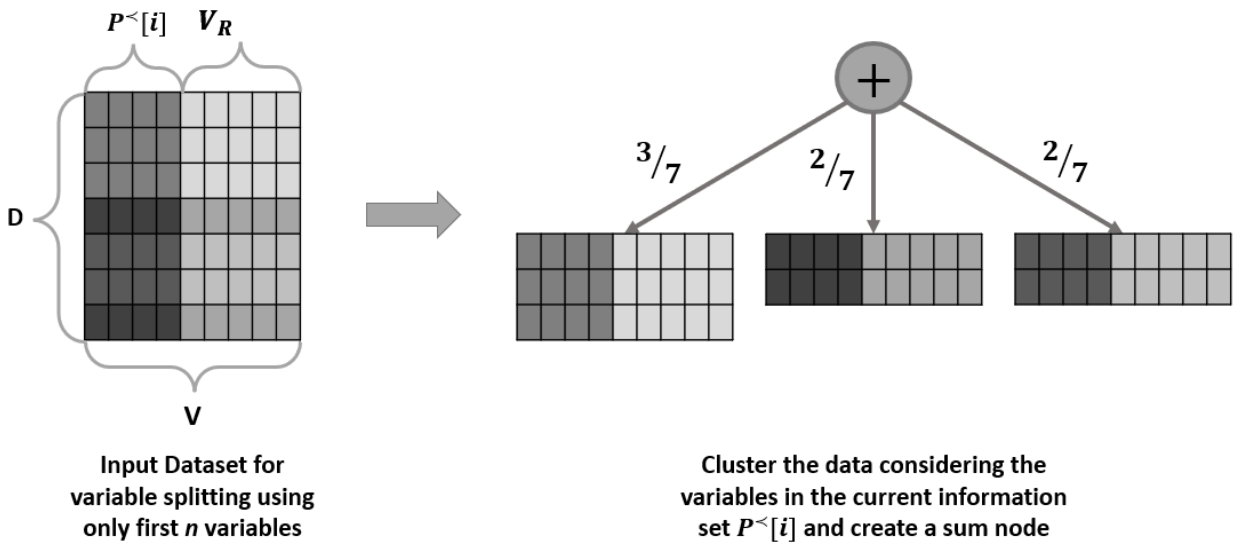


Figure 5.5: Illustration of the clustering operation for the LearnSPMN* algorithm when $k = 3$

left, a new *product* node is created having the SPMN structures that are learned from these subsets as its children. An overview of this operation is presented in the Figure 5.4.

Since only n variables are used for the independence testing rather than all the $|V|$ variables, the computation time for the product nodes is reduced. Also, as variables from the next information sets also might be used for the splitting, any correlations between the variables in $P^{\prec}[i]$ and V_R are preserved. Additionally, having all the variables from V_R in the same subset helps in respecting the partial order.

5.3.4 Clustering for LearnSPMN*

In the case when only one independent subset is found during the variable splitting, the algorithm finally proceeds to discover clusters within the data for creating a sum node. Similar to the clustering operation given in the Section 4.2.2 for the SPNs, the LearnSPMN* algorithm also uses the X-meansWithLimit algorithm for computing the clusters from the data.

As explained previously, the parameter k is used to control the number of clusters and hence the branching at the sum node. The dataset D is partitioned into a maximum of k clusters of similar instances. The only change is that instead of using all the variables present in the dataset for clustering, only the

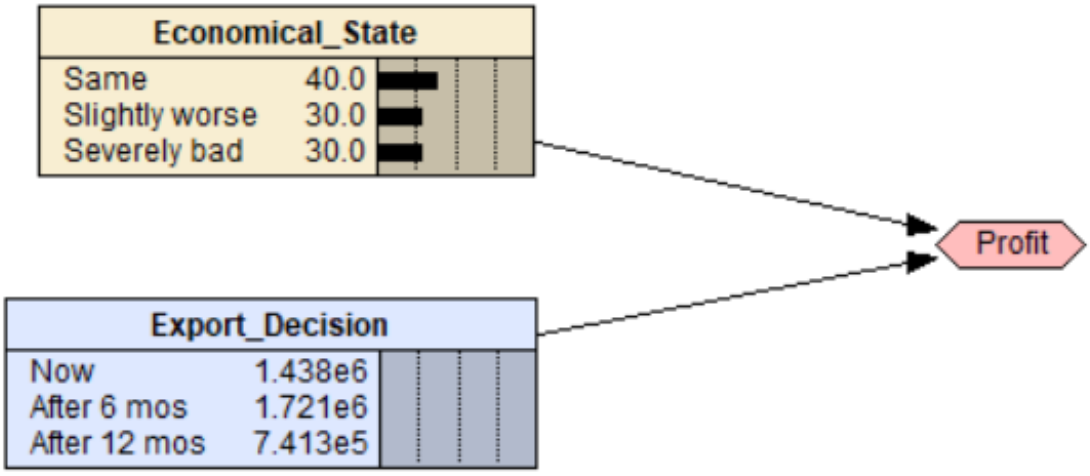
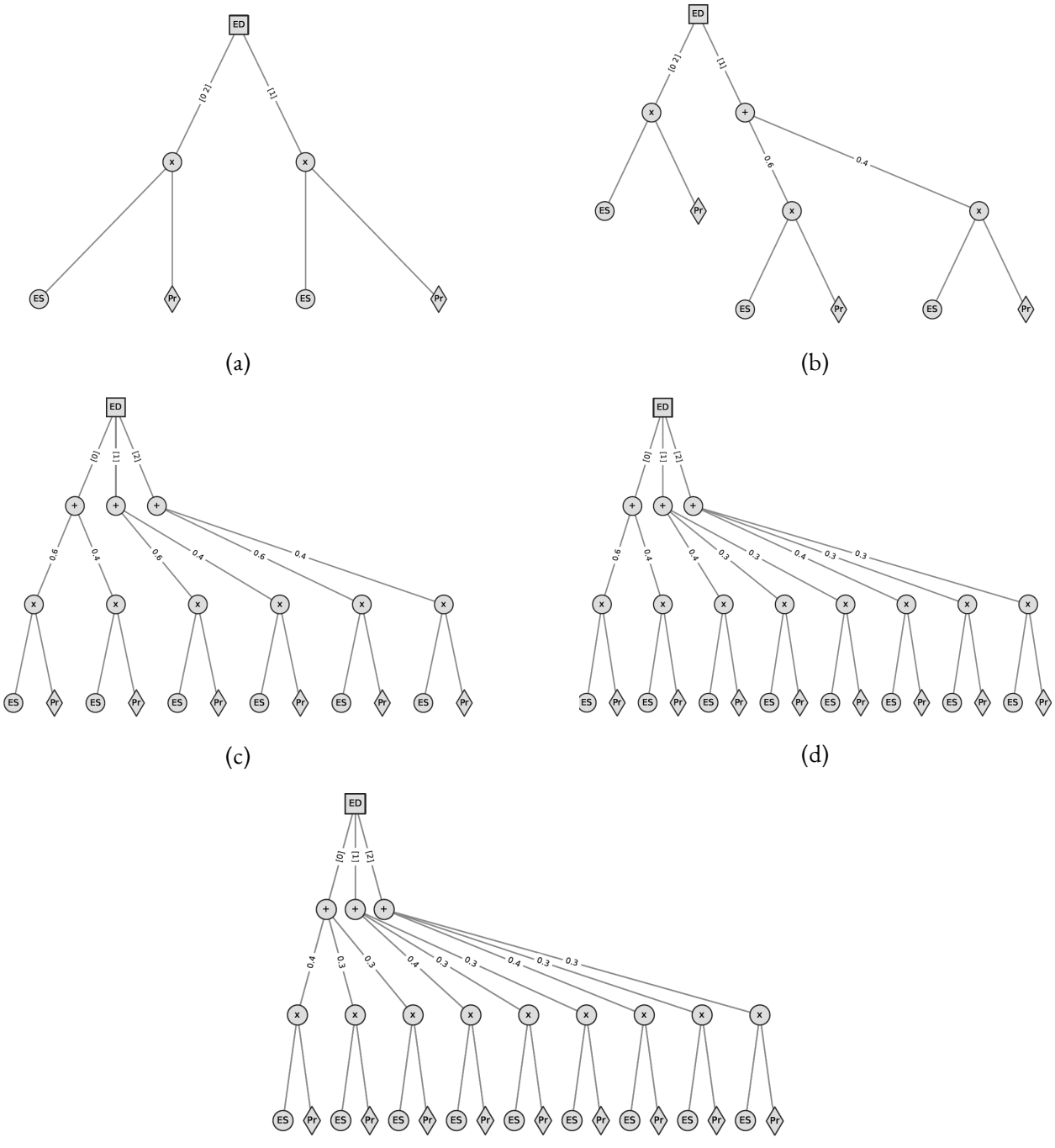


