A Benchmark for Maximum-A-Posteriori Inference Algorithms in Discrete Sum-Product Networks

Heitor Reis Ribeiro

Dissertation presented
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
to obtain the title
of
Master in Computer Science

Programa: Mestrado em Ciência da Computação

Orientador: Prof. Dr. Denis Deratani Mauá

During the development of this work the author received financial aid from CAPES

São Paulo, February, 2021

A Benchmark for Maximum-a-Posteriori Inference Algorithms in discrete Sum-Product Networks

This version of the dissertation contains the corrections and alterations suggested by the judging committee during the work's original version's defense that occurred in 21st of May, 2021. A copy of the original version is available at the Instituto de Matemática e Estatística from the University of São Paulo.

Judging committee:

- Prof. Dr. Denis Deratani Mauá - IME-USP
- Prof. Dr. Ricardo Cerri - UFSCAR
- Prof. Dr. Ronaldo Cristiano Prati - UFABC

Resumo

RIBEIRO, H. R. **Um Benchmark para Algoritmos de Inferência de Maximum-a-Posteriori em Redes Soma-Produto discretas**. 2021. 56 f. Tese (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

A solução para problemas de Inferência de Maximum-a-Posteriori em redes de Soma-Produto resultam na configuração mais provável das Variáveis Aleatórias representadas em sua estrutura; um passo importante em raciocínio probabilístico que pode ser usado para muitas aplicações, como preenchimento automático de imagens. Já foi provado que este problema é NP-difícil (até para aproximar) em redes de Soma-Produto. Vários algoritmos já foram desenvolvidos para obter uma solução boa ou exata para esse problema, mas os experimentos realizados até agora foram limitados. Nesta dissertação nós fornecemos descrições, análises, e um benchmark para realizar mais testes experimentais para algoritmos que resolvem esse problema. Nós concluímos que, dada uma janela de tempo limitada, um algoritmo de Busca Local iniciado com uma solução retornada pelo algoritmo Argmax-Product alcança, em média, os melhores resultados nos conjuntos de dados testados.

Palavras-chave: Modelos Probabilísticos, Redes Soma-Produto, Maximum-a-Posteriori.

Abstract

Heitor Reis Ribeiro. **Algorithms for Maximum-a-Posteriori inference in discrete Sum-Product Networks**. 2021. 56 f. Tese (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

The solution to Maximum-a-Posteriori Inference problems in Sum-Product Networks provides the most probable configuration of the Random Variables encoded in its structure; a key step in Probabilistic reasoning that can be used for many applications, such as image auto-completion. It has been proven that this problem is NP-Hard (even to approximate) in Sum-Product Networks. Multiple algorithms have been developed to reach either approximate or exact solutions to this problem, but the experiments have been limited. In this Dissertation we provide descriptions, analysis, and a benchmark for experimental testing for algorithms that solve this problem. We conclude that, given limited time, a Local Search algorithm starting with a solution found by the Argmax-Product algorithm reaches, on average, better results on the tested datasets.

Keywords: Probabilistic Models, Sum Product Networks, Maximum-a-Posteriori.

Contents

List of Abbreviations

AMP	Argmax Product
BS	Beam Search
BT	Best Tree
FC	Forward Checking
FV	First Variable
KBT	K-Best Tree
LaM	Largest Margina
LoM	Lowest Marginal
LaE	Largest Entropy
LoE	Lowest Entropy
LS	Local Search
MAP	Maximum-a-Posteriori
MC	Marginal Checking
MP	Max Product
MRF	Markov Random Field
MS	Max Search
Ord	Ordering
PGM	Probabilistic Graphical Model
RV	Random Variable
SPN	Sum-Product Network

List of Symbols

$\mathcal{S}$	Sum-Product Network
α	Performance guarantee
P	Probability distribution function
X	A random variable
$val(X)$	The support of a random variable X
$\boldsymbol{X}$	A set of random variables
$\mathcal{X}$	Partial assignments for a single random variable
$\boldsymbol{\mathcal{X}}$	Partial assignments for a set of random variables
$\boldsymbol{x}$	Complete assignment for a set of random variables
$sc(\mathcal{S})$	Scope of SPN $\mathcal{S}$