Figure 5.6: Influence Diagram for the Export Textiles domain

variables that are present within the current information set $P^{\prec}[i]$ are used for this operation. This is because the clustering over all the variables might violate the partial order. Then a sum node is returned having the SPMNs learned from the clusters as its children and the relative proportions of the instances in these clusters as their respective edge weights. Thus, the upper limit k on the number of clusters is used to flexibly manage the complexity at the sum nodes by respecting the partial order using this approach. This is illustrated in the Figure 5.5.

5.4 Understanding ANYTIMESPMN with an Example

For understanding the changes in the structure of the network over the course of the ANYTIMESPMN algorithm, consider the *Export Textiles* domain as introduced in (Er & Lezki, 2012). This domain consists of one random variable '*Economical State*' (ES), a decision variable '*Export Decision*' (ED) and a utility value '*Profit*' (Pr). The task here is to select the decision ED that maximizes the utility profit Pr. The Figure 5.6 displays the influence diagram for this domain. The decision variable ED has three decision values, and the variable ES is a discrete random variable having three possible values.



(a) (b)

(c) (d)

(e)

Figure 5.7: Sum Product Max Networks given by the ANYTIMESPMN algorithm for the *Export Textiles* domain. The sub-image (a) is the initial network, and the sub-image (e) is the final network. The sub-images (b), (c) and (d) are the intermediate networks learnt over the course of the ANYTIMESPMN algorithm

The SPMN networks that were returned by the AnytimeSPMN algorithm for the *Export Textiles* domain are shown in the figure 5.7. We can observe that the max node has only two branches with grouped decision values in the first two networks, while the branches are increased to 3 after the second network. The depth in the case of the first network after the decision node is limited to one by naïve factorizing the variables. The intermediate networks are not able to model the underlying distribution accurately since they are approximated by the parameters. As evident from the sum nodes, the final network is able to capture the complete probability distribution as given by the ID. Also as expected, the number of nodes and the accuracy of the models is seen to be increasing over the iterations of the anytime algorithm.

5.5 Validity of the Generated SPMNs

Theorem 5.5.1 *The SPMNs returned at each iteration of the* AnytimeSPMN *algorithm are valid.*

The AnytimeSPMN algorithm makes use of the LearnSPMN* algorithm that requires the parameters k, n, d and d_{max} to learn SPMNs. We need to show that for any combination of the parameters in the given ranges, the SPMN structures hold the properties that ensure validity. So, we prove that the SPMNs at each iteration are valid by using induction from leaves to the root of the network.

Base Case:

Validity is trivial in the case of the leaf nodes since they only represent the univariate distributions or the utility values and are not affect by the parameters of the algorithm. For the base case of validity, consider the case of a SPMN having partial order $P^{\prec} = [[D], [U]]$. This case only has one decision node connected to d utility nodes that are leaves. The evaluation of this SPMN gives $S = max_{v_g} U(v_g)$ which maximizes the utility for the decision groups.

Induction Hypothesis:

Let n^0 be an internal node having children $n^1, ..., n^p$. Let the scope of n^0 be V^0, σ^0 be a policy using decisions in the scope, a state of V^0 be x^0, let the expansion of the sub-network rooted at n^0 be S^0, and the unnormalized probability of x^0 under S^0 be $\Phi^0(x^0, \sigma^0)$. The same notations apply for

the other nodes as well. Thus, by induction hypothesis, the scope of sub-SPN rooted at n^p is $S^p = max_{\sigma^p} \sum_{x^p} \Phi^p(x^p, \sigma^p) U(x^p, \sigma^p)$.

Induction Step:

Thus, if n^0 is a max node for decision variable D^0 having d children corresponding to the decision value groups in v_g, the expansion $S^0 = max_{v_g}\{S^1, ..., S^d\}$. By induction hypothesis, $S^1, ..., S^d$ return the MEUs yielded by sub-SPMNs rooted at nodes $n^1, ..., n^d$ that are learned from data having the same scope V_R. Thus, maximization over these values returns the MEU at the max node and the node n^0 is max-complete. The decision variable D^0 is processed at the node n^0 and $D^0 \notin V_R$. Since the children $n^1, ..., n^d$ have the scope V_R, the decision variable D^0 appears at the most once in the paths from the root to the leaves. Hence, max-uniqueness is also ensured.

If the node n^0 is a sum node, it could have 2 to k number of children. The dataset D is clustered into k clusters by considering the variables in the current information set $P^{\prec}[i]$, but the scope of the clusters remains identical. Thus, sum-completeness is guaranteed irrespective of the value of k. For the sub-SPMN rooted at sum node n^0 with k children, the expansion is $S^0 = w_{01}S^1 + ... + w_{0k}S^k$.

$$\therefore S^0 = w_{01}(max_{\sigma^1} \sum_{x^1} \Phi^1(x^1, \sigma^1) U(x^1, \sigma^1)) + ... + w_{0k}(max_{\sigma^k} \sum_{x^k} \Phi^k(x^k, \sigma^k) U(x^k, \sigma^k))$$

If the scopes of all the children are same, $V^0 = V^1 = ... = V^k$, then a one-to-one correspondence is established between the monomials of the expansions S^0 and the states of V^0. Therefore,

$$\therefore S^0 = max_{\sigma^0} \sum_{x^0} [w_{01}\Phi^1(x^0, \sigma^0) + ... + w_{0k}\Phi^k(x^0, \sigma^0)] U(x^0, \sigma^0))$$

This expression too maximizes the expected utility at the sum node n^0 and so the sub-SPMN rooted at this node remains valid.

Now consider the case of n^0 being a product node. Let the variable splitting performed on the subset of first n variables in V^0 find p independent subsets having scopes $V^1, ..., V^p$ where $p \leq n$. Since the subsets are independent of each other, $V^i \cap V^j = \phi$ for all distinct subset pairs V^i and V^j. The subsets having variables in V_R are merged together and the remaining variables are distributed as mentioned. Due to this, only one branch of the product node has decision or utility variables in its scope and the product

node n^0 having children of scopes $V^1, ..., V^q$ is decomposable. Hence, the expansion of V^0 is,

$$\begin{aligned} S^0 &= S^1 \times ... \times S^{q-1} \times (max_{\sigma^q} \sum_{x^q} \Phi^q(x^q, \sigma^q) U(x^q, \sigma^q)) \\ &= \sum_{x^1} \Phi^1(x^1) \times ... \times \sum_{x^{q-1}} \Phi^{q-1}(x^{q-1}) \times (max_{\sigma^q} \sum_{x^q} \Phi^q(x^q, \sigma^q) U(x^q, \sigma^q)) \\ &= max_{\sigma^0} \sum_{x^0} \Phi^0(x^0, \sigma^0) U(x^0, \sigma^0) \end{aligned}$$

This is because $V^i \cap V^j = \phi$ for all distinct subset pairs V^i & V^j and $V^0 = V^1 \cup ... \cup V^q$. Hence, the policy $\sigma^0 = \sigma^q$ and $\Phi^0(x^0, \sigma^0) = \Phi^1(x^1)...\Phi^{q-1}(x^{q-1})\Phi^q(x^q, \sigma^q)$. So, the sub-SPN rooted at the product node n^0 is valid.

If the depth of the branch from the mentioned point exceeds d_{max}, then the variables V are naïve-factorized. We had already proved validity for naïve factorization in Section 4.4.

Thus, the parameters k, n, d and d_{max} for the flexible structure learning algorithm LearnSPMN* for SPMNs do not interfere with properties required for validity irrespective of the parameter values. Since the network remains sum-complete, decomposable, max-complete and max-unique at each iteration of the AnytimeSPMN algorithm, the algorithm always generates a valid network structure.

5.6 Policy Evaluation

Now that the anytime structure learning algorithm for the modified SPMN structure has been introduced, it is also important to understand how the policies are evaluated for these networks. If the best decision value for a decision variable was to be evaluated in the case of the original SPMN structure, then the expected utility for all the possible decision values given the previous decisions and evidence is computed. Then the decision value that gives the maximum expected utility value is considered for the policy.

This method could also be applied for the modified structures that are returned by the anytime algorithm. But now in this case, all the possible decision values that return the maximum expected utility value are considered for the policy. For a particular decision variable, the agent may select any of the decision values that are available for that variable in the policy at random. Since a group of decision values is

assigned to a branch of a max node, multiple decision values could yield the highest expected utility value. This approach still reduces the inference time as compared to the original structure evaluation since all the structures given by the anytime algorithm except the final structure have smaller network sizes. The final network structure has max node splits similar to the original structure and thus would require the same inference time if the network size is same.

But in the case of the domains having less number of decision values per decision variable, the decision value grouping for all the max nodes corresponding to a particular decision variable would be the same. So, evaluating the expected utilities for all the decision values in the same group would not be necessary since they give the same utility value. Thus, any one decision value from a group could be used to get the expected utility for that group. If this value is the maximum, then all the decision values in that group are considered for the policy.

The second approach provides a good policy only if all the max nodes related to a decision variable have the same grouping. The first approach provides a good policy irrespective of the groupings, but needs more computation. Thus, a hybrid method for policy evaluation is used. The second approach is used for the cases where it would give a good policy, while the first approach is used for the rest of the cases.

Chapter 6

Experiments

In the previous chapters we have introduced two anytime algorithms, AnytimeSPN and AnytimeSPMN, for learning the structures for the sum product networks and the sum product max networks respectively. Now it is important to understand how effective these algorithms are in generating the networks flexibly and whether these algorithms show an increase in the performance of the models as expected for the anytime algorithms. These anytime algorithms for learning the SPNs and the SPMNs have been implemented using the open-source SPFlow library (Molina et al., 2019). The AnytimeSPN algorithm is available in the *anytime_spn* branch of the https://github.com/SwarajPawar/SPFlow.git repository, and the AnytimeSPMN algorithm is provided in the *anytime_spmn* branch.

In this chapter, we describe the experiments that were conducted to evaluate the efficacy of these two algorithms and also analyze the results that were produced by them. In the Section 6.1 of this chapter, we discuss the datasets that were used in the experiments. The sub-section 6.1.1 discusses the testbed for the SPNs, while the sub-section 6.1.2 gives the datasets that were used for SPMNs along with the modifications made to the domains. Then the Section 6.2 presents the results and analysis for the AnytimeSPN algorithm by presenting the performance profiles, computation times and the changes in the network sizes. Similarly, the Section 6.3 analyzes the results given by the AnytimeSPMN algorithm by discussing the improvement in the networks and their policies as observed from their performance profiles and the other related plots.

6.1 Datasets for Evaluation

6.1.1 SPN Evaluation Testbed

For the anytime approach using the AnytimeSPN algorithm for learning the sum product networks, we evaluate the performance of this algorithm on a testbed that is selected from the datasets provided by the *SPFlow* repository. We selected six datasets from the repository for conducting our experiments. These datasets consist of random variables having binary values. The repository has split these datasets into the train, validation and test sets. The instances from the train and test sets are combined into one dataset for the experiments. The number of domain variables and the number of instances for the combined train and test datasets are listed in Table 6.1. All of these datasets are present within the *src/spn/data/binary* folder of the *anytime_spn* branch of the repository.

Table 6.1: Data sets for evaluating AnytimeSPN. $|V|$ indicates the number of random variables in the dataset.

Dataset	$\|V\|$	**# Instances**
NLTCS	16	19417
MSNBC	17	349591
KDDCup 2K	65	215047
Jester	100	13116
Audio	100	18000
Netflix	100	18000

6.1.2 SPMN Evaluation Testbed

The AnytimeSPMN algorithm, introduced for learning the sum product max networks with improving performance, was evaluated over a testbed of datasets that were generated from the RDDLSim domains (Sanner, 2010). The domains provided by the RDDLSim repository are sequential domains. In order to make them suitable for our experiments, these domains were modeled as decision making domains consisting of a finite number of steps. The number of steps for the domains is the number

Table 6.2: Data sets for evaluating ANYTIMESPMN. Here $|X|$ denotes the number of discrete random variables, $|D|$ is the number of decision variables and $|d|$ is the number of decision values per decision variable.

Dataset	$\|X\|$	$\|D\|$	$\|d\|$	Optimal MEU
Elevator	35	6	4	0.5
Navigation	36	5	4	-4.047
Game of Life	36	3	9	10.808
Skill Teaching	72	5	4	-7.181
Crossing Traffic	72	5	4	-4.0

of decisions to be taken. Each instance in the dataset is generated by simulating a random agent in the environment for the given number of time steps $|D|$. The initial state, along with the states observed after performing each random action are recorded. The total reward gained after taking all the $|D|$ decision is recorded as the utility U. The generated sequences respect the partial order $P^{\prec}$, since the variables in the information set I_i represent the state observed after an action D_i at the time step i. These generated datasets are available within the *src/spn/data* folder of the *anytime_spmn* branch.