List of Figures

List of Tables

List of Algorithms

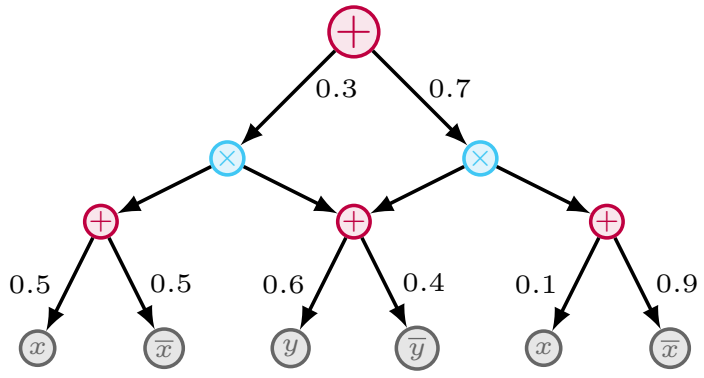
Chapter 1

Introduction

1.1 What are Sum-Product Networks?

Sum-Product Networks (SPNs) are probabilistic models introduced in [PD11] that represent a joint probability distribution over a set of random variables as a rooted, directed graph of arithmetic operations (Sums and Products). Sum operations are represented with sum nodes (⊕), and they are connected to other nodes with a weighted arc. Product operations are represented with product nodes (⊗) and they are connected to other nodes with unweighted arcs. The leaves in these graphs are indicator nodes (◎) and they represent one particular assignment to a random variable. SPNs have been used in problems such as activity recognition [AT15], language modeling [CKP+14], and image classification [SC16]. Figure 1.1 shows an example of a small SPN over two variables, X and Y, both binary, where x represents an indicator of $X = 1$, and $\overline{x}$, $X = 0$. The same for variable Y.

Similar to the more known Probabilistic Graphical Models (PGMs), SPNs are used as a way of reasoning under uncertainty, that is, using data with an unknown probabilistic distribution that can be inferred with a large enough data sample, such as a collected or generated dataset, in order to make future decisions (or predictions) over it. For example, given a dataset of census data (age, education, occupation, marital status, income, etc.), a probabilistic graphical model can predict, with measurable uncertainty, the value of (at least) one of the obtained variables (such as income) given the value of some of the other variables – a process called conditional inference. The main advantage of SPNs is the calculation of exact, marginal and conditional inferences in linear time in relation to their size – operations that are NP-Hard on the other PGMs.



x	y	Probability
0	0	0.312
0	1	0.468
1	0	0.088
1	1	0.132

Figure 1.1: *An example SPN over two binary Random Variables X and Y.*

1.2 The Maximum-A-Posteriori Problem

A key component in artificial intelligence problems that use graphical models to reason under uncertainty is finding the maximum probability configuration, that is, an attribution (or assignment) to a set of variables that yields, upon inference, the highest probability value. One can use such an operation to answer questions such as "What is the most probable income for a person older than 35 that lives in a specific city?" or "What is the most probably entire image given half of its pixels?", given enough relevant data to be analyzed. It has been used previously in image segmentation [GG84], 3D image reconstruction [BVZ98], natural language processing [KRC+10], and sentiment analysis [ZNSS11].

The operation of finding this configuration on a PGM is named Maximum-a-Posteriori (MAP) Inference. In SPNs with minimum height of 3, this type of inference is NP-Hard [CMdC17] even for finding an approximation of the value obtained by the configuration. That is, under the standard assumption of complexity theory that $P \neq NP$, one cannot efficiently find such a configuration for an arbitrary SPN.[1]

There have been few studies that examine this problem. Besides the previously mentioned article, where the problem's complexity was proven, the MAP Inference problem has been described in more detail in [Peh15]. Up until then, we had one algorithm for this problem, the Max-Product (first described in [PD11]), which is fast (it takes linear time on the size of the SPN), but has no guarantee about the solution it finds. In [CMdC17] they introduced the ArgMax-Product algorithm, an extension on Max-Product that takes quadratic time on the size of the SPN, and its results are never worse than the results found by Max-Product.

Then in [MJT18] the authors proposed and evaluated two new approximate algorithms: Beam Search, an adaptation of hill-climbing to SPNs that uses derivatives to find new solutions, and K-Best Tree, an extension of Max-Product that finds K solutions for each node and tries to evaluate the best one. They also introduced the first (and, for now, only) exact algorithm: Max Search, which is slow but can use different heuristics to achieve a faster running time, and can be stopped at anytime to obtain a solution that improves with more time given. However, their evaluations had limitations: They were restricted to binary SPNs, and they only checked running time and in how many runs the algorithms reached the highest value within the allocated time. They did not test the quality of the found solutions, and had only a couple of heuristics for Max Search.[2]

Recently, [MRKA20] proposed a new approximate algorithm, the Hybrid-Product Belief Propagation, and a new exact algorithm based on a MILP reformulation of the MAP problem, but they performed similar or worse than previous algorithms. An up-to-date survey about SPNs was also recently given in [PSCD20], with some discussion about the MAP problem, but no new algorithms were developed.

1.3 Proposed Setup and Outcomes

One approach of dealing with the MAP problem is using the approximate algorithms, since they find a result that is "good enough": not a measurable approximation, since such a thing would also be NP-Hard, but that is usually[3] not far from the optimal solution. These algorithms may even find the optimal solution in some cases, but are not reliable enough to do so consistently. Still, their running-time is considerably lower than the algorithms that find the optimal solution (both the trivial greedy-search and Max Search), and thus they may have utility.

[1] There are some very specific circumstances that allow for a efficient way of finding this configuration. For example, if part of the solution is already known, and the unknown part has enough value restrictions for the solution space to be small, we can obtain an instance of this problem that can be efficiently computed.

[2] Their proposed exact algorithm, Max Search, works for all (valid) discrete SPNs, but the version they implemented (with the chosen heuristics) was specifically developed for binary ones.

[3] This "usually" shall be shown on the tests, and even then, the difference between the optimal solution and their approximate solution may be large enough for some of these algorithms to not be desirable.

In this dissertation we examine, experiment, and provide empirical results for algorithms designed to find a solution to the Maximum-a-Posteriori inference problem in Sum-Product Networks (SPNs) over discrete random variables. We created a benchmark of MAP Inference problems for SPNs with a sizeable collection of representative instances, and evaluated the performance of existing algorithms in terms of speed and quality of the found solution, under different metrics. In particular, we measure the solutions found by approximate algorithms and compare them against the fastest algorithm available, Max-Product, both in time and in the values found using many datasets, including non-binary ones.

We also test different applications of these algorithms, such as applying the found solutions to datasets with missing instances, testing classification models on auto-completed instances, and testing different heuristic choices for the Max Search algorithm that were not included in the algorithm's original article.

The main questions we seek to answer are:

1. Given a limited time-frame, which algorithms perform best in comparison with Max-Product?
2. Which heuristics in Max Search find good solutions faster?
3. Does a larger value for a MAP solution provide any gain in a downstream task (such as classification) that depends on that solution?

Our experiments show that, given a limited time, the approximate algorithms most often outperform the value found by Max Search, and thus they may be preferred for some applications. Also, the Max-Product algorithm is the quickest way of finding an initial solution that can be improved upon given more time allocated; and the ArgMax-Product algorithm introduced in [CMdC17] is a slower way of finding solutions that are as good as or better than the Max-Product's solution.

For the algorithms that need an initial solution to improve upon, such as the Max Search (exact solution) or Local Search (approximate solution), the experiments show that Local Search outperforms Max Search on larger SPNs if given the Max-Product solution to improve upon, and on most other SPNs if given the ArgMax-Product solution.

Our main contributions are:

- An SPN library containing implemented algorithms for learning SPNs from data, and executing the MAP algorithms. It is available at https://gitlab.com/marcheing/spn-rs.
- A comparative study of the running-time and MAP values obtained by the existing algorithms.
- An examination of the heuristics for variable and Upper Bound selection on Max Search.

Chapter 2

Background: introduction to SPNs and MAP inference

2.1 Sum-Product Networks

We assume the reader is familiar with probability distributions and (discrete) random variables. We represent random variables by capitalized letters such as X and Y, a set of random variables by bold capitalzed letters such as $\boldsymbol{X} = \{X_1, X_2, \ldots, X_n\}$, the support of a variable X as $val(X) = \{x_1, x_2, \ldots, x_n\}$, and the set of supports of a set of random variables $\boldsymbol{X}$ as $val(\boldsymbol{X}) = \times_{j=1}^{n} val(X_j)$ (the cartesian product of all the $val(X_j)$ sets). The size of a set $\boldsymbol{X}$ is represented by $|\boldsymbol{X}| \geq 0$. A partial assignment to a single variable X is defined as $\mathcal{X} \subseteq val(X)$, and if there is only one value for the variable X in $\mathcal{X}$, we call it an instantiation of X. A partial assignment to a set of variables $\boldsymbol{X}$ is defined by $\boldsymbol{\mathcal{X}} = \times_{j=1}^{n} \mathcal{X}_j$, that is, $\boldsymbol{\mathcal{X}} \subseteq val(\boldsymbol{X})$. Given the variables $\boldsymbol{X}$, if a partial assignment to this set of variables $\boldsymbol{\mathcal{X}}$ contains a single assignment to every variable $X \in \boldsymbol{X}$ (that is, one value for each variable), we call $\boldsymbol{\mathcal{X}}$ a complete assignment and represent it with a lowercase bold letter $\boldsymbol{x}$. We define the operation of obtaining the values assigned to a variable X on a partial assignment $\boldsymbol{\mathcal{X}}$ with $\boldsymbol{\mathcal{X}}[X]$, the result being a partial assignment $\mathcal{X}$ to a single variable X. We represent the operation of removing the values assigned to a variable X from a partial assignment to a set of variables $\boldsymbol{\mathcal{X}}$ with $\boldsymbol{\mathcal{X}} \setminus \{X\} = \boldsymbol{\mathcal{X}}'$, where $\boldsymbol{\mathcal{X}}'$ is now a partial assignment to a set of variables excluding X. We can combine partial assignments to disjunct sets of variables $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ with the operation $\boldsymbol{\mathcal{X}} \times \boldsymbol{\mathcal{Y}}$.

With these notations, we can define Sum-Product Networks properly.

Definition: Sum-Product Networks (SPNs) [1]

An SPN over a set of random variables $\boldsymbol{X}$ is a rooted, directed acyclic graph (DAG) containing three types of nodes:

- Sum Nodes ⊕.
- Product Nodes ⊗.
- Leaf/Indicator Nodes ◎, each one associated with a variable $X \in \boldsymbol{X}$ and a value $x \in val(X)$.
- Whenever there is an arc from node A to node B, we say that A is a *parent* of B, and B is a *child* of A.
- The set of all child nodes of a node A will be represented by $ch(A)$. The set of all parent nodes of a node B will be represented by $pa(B)$.

[1] We use a different definition from the original one in [PD11] and the simplified one in [GP13] in order to avoid the definition of two characteristics that will be assumed by all SPNs used in this dissertation: Completeness (the scope restriction for the sum nodes) and decomposability (the scope restriction for the product nodes).