Since most of the RDDLSim domains model a continuous interaction between the agent and the environment rather than having a finite goal state, a few modifications have been made to these domains for conducting our experiments. In the case of the *Elevators* domain that consists of an instance having three floors, a person may randomly show up at the middle floor with a pre-defined probability and may wish to go to the top floor or the bottom floor. This domain is modified such that the elevator at the first floor is the start state and a person is present on the second floor waiting to go up. Here, the goal for the elevator is to take that person to the third floor which would yield a reward of $+5.0$. The *Navigation* domain originally has a grid of cells of size 3×3. This problem is converted to a domain with a 3×2 grid by simply removing the middle column. The initial state of the *Crossing Traffic* domain, having a 3×3 grid, is modified to have the robot start from the cell that is diagonally opposite to the goal cell. The *Game of Life* and the *Skill Teaching* domains are kept unchanged. All of these domains are available in the https://github.com/SwarajPawar/rddlsim.git repository and having the files named as *domain_name_inst_mdp__2.rddl* in the *files/final_comp/rddl* directory.

6.2 Performance Profiles for AnytimeSPN

The performance profile displays the improvement in the models returned by an anytime algorithm. The log-likelihood of the sum product networks that are learned by the AnytimeSPN algorithm is considered as the quality criterion for the performance profiles. Along with this quality measure, we also record the trend in the size of the networks learned and the computation time required for learning them.

We use a 3-fold cross-validation for evaluating our experimental results. We also compare the evaluation metrics as given by the LearnSPN algorithm from the SPFlow repository with the anytime results. Additionally, we compare the results of the model that was learned without imposing any constraints on the number of clusters and the number of variables for splitting. That is, there is no limit on the number of clusters to be partitioned by the X-means algorithm and all the scope variables are used for independence testing operation during the learning of the model. This model is expected to showcase the true representation of the data and it is considered as the $Upper Limit$ for our results. The anytime approach would eventually yield this model if allowed to continue without any termination criterion.

We use the entire datasets that were formed by combining the train and test sets given in the SPFlow repository for learning the networks models for the *LearnSPN* and the *Upper Limit* baselines. We also use the same data for evaluating these baseline models. The train and test splits that obtained by the 3-fold cross validation split are used for training and evaluating the models yielded by the AnytimeSPN algorithm. This algorithm terminates after reaching convergence when the standard deviation in the log-likelihood of the past three models is less than 10^{-3}.

Also, the *monotonicity* property of the anytime algorithms given by (Zilberstein, 1996) states that the algorithm can return the best model generated so far rather than the last model to maintain a non-decreasing performance profile. To demonstrate the strength of the introduced algorithm, the performance profiles in these results show the quality of the model returned at a particular iteration rather than the best model at that point. The results of our experiments using the AnytimeSPN algorithm are presented in the Figures 6.1, 6.2 and 6.3.

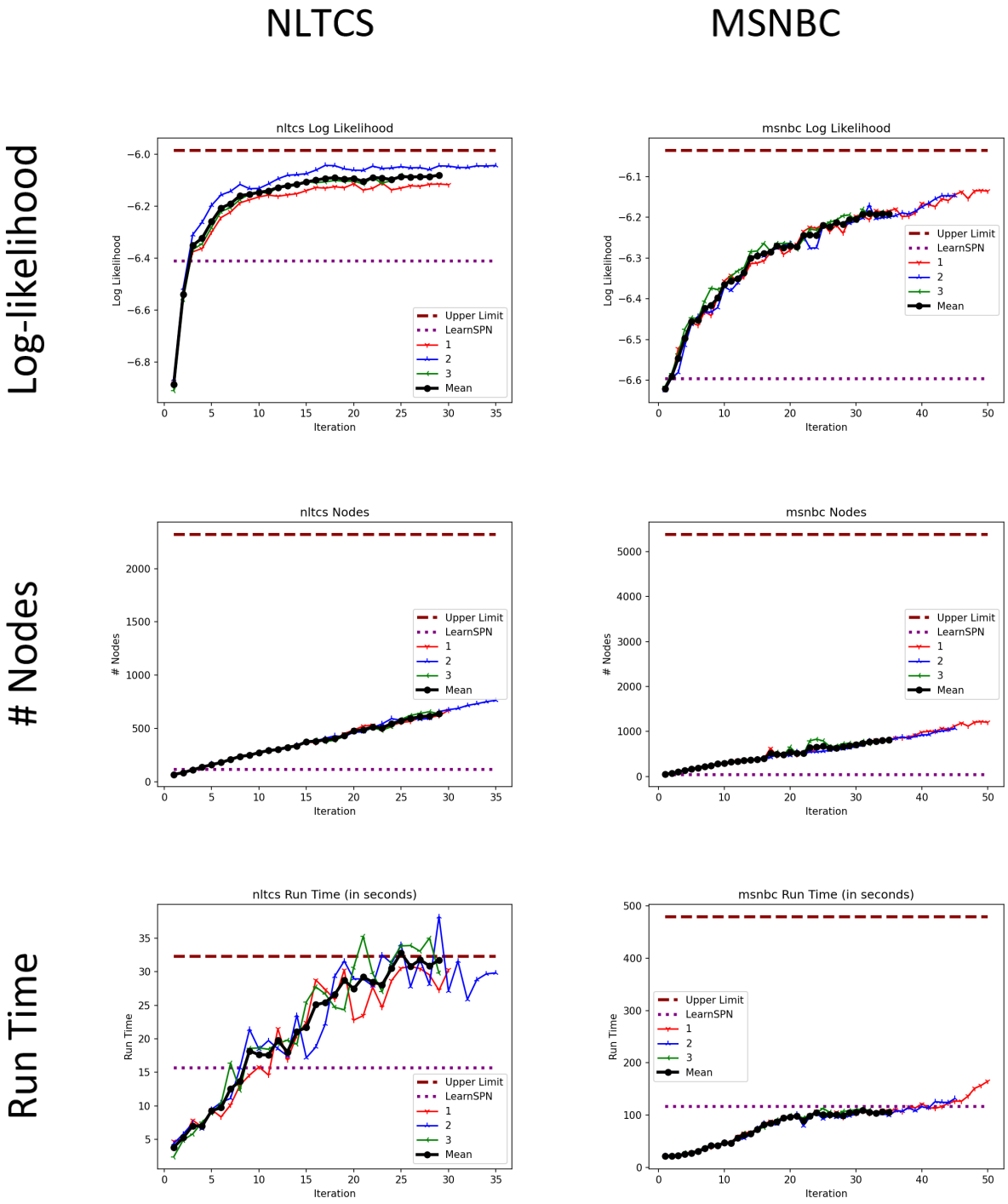


Figure 6.1: Results given by the ANYTIMESPN algorithm for the *NLTCS* and *MSNBC* datasets. The top row illustrates the trend in log-likelihood as the algorithm progresses, the middle row shows the trend in the structure sizes and the last row shows the learning time observed in seconds.