With an associated scope function:

Definition: Scope of a Node

- The scope of a leaf/indicator node associated with a Variable X is $\{X\}$.
- The scope of a sum or a product node is the union of the scopes of their child nodes.

And following these restrictions:

- Each sum node is connected to its children by weighted arcs with non-negative weights.
- Each child of a sum node has the same scope as the other children of the same sum node.
- Each product node is connected to its children by unweighted arcs.
- Each child of a product node has a different scope from the other children of the same product node.

This definition does not include any restriction enforcing normalization of the weights on a sum node, but the rest of the SPNs shown in this dissertation will. That is, given a sum node S and its weights $w_1, w_2, \ldots, w_n$, we have $\sum_{i=1}^{n} w_i = 1$. Even if the weights of an SPN are not normalized, they can be normalized in linear time [PTPD15]. An SPN with all weights normalized is called a normalized SPN.

2.1.1 Value of an SPN

We represent an SPN with the symbol $\mathcal{S}$, and use it to compute the value of an SPN given a partial assignment $\boldsymbol{\mathcal{X}}$ as follows:

- The value of an indicator associated with a variable X and value x is 1 if $x \in \boldsymbol{\mathcal{X}}[X]$ and 0 otherwise.
- The value of a sum node is the weighted sum of the value of its child nodes (using the weight parameters on the arc between the sum node and the child node).
- The value of a product node is the product of the value of its child nodes.
- The value of the SPN $\mathcal{S}$ is the value of the node at the root.

We represent the operation of obtaining the value of an SPN $\mathcal{S}$ given a partial assignment $\boldsymbol{\mathcal{X}}$ as $\mathcal{S}(\boldsymbol{\mathcal{X}})$. This value inference takes linear time in the size of the SPN[PD11]. Figure 2.1 contains an example of this inference on an SPN over three variables X, Y, Z with the partial assignment $\boldsymbol{\mathcal{X}} = \{0, 1, 2\}$, showing the original SPN at the top and the value of each node during inference at the bottom.

Given that obtaining the value of an SPN is a function of partial assignments to the variables of the SPN, and, on a normalized SPN, it is always a real number between 0 and 1, SPNs represents a joint probability distributions.

2.1.2 A note on marginalization

The term "marginalization" is used many times in this dissertation with a particular meaning that may confuse the reader given knowledge of its original definition. This section aims at explaining this term's differences of meaning when applied to SPNs and the MAP algorithms.

Suppose we have an SPN $\mathcal{S}$ over three discrete random variables X, Y, and Z. Using $\mathcal{S}$, we can obtain the value of queries over the probability distribution of these three variables, such as $P(X = x_1, Y = y_2, Z = z_2)$, which will be represented by $\mathcal{S}(\{x_1\} \times \{y_2\} \times \{z_2\})$ or, more succintly, with $\mathcal{S}(x_1, y_2, z_2)$.

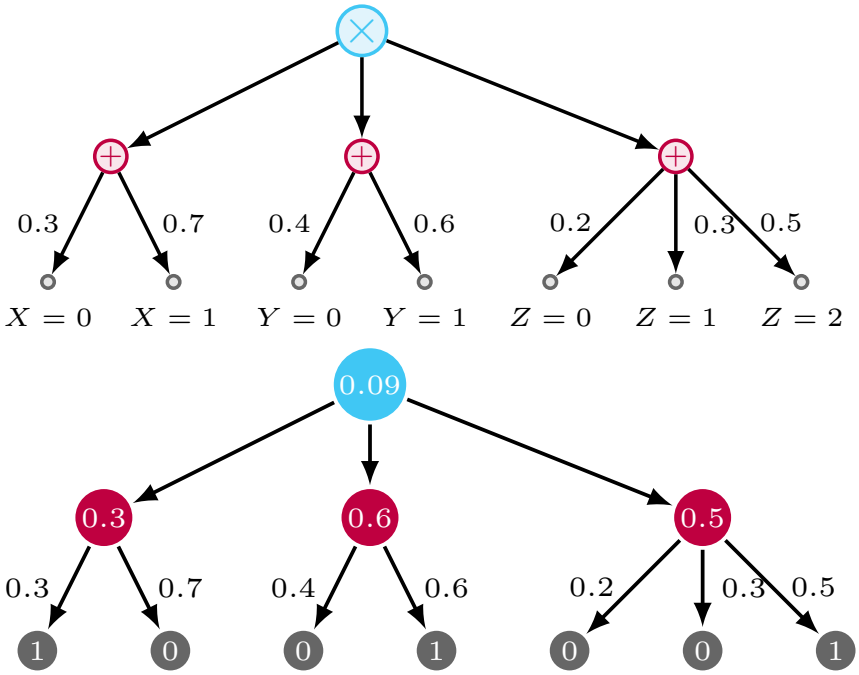


Figure 2.1: *On top, an SPN on three variables. On bottom, the SPN's value inference on the assignment* $X = 0, Y = 1, Z = 2$.

However, there are also queries over the probability distribution without assigning specific values to some variables. For example, $P(X = x_2, Z = z_3)$ ignores the assignment of the variable Y, which means a probability over the event $\{X = x_2, Z = z_3\}$ and considering all possible instantiations of Y. Given the definitions of the probability distribution, this is the same as $\sum_{y \in val(Y)} P(X = x_2, Y = y, Z = z_3)$. This is often referred to as a query with a marginalized variable, and their value is obtained by the sum of other queries.

The previous sum, in SPNs, can be obtained in a single evaluation by setting every indicator for each of the marginalized values to 1. The notation for its value, $\sum_{y \in val(Y)} \mathcal{S}(x_2, y, z_3)$, therefore, can quickly become cumbersome when dealing with more than just one variable marginalized. For example: $\sum_{y \in val(Y)} \sum_{z \in val(Z)} \mathcal{S}(x_2, y, z)$. Because of this we will follow the convention of using $\mathcal{S}(x_2, z_3)$ to mean that, in this query to the SPN $\mathcal{S}$, the value of the variable Y is marginalized.

Additionally, depending on the context, there will be situations in which the "marginalization" will not include all values from a given variable. Consider the partial assignments $\boldsymbol{\mathcal{X}}$ such that there is a value $y_2 \in val(Y)$ and $\{y_2\} \notin \boldsymbol{\mathcal{X}}$. If we marginalize in Y given the set $\boldsymbol{\mathcal{X}}$ and the query $\{X = x_1, Z = z_3\}$, what this actually means is the sum $\sum_{y \in \{y_1, y_3\}} \mathcal{S}(x_1, y, z_3)$.

Therefore, whenever we use the term "marginalization" on a variable X, the context will make it clear whether we are marginalizing on $val(X)$, or on $\boldsymbol{\mathcal{X}}[X]$.

2.1.3 Learning SPNs from data

Simple SPNs, such as the examples in the figures, can be built by hand, but when dealing with datasets of different domains and applications, this will not be the case. In order to obtain the structure and weights of an SPN, we need a learning algorithm.

[GP13] introduced a simple, recursive algorithm for learning the structure of an SPN and the weights of its sum nodes. It is called LearnSPN and works as follows:

The input is a set T of instantiations of variables $V_1, V_2, \ldots, V_n \in \boldsymbol{V}$. The output is an SPN representing a distribution over $\boldsymbol{V}$ learned from T. The most important steps are:

1. Handle the case of a univariate distribution, where $\boldsymbol{V} = \{X_i\}$. In this scenario, the data instances will be used to generate a probability distribution estimation for the variable X_i.

2. Check if the set of variables $\boldsymbol{V}$ can be partitioned into approximately independent subsets, given the values in T.

   - If they can be partitioned, run LearnSPN on each of the partitions $V_1, V_2, \ldots, V_n$ where $\boldsymbol{V} = \bigcup_{i=0}^{n} V_i$ and create a product node as a parent of the n SPNs returned.
   - If they cannot be partitioned into approximately independent subsets, then partition the set T into subsets of similar instances $T_1, T_2, \ldots, T_n$ where $T = \bigcup_{i=0}^{n} T_i$ and create a sum node as a parent of the n SPNs returned, having weight equal to $\frac{|T_i|}{|T|}$ for each T_i.

Algorithm 1: LearnSPN Algorithm

Data: A set of instances T and a set of variables $\boldsymbol{V}$
Result: An SPN $\mathcal{S}$ representing a distribution over $\boldsymbol{V}$ learned from T

```
if |V|= 1 then
    return A sum node having one child for each value in T, with weights equal to an
      estimation of their probability;
else
    if You can partition V into approximately independent subsets {V_1, V_2, ..., V_n} then
        Obtain the partitions {V_1, V_2, ..., V_n};
        return A product node with n children obtained from LearnSPN(T, V_i);
    else
        Partition T into subsets of similar instances T = {T_1, T_2, ..., T_n};
        return A Sum node with n children obtained from LearnSPN(T_i, V) for each T_i with
          weights |T_i| / |T| for each T_i;
    end
end
```

There are two major decisions related to the partitioning of variables and data instances to be made during the implementation of this algorithm: Approximate independence of variables, and instance similarity. The implementation used for the experiments in this dissertation used the Two-Way Likelihood Ratio[2] for the former, and the K-Medoids[3] algorithm for the latter.

For the Univariate distribution estimation (when $|\boldsymbol{V}|= 1$), and since this dissertation will only run these experiments against datasets with discrete variables, we will generate a multinomial distribution with event probabilities obtained from the frequency of such events found in the data instances.

2.2 Maximum-a-Posteriori problem in SPNs

Completing the missing pixels of an image, finding the most probable classification, or simply finding the values that are more probable as assignments to a subset of the variables are all applications of the Maximum-a-Posteriori (MAP) Inference with SPNs, and simply describing them is enough to understand their usefulness, and thus, the usefulness of this type of inference.