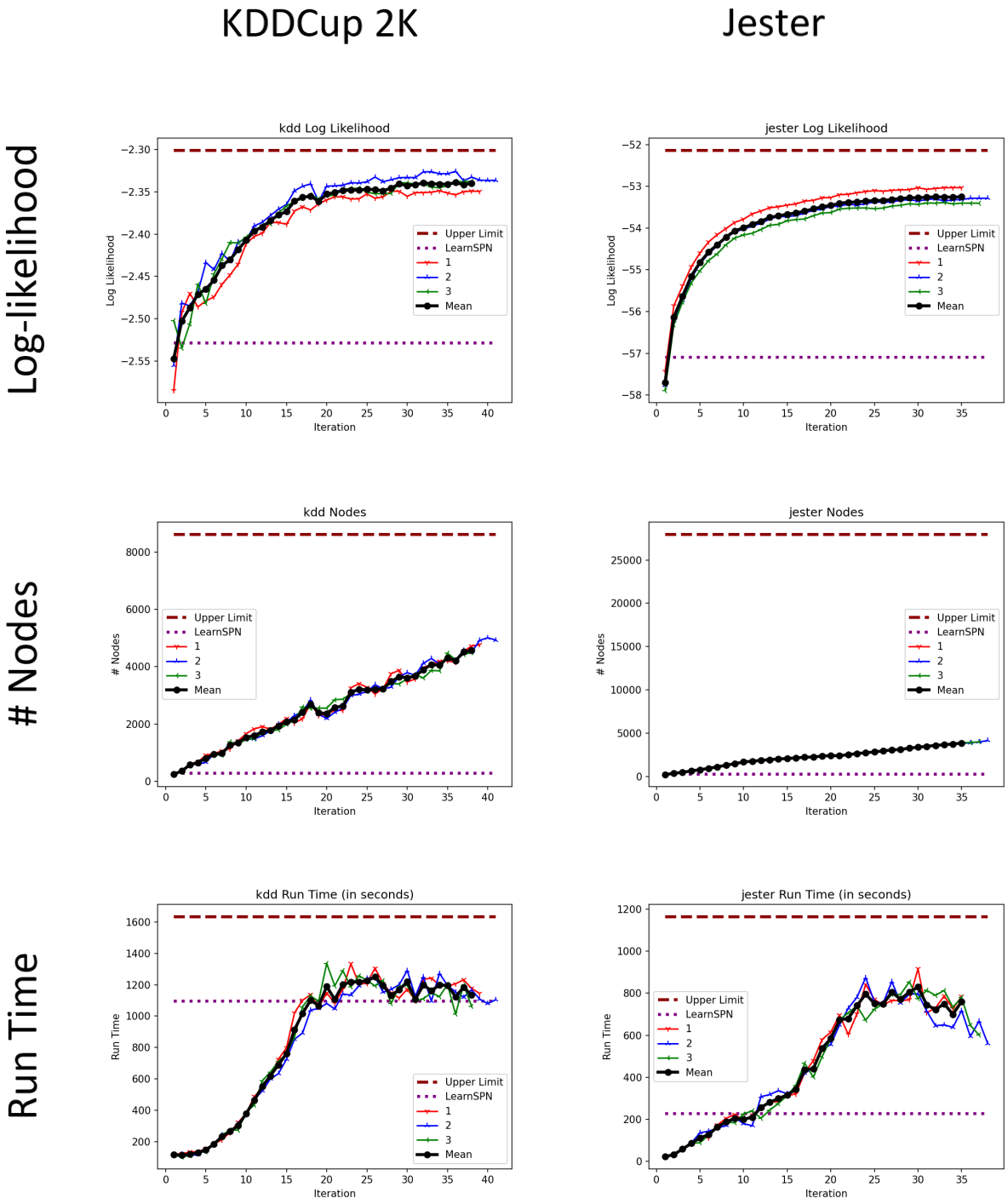


Figure 6.2: Results given by the ANYTIMESPN algorithm for the *KDDCup-2K* and *Jester* datasets. The top row illustrates the trend in log-likelihood as the algorithm progresses, the middle row shows the trend in the structure sizes and the last row shows the learning time observed in seconds.

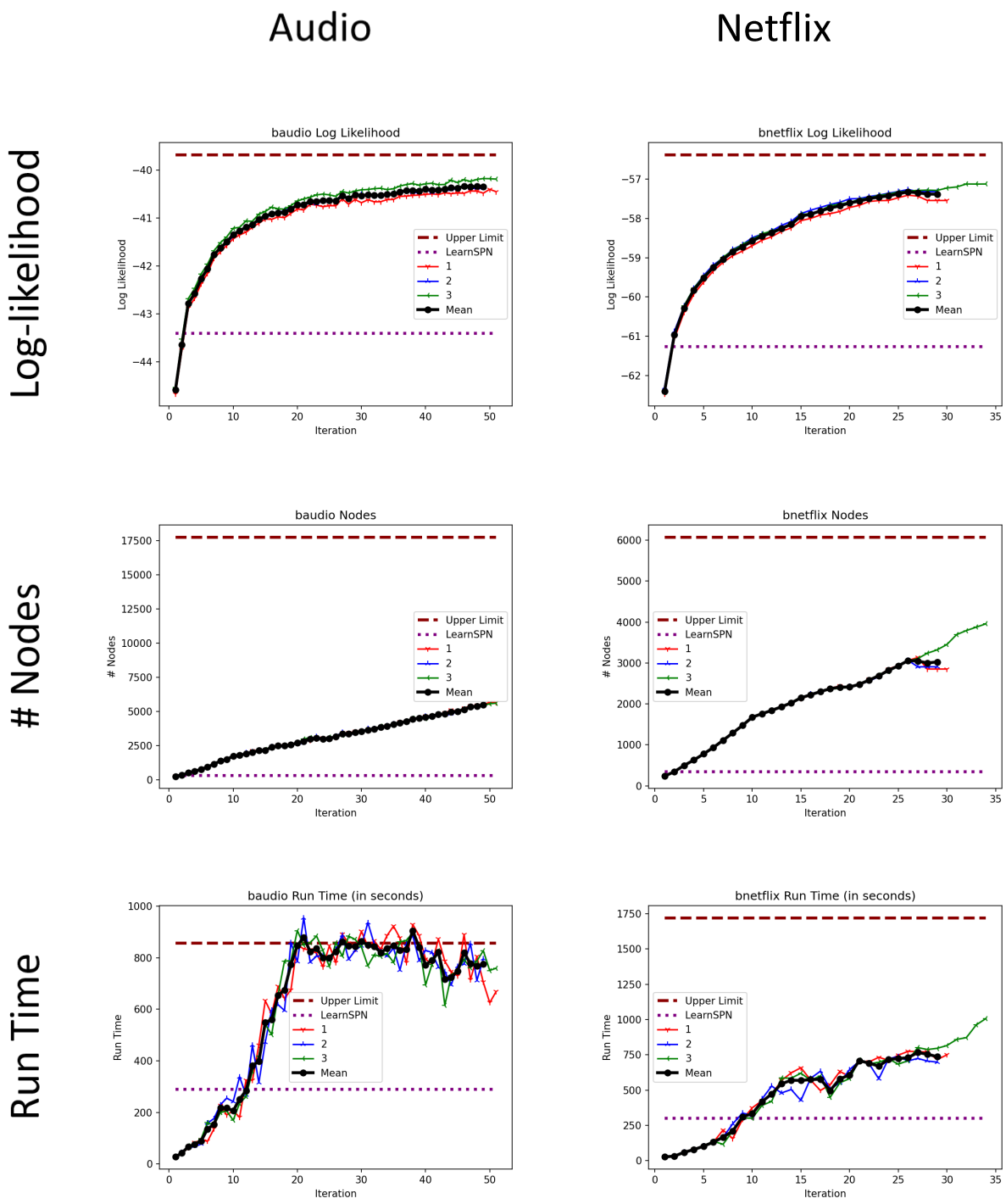


Figure 6.3: Results given by the ANYTIMESPN algorithm for the *Audio* and *Netflix* datasets. The first row illustrates the trend in log-likelihood as the algorithm progresses, the middle row shows the trend in the structure sizes and the last row shows the learning time observed in seconds.