Given a set of variables $\boldsymbol{X} = \{X_1, X_2, \ldots, X_n\}$, a Query set of variables $\boldsymbol{Q} \subseteq \boldsymbol{X}$, an Evidence set of variables $\boldsymbol{E} \subseteq \boldsymbol{X}$, and a Marginalized set of variables $\boldsymbol{M} \subseteq \boldsymbol{X}$, with $\boldsymbol{Q}$, $\boldsymbol{E}$, $\boldsymbol{M}$ disjunct, and $\boldsymbol{Q} \cup \boldsymbol{E} \cup \boldsymbol{M} = \boldsymbol{X}$, a complete evidence for the variables in $\boldsymbol{E}$ represented by $\boldsymbol{e}$, the MAP problem in an SPN $\mathcal{S}$ is defined as:

$$MAP(\boldsymbol{Q}, \boldsymbol{e}, \boldsymbol{M}) = \arg\max_{\boldsymbol{q} \in val(\boldsymbol{Q})} \mathcal{S}(\boldsymbol{q} \times \boldsymbol{e} \times val(\boldsymbol{M})) \tag{2.1}$$

The interpretation of this formula is the following:

[2] This is also known as the G-Test of independence, and more information can be found in [Hoe12]
[3] This is also known as the Partitioning around Medoids (PAM) algorithm, described in [GG84]

- There is a set of variables $\boldsymbol{Q}$ and we want to obtain assignments for all of them such that the value of the SPN is maximal. For example, they can be the missing pixels of an image, or the classification variable.
- There is a set of variables $\boldsymbol{E}$ with assignments $\boldsymbol{e}$, and we cannot change their assigned values. They are interpreted as, for example, the given pixels of an image that are not missing, or the entire image for a classification task.
- There is a set of variables $\boldsymbol{M}$, and we cannot change their values, but they are marginalized (either with all the values the variables can assume, or some subset of these values). They can be interpreted, in the previous examples, as the pixels or variables that are not important or can safely be ignored.

With these interpretations, we have that the MAP inference obtains the maximum probability configuration given a certain query, evidence, and marginalized sets. The search space for this problem is the product of the size of the value sets for every variable in $\boldsymbol{Q} = \{Q_1, Q_2, \ldots, Q_j\}$, that is $\prod_{i=1}^{j} |val(Q_i)|$.

The obtained configuration is, therefore, the complete assignment for the variables in $\boldsymbol{Q}$.

The MAP inference is an NP-Hard problem [CMdC17] in the SPNs described so far[4], meaning there are no polynomial-time algorithms that can obtain the exact solution to this problem in valid SPNs if $P \neq NP$.

However, we can still use algorithms that may reach a solution "close enough" to the optimal solution that their found results can still be used on the same applications [5]. This "close enough" measure is named *performance guarantee* and will be denoted with the symbol α.

Consider an SPN $\mathcal{S}$ and its MAP configuration c, where $\mathcal{S}(c) = v$ is the optimal MAP value. Following the definition in [WS11], an approximate algorithm that reaches an α-approximation of the optimal value v will reach a value v' such that $\alpha v \leq v' \leq \alpha$. According to [CMdC17], a provably achievable performance guarantee is $2^{|S|^{\epsilon}}$ for SPNs with height larger than or equal to 3, where $|S|$ is the number of nodes in the SPN, and $0 \leq \epsilon < 1$.

[4] There are different kinds of SPNs, with specific structural restrictions that are learned with different algorithms. In particular, with Selective SPNs, whose learning algorithm is introduced in [PGD14], the MAP problem is not NP-hard since it can be obtained with Max-Product, an algorithm that runs in linear time on the size of the SPN and will be described on chapter 3

[5] It is important to differentiate applications in this step since, for some of them, quickly obtaining a value that is close to the optimal is enough, and in those cases, how much time can be expended in return of a closer approximation becomes the critical trait we will measure.

Chapter 3

Algorithms for MAP Inference in SPNs

There are two main classes of algorithms for the MAP problem in SPNs.

- Exact Algorithms will find the exact solution for the MAP problem.
- Approximate Algorithms will find a complete assignment that may not be the solution to the problem.

The approximate algorithms are generally faster than the exact algorithms and their solution may have use, depending on the problem. For example, image completion with an approximate algorithm can generate an output that looks good enough for the application, and does not need to spend more time in order to find a perfect solution that differs by a small amount of pixels.

Some algorithms are characterized as *Anytime*, meaning that they can be stopped during execution and still return a meaningful solution (in our case, a complete assignment). Giving more time to the algorithm would provide a better solution, even if it is still restricted to an approximate solution.

The given examples in this chapter are run on a small SPN (see figure 3.1) that illustrates the difficulties of reaching the MAP configuration with the simpler algorithms.[1] It is easy to see that the MAP configuration for this SPN is $\{X = x_1, Y = y_1\}$ with the value 0.34.

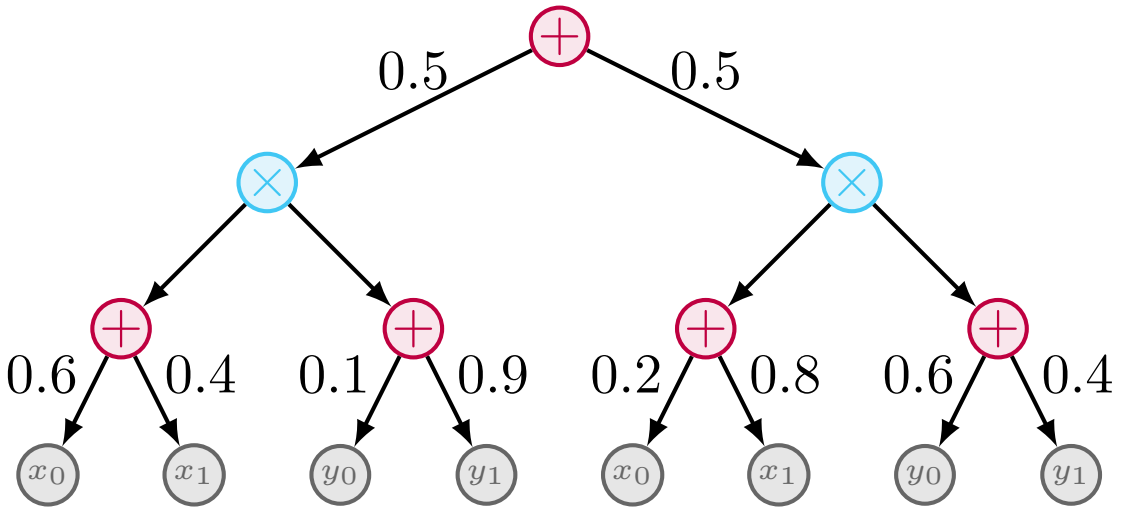


Figure 3.1: *A simple SPN.*

[1] The duplicated Indicators only help with the visualization, since the implementation does not duplicate these nodes.

3.1 Approximate Algorithms

3.1.1 Max-Product or Best Tree

Max-Product was introduced in the same article that presented Sum-Product Networks [PD11]. As noted in [Peh15], this algorithm is not guaranteed to find the MAP configuration on SPNs that are not either selective [PGD14] or augmented [PGPD16].

This algorithm runs in linear time on the size of the SPN (the same complexity as the evaluation of a partial assignment), making it especially useful to efficiently find lower bounds to the MAP value.

Max-Product($\mathcal{S}$) replaces all sum nodes in $\mathcal{S}$ with max nodes in order to perform a bottom-up search that greedily selects the highest output on each of them, returning a configuration.
The major steps of the algorithm are:

1. Change all sum nodes in $\mathcal{S}$ to Max Nodes
2. Perform a bottom-up evaluation on the SPN, propagating the maximum-valued configurations upwards
3. Return the configuration at the root

We now examine these steps in detail.

1. Change all sum nodes in $\mathcal{S}$ to Max Nodes

   A Max Node outputs the highest value of its child nodes, multiplied by the weight of that child node.

   This step is purely symbolical, since the functionality of a sum node can be emulated on the algorithm implementation – that is, without creating a new node type.

2. Perform a bottom-up evaluation on the SPN, propagating the maximum-valued configurations upwards

   A recursive step that makes use of the max nodes and evaluates the SPN in a way of searching for the maximum values of each node. Leaf nodes (indicators) have only one value, and thus it is returned. Product nodes return the product of their children. Max nodes return the maximum value of their children as calculated with the rules described on the previous step.

   When this calculation reaches the root, each node will have an associated value it can output. This value is the Max-Product of each node.

3. Return the configuration at the root

   The Max-Product value at the root is the Max-Product value of the SPN. Since the SPN is valid, this value is the result of the independent selection of indicators, one for each variable, and thus results in a valid complete assignment,[2] which will be the returned configuration.

   The path used to retrieve the variable assignments is one particular induced SPN from the SPN $\mathcal{S}$ [ZPG16], also called a Parse Tree [MJT18]. Since this is the parse tree with the largest value, the algorithm is sometimes called Best Tree.

[2] The restriction for a valid SPN in this step is necessary for the two cases of root nodes: A sum node will have the same scope for each child, which will be the scope of the SPN, and thus will select only one value for each variable. A product node must have disjoint scopes for its child nodes, and thus will merge the Max-Product configurations found on each one; a disjoint union of these configurations will form the resulting Max-Product configuration of the SPN.

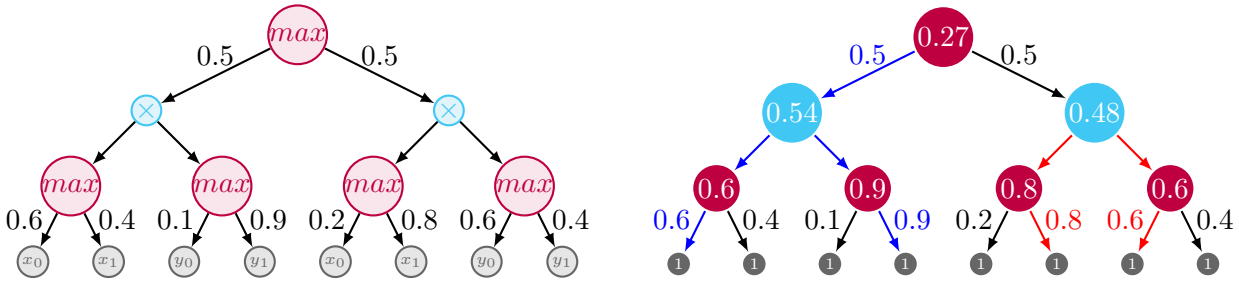


Figure 3.2: *Example of the Max-Product Algorithm. The assignment found at the end is* $\{x_0, y_1\}$ *with value* 0.31 *at the root. Note that the algorithm does not calculate the value of the found configuration.*