As evident from the experimental results for the AnytimeSPN algorithm that are presented in figures, the log-likelihood curves given by the models for the datasets showcase the performance profile as expected for an anytime technique. As desired, the algorithm is able to demonstrate diminishing results because the improvements in the log-likelihood are greater in the early iterations and they continue reducing over time as it reaches convergence. The algorithm is able to learn the models having nearly non-decreasing log-likelihoods as the computation progresses. Also, we observe a few minor drops in the log-likelihood as we do not return the best generated models at each iteration which could be done to preserve monotonicity. But the algorithm is seen to be able to recover from such drops immediately in the following iterations. Thus, the AnytimeSPN algorithm is clearly able to preserve the characteristics and show the desired properties of an anytime algorithm.

One of the interesting observations inferred from the results is that the anytime algorithm can learn much complex networks having more nodes than the model learned by the LearnSPN algorithm by utilizing much less time. The algorithm is able to learn models that show a better log-likelihood as compared to the *LearnSPN* model in lesser computation times. Hence, the AnytimeSPN algorithm is able to reduce the computation time required by orders of magnitude to provide a performance comparable to the LearnSPN model. The reason for the reduction in the learning time is because only a subset of variables of size n is used for variable splitting instead of using all of them. But the performance of the same model is improved as it able to represent the distribution in a better way because of having a value of k greater than the value 2 that is used for the LearnSPN algorithm.

The AnytimeSPN algorithm for the SPNs is able to achieve a performance level that is comparable to that of the model considered for the *Upper Limit*. But the algorithm is able to achieve such a performance in significantly lesser number of nodes. In most of the datasets, such a performance is observed in less than half of the number of nodes required for the *Upper Limit* network. It is able to reach this performance level in less or equivalent run-times as compared to the upper baseline using smaller network sizes. This quality of the models is observed when the algorithm is closer to the convergence. Apart from this, the results show a steady and linear increase in the network sizes.

6.3 Performance Profiles for AnytimeSPMN

The sum product max networks are used to model the decision making problems. Thus, the quality measure for the performance profile of the AnytimeSPMN algorithm must indicate an estimate of the quality of the policies that are learned. Therefore, we also include the profiles given by averages rewards yielded by the policies of the models along with the log-likelihood that provides an estimate of how well the data distribution is modeled. Additionally, we also plot the changes in the size of the networks and the computation time over the duration of the algorithm till the algorithm terminates after reaching the convergence criterion.

The complete dataset that was generated by the simulating the random agent was used for the training of the models given by the AnytimeSPMN algorithm to evaluate the policies. The average rewards were computed over 25 batches of $20,000$ simulations of the policies that were given by the models. For this evaluations, the mean values along with the standard deviations over the given batches are recorded. Similar to the evaluation of the AnytimeSPN algorithm, we use 3-fold cross-validation for computing the log-likelihood of the models learnt by the AnytimeSPMN algorithm.

We compare the results that are obtained from the AnytimeSPMN algorithm with the results given by the model that is learned using the LearnSPMN algorithm. The evaluations for the *Learn-SPMN* models were performed in the same way as described for the anytime algorithm. Additionally, we also compare the average rewards obtained by the models with the optimal maximum expected utility value for the decision making domains. This MEU value is obtained by simulating the policy given by the value iteration solver that is provided in the RDDLSim code. These optimal values are given in the Table 6.2 and the average rewards given by the anytime algorithm is expected to reach these values towards convergence. We also compare the anytime approach rewards against the average reward that is given by a random policy. The Figures 6.4 and 6.5 illustrates the performance profiles for both the quality measures given by the anytime algorithm for the SPMNs along with the trends in the network sizes and the computation time.

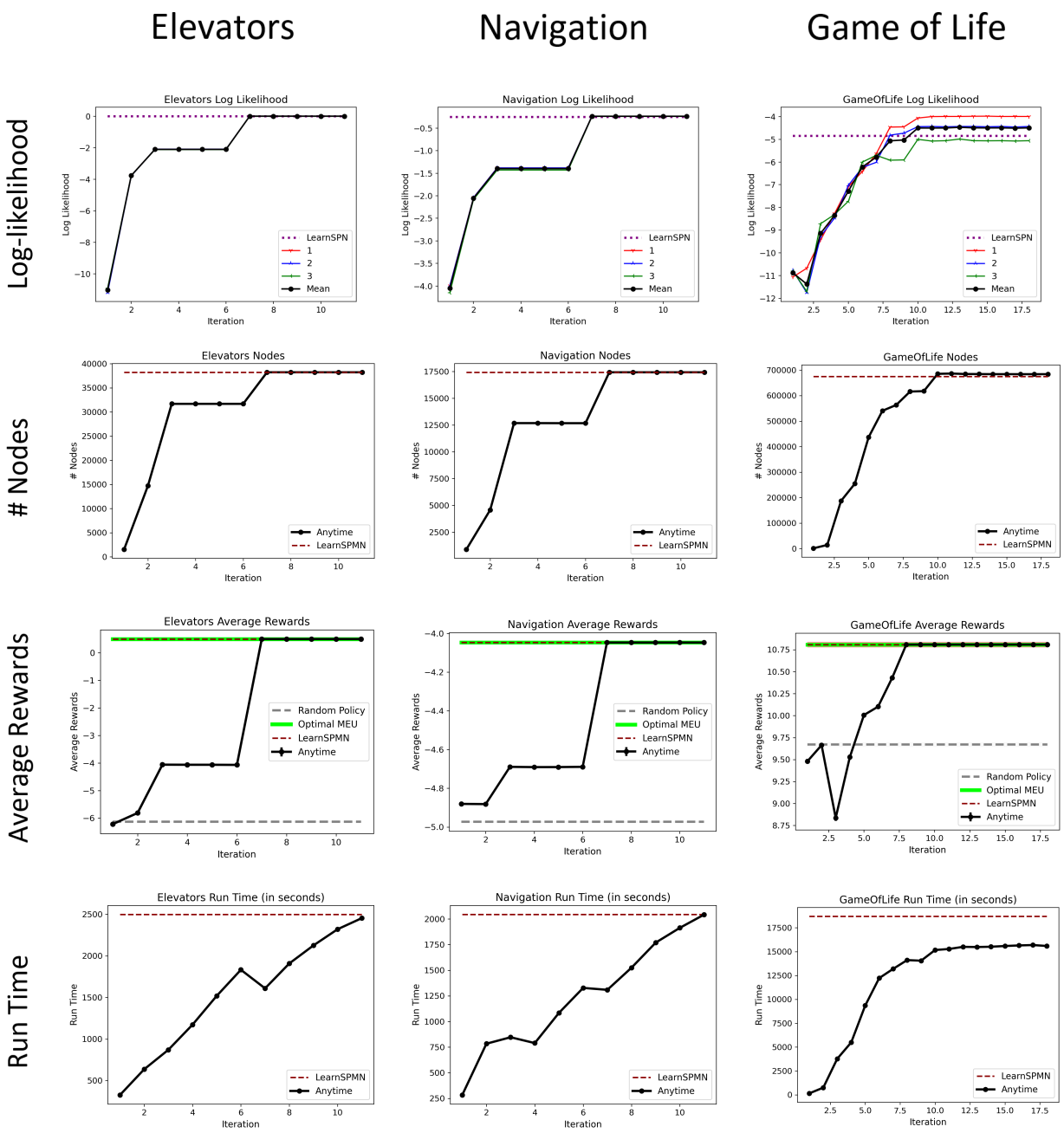


Figure 6.4: Results given by the ANYTIMESPMN algorithm for the *Elevators*, *Navigation* and *Game of Life* datasets. The first row gives the trend in the log-likelihood, the second row shows the number of nodes, the third row illustrates the trend in the average rewards, and the last row shows the learning time in seconds for the anytime approach.

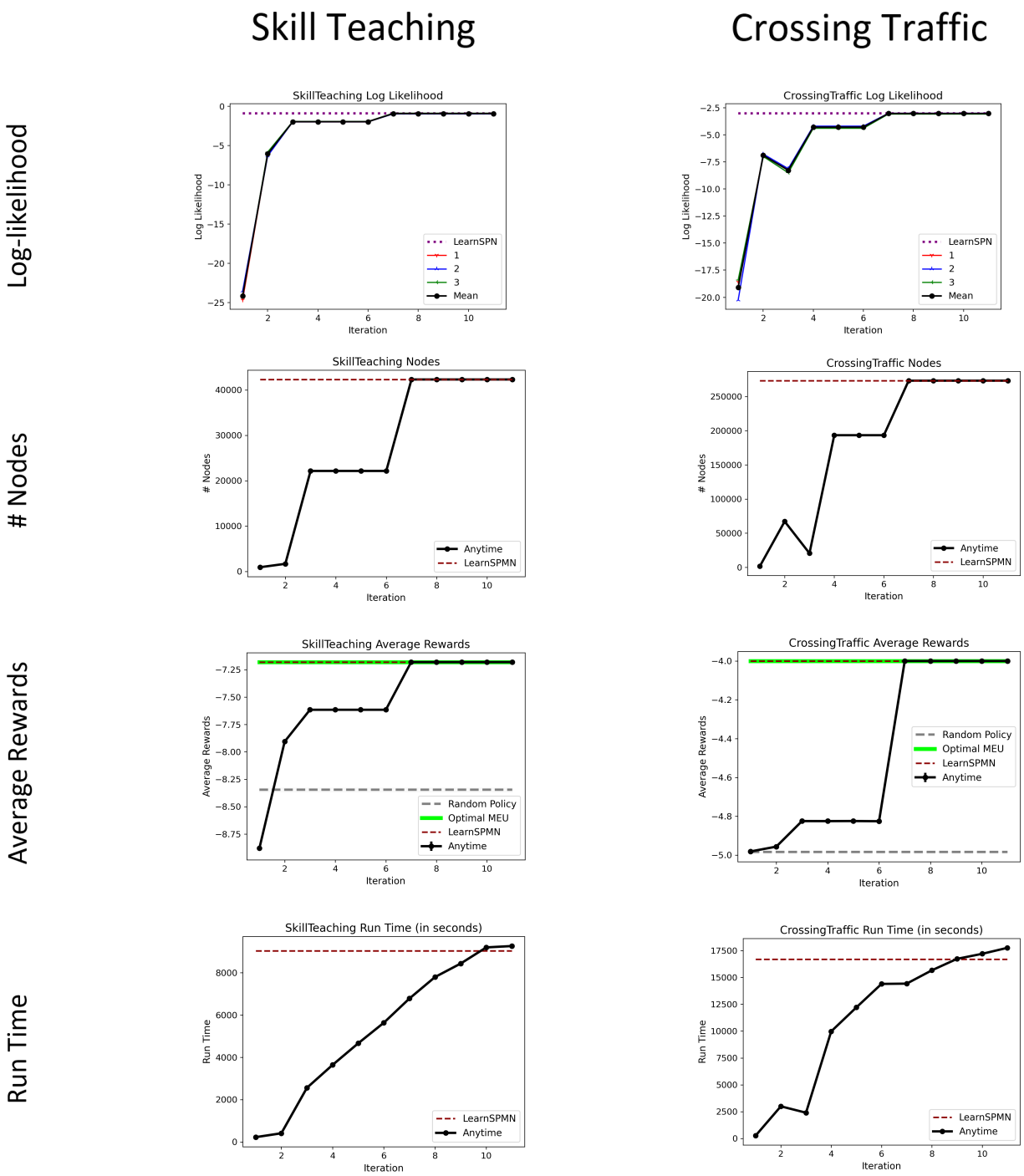


Figure 6.5: Results given by the ANYTIMESPMN algorithm for the *Skill Teaching* and *Crossing Traffic* datasets. The first row gives the trend in the log-likelihood, the second row shows the number of nodes, the third row illustrates the trend in the average rewards, and the last row shows the learning time in seconds for the anytime approach.

As seen from the figures, the performance profiles having log-likelihood of the models as the quality measure for datasets is seen to be increasing as the algorithm proceeds. This again shows the expected behavior of an anytime algorithm. The improvement in the log-likelihood is steeper in the initial iterations. The algorithm is able to observe a non-decreasing log-likelihood in all the cases except the *Crossing Traffic* domain. As mentioned previously, the algorithm returns the last model rather than the best one. This helps us to see that the *Crossing Traffic* domain is able to recover from the drop immediately in the next iteration. Since the size of the networks is in the order of $100,000$, we are not able to directly visualize the change in the network structure during these iterations. But a possible cause for the drop might be the combined increase of the k and d parameters which might have led to a shallower structure with less nodes in this case.

Also, average rewards that are obtained by the AnytimeSPMN algorithm are able to reach the optimal MEU in the later stages of the algorithms. As seen by the improvements in the rewards, the anytime algorithm is able to discover better policies over the time. Thus, the approximation of the policies decreases and their quality improves as expected. Even though a drop was observed in the log-likelihood and the network size of the *Crossing Traffic* domain, the rewards are seen to be improved. The only drop in the rewards is observed in the case of the *Game of Life* domain. The possible reason for this drop is that the best grouping of the decision values might not have been found due to large possibilities of groupings in this case. Also, the anytime algorithm is able to produce networks that perform better than a random policy in most of the iteration except some initial iterations in some of the cases.

The improvement in the log-likelihood and the average rewards is coherent with the increase in the network sizes. Also, the computation time required to learn the models increases over the iteration. An interesting observation is that the computation time required for the models in the earlier phases is less than the models showing similar performance in the later iterations of the algorithm. The anytime algorithm is able to produce optimal models within lesser learning times than the computation time required for the LearnSPMN algorithm.

Chapter 7

Conclusion and Future Work

The Sum Product Networks are appealing since their structures could be learned directly from the data and they provide tractable inference. The Sum Product Max Networks that extend and generalize the SPNs for decision making are interesting as compared to the hand-designed previous models since they are driven by structure learning directly from the data. But the previously introduced structure learning algorithms for SPNs and SPMNs yielded models having no bound over the size or the computation time.

In this work, we introduced the AnytimeSPN algorithm for the Sum Product Networks and the AnytimeSPMN algorithm for the Sum Product Max Networks that are able to learn the structure of these networks with improving performance. These algorithms make use of the LearnSPN* and the LearnSPMN* algorithms respectively that learn the networks structures flexibly by respecting the controlling parameters. These algorithms are successfully able to trade the computation time and the network complexity with the performance or the quality of the models.

We explain in detail the meaning and the role of the parameters used by the algorithms that control the structure of the networks. For both the algorithms, we present a control over the branching of the sum nodes using the parameter k and a reduction in the computation time for variable splitting using the parameter n. Additionally for the AnytimeSPMN algorithm, the parameter d is used to control the branching of the max nodes and the parameter d_{max} is used to control the depth of the branches. Also,

some modifications are introduced for the max nodes in the SPMN structure to control the approximation of the policies.

It is important that every network structure returned by the anytime algorithms is a valid structure. Thus, we also present the validity proofs to show that the networks returned at each iteration of the algorithms satisfy the properties of a valid network. We show that the validity of these networks is not affected by any possible combination of the parameter values.

The testbed of the datasets used for conducting the experiments for evaluating the algorithms is well explained. Some of the modifications that are made to the sequential decision making domains given in the RDDLSim repository for the convenience of our experiments are also stated. We also introduced the baseline models that are used to understand the efficacy of the anytime algorithm results by comparison. The method used for evaluating the models is also mentioned.

The experimental results are visualized and analyzed using the performance profiles. The quality of the SPN models improves over time as estimated by the log-likelihood measure. Similarly the performance of the SPMN is improved over the algorithm duration as given by the log-likelihood and the average rewards. The SPN structures learned by the AnytimeSPN algorithm in lesser computation time perform better than the LearnSPN algorithm and perform comparable to the *Upper Limit* results using significantly less nodes. Similarly, the models given by the AnytimeSPMN are able to learn the optimal policy using less computation time than that required for the LearnSPMN algorithm. These algorithms can be applied to the areas where the computation time is more critical than the quality of the results.

A future direction to this work could be extending the anytime technique for learning the template network structures for the recurrent variants whose size currently remains unbounded. Also, a more sophisticated method for grouping the decision values for the max nodes should be discovered which might help to prevent any drops in the average rewards.

Bibliography

Adel, T., Balduzzi, D., & Ghodsi, A. (2015). Learning the structure of sum-product networks via an svd-based algorithm. *Proceedings of the 31st Conference on Uncertainty in Artificial Intelligence*, 32–41.

Chow, C., & Liu, C. (1968). Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, *14*(3), 462–467. https://doi.org/10.1109/TIT.1968.1054142

Darwiche, A. (2003). A differential approach to inference in Bayesian networks. *Journal of the ACM*, *50*(3), 280–305. https://doi.org/10.1145/765568.765570

Dean, T., & Boddy, M. (1988). An analysis of time-dependent planning. *Proceedings of the 7th National Conference on Artificial Intelligence*, 49–54.

Dennis, A., & Ventura, D. (2012). Learning the architecture of sum-product networks using clustering on variables. *Advances in Neural Information Processing Systems*, *25*.

Dennis, A., & Ventura, D. (2015). Greedy structure search for sum-product networks. *Proceedinds of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, 932–938.

Di Mauro, N., Esposito, F., Ventola, F. G., & Vergari, A. (2017). Alternative variable splitting methods to learn sum-product networks. *AI*IA 2017 Advances in Artificial Intelligence*, 334–346.

Er, F., & Lezki, S. (2012). The usage of influence diagram for decision making in textiles. *Asian Social science*, *8*(11), 163–169. https://doi.org/10.5539/ass.v8n11p163

Ester, M., Kriegel, H.-P., Sander, J., & Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 226–231.

Fukunaga, K., & Hostetler, L. (1975). The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, *21*(1), 32–40. https://doi.org/10.1109/TIT.1975.1055330

Gens, R., & Domingos, P. (2012). Discriminative learning of sum-product networks. *Advances in Neural Information Processing Systems*, 3248–3256.

Gens, R., & Domingos, P. (2013). Learning the structure of sum-product networks. *Proceedings of The 30th International Conference on Machine Learning*, 873–880.

Hayes, L., Doshi, P., Pawar, S., & Tatavarti, H. T. (2021). State-based recurrent spmns for decision-theoretic planning under partial observability [Main Track]. *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, 2526–2533. https://doi.org/10.24963/ijcai.2021/348

Horvitz, E. (1987). Reasoning about beliefs and actions under computational resource constraints. *Proceedings of the Third Conference on Uncertainty in Artificial Intelligence*.

Koller, D., & Friedman, N. (2009).

Liu, Y., & Luo, T. (2019). The optimization of sum product network structure learning. *Journal of Visual Communication and Image Representation*, *60*, 391–397. https://doi.org/10.1016/j.jvcir.2019.02.012

Melibari, M., Poupart, P., & Doshi, P. (2016). Sum-product-max networks for tractable decision making. *Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI)*, 1846–1852.

Molina, A., Vergari, A., Stelzner, K., Peharz, R., Subramani, P., Mauro, N. D., Poupart, P., & Kersting, K. (2019). Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks. *arXiv preprint arXiv:1901.03704*.

Peharz, R., Geiger, B. C., & Pernkopf, F. (2013). Greedy part-wise learning of sum-product networks. *Machine Learning and Knowledge Discovery in Databases*, 612–627.

Pelleg, D., & Moore, A. (2000). X-means: Extending k-means with efficient estimation of the number of clusters. *Proceedings of the Seventeenth International Conference on Machine Learning*, 727–734.

Poon, H., & Domingos, P. (2011). Sum-product networks: A new deep architecture. *12th Conf. on Uncertainty in Artificial Intelligence (UAI)*, 2551–2558.

Rooshenas, A., & Lowd, D. (2014). Learning sum-product networks with direct and indirect variable interactions. *Proceedings of the 31st International Conference on Machine Learning, ICML*, *32*(1), 710–718.

Sanner, S. (2010). *Relational dynamic influence diagram language (rddl): Language description*.

Schwarz, G. (1978). Estimating the dimension of a model. *The Annals of Statistics*, *6*(2), 461–464. https://doi.org/10.1214/aos/1176344136

Tatavarti, H. T. (2020). *Recurrent sum-product-max networks for decision making in perfectly-observed environments*.

Tatavarti, H. T., Doshi, P., & Hayes, L. (2021). Recurrent sum-product-max networks for decision making in perfectly-observed environments. *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling*, *31*.

Vergari, A., Di Mauro, N., & Esposito, F. (2015). Simplifying, regularizing and strengthening sum-product network structure learning. *Machine Learning and Knowledge Discovery in Databases*, *9285*, 343–358. https://doi.org/10.1007/978-3-319-23525-7_21

Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Mag.*, *17*, 73–83.

Zilberstein, S., & Russell, S. (1995). Approximate reasoning using anytime algorithms. *318*. https://doi.org/10.1007/978-0-585-26870-5_4