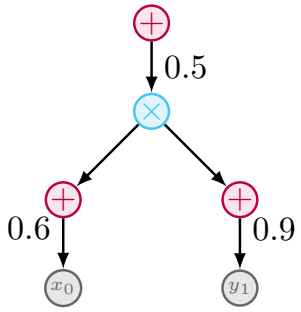


Figure 3.3: *The Parse Tree generated from a run of Max-Product on the example SPN, corresponding to the assignment* $\{x_0, y_1\}$

Max-Product Pseudocode

Algorithm 2: Max-Product

Data: An SPN $\mathcal{S}$
Result: A complete assignment $\boldsymbol{x}$
if $\mathcal{S}$ *is a sum node with weights* $\boldsymbol{w}$ *and child nodes* $\boldsymbol{c}$ **then**
  return *The partial assignment* $\boldsymbol{\mathcal{X}} = argmax_{c_i \in \boldsymbol{c}} w_i \mathcal{S}(\textit{Max-Product}(c_i))$;
else if $\mathcal{S}$ *is a product node with child nodes* $\boldsymbol{c}$ **then**
  return *The union of partial assignments of Max-Product*(c_i) *for* $c_i \in \boldsymbol{c}$;
else
  $\mathcal{S}$ `is an indicator for variable` X_i `with value` x_j;
  return *The partial assignment to variable* X_i, $\{x_j\}$;
end

3.1.2 ArgMax-Product

Argmax-Product, first presented in [CMdC17] computes a complete assignment for the variables of an SPN, and the value obtained by evaluating the SPN on this assignment, guaranteeing an equal or better result than the one found by Max-Product. The difference lies in the checking step that slows the algorithm by calculating the value of a sub-SPN. An example execution of this algorithm is shown in figure 3.4, where the two values that need to be recalculated on sub-SPNs are shown as v_1 vs v_2, where the blue one is chosen instead of the red one.

ArgMax-Product($\mathcal{S}$) Performs a bottom-up search on an SPN $\mathcal{S}$, selecting for the maximum evaluated values of sum nodes, and returning a complete assignment, and the value obtained when this assignment is evaluated.
The major steps of the algorithm are:

1. Perform a bottom-up search of maximum assignments on each node, measured by the value of each node

2. Return the complete assignment and value found at the root

We now examine these steps in detail.

1. Perform a bottom-up search of Maximum configurations on each node, measured by the value of each node

   If the node is a leaf node (indicator) over variable X and value x, we return its maximum value: 1 and the partial assignment is represents: $\{x\}$.

   If the node is a product node, we return the union of the maximizations found on its children, and the product of the values they found.

   If the node is a sum node we test each partial assignment obtained from running Argmax-Product on its children, evaluating them as if the current child node was the root of an SPN. The assignment that, when multiplied by the weight associated with its sub-SPN root, returns the highest value, is stored (along with its value). As an example, a sum node with 3 children will have to test 3 partial assignments on each of the 3 child nodes (9 evaluations of sub-SPNs) in order to check which one reaches the largest value. However, each child node that arrived at an Argmax-Product solution has already been evaluated in order to have that configuration, and thus we have at least one configuration less to check for each child node.

   This step is costly for sum nodes due to the amount of potential assignments to test. Consider the sum node S and its child nodes $ch(S)$. The number of sub-SPNs to evaluate is, at most, $n = |ch(S)|$, each one taking linear time on the size of the sub-SPN rooted at the child node. If child nodes reach equal partial assignments, then this number can be lowered.[3]

2. Return the complete assignment and value found at the root

   As in the Max-Product algorithm, this one will find a parse tree used to extract a complete assignment. However, unlike the previous algorithm, the value of the assignment has already been calculated and will be returned as well.

ArgMax-Product Pseudocode

Algorithm 3: ArgMax-Product

Data: An SPN $\mathcal{S}$
Result: A complete assignment $\boldsymbol{x}$ and the value $\mathcal{S}(\boldsymbol{x})$
if $\mathcal{S}$ *is a sum node with child nodes* $\boldsymbol{c}$ **then**
  return *The configuration* $\boldsymbol{\mathcal{X}}$ *and* $\mathcal{S}(\boldsymbol{\mathcal{X}})$ *such that* $\mathcal{S}(\boldsymbol{\mathcal{X}}) = max_{c_i}\mathcal{S}(\textit{ArgMax-Product}(c_i))$;
else if $\mathcal{S}$ *is a product node with child nodes* $\boldsymbol{c}$ **then**
  return *The union of assignments of ArgMax-Product*(c_i) *for* $c_i \in \boldsymbol{c}$ *and the product of their values*;
else
  $\mathcal{S}$ `is an Indicator for variable` X_i `with value` x_j;
  return *The partial assignment for variable* X_i, $\{x_j\}$ and 1;
end

3.1.3 Local Search

Local Search($\mathcal{S}$, $\boldsymbol{x}$) searches for local optimal complete assignments, starting with an initial complete assignment $\boldsymbol{x}$, by modifying the associated value of one variable at a time, evaluating it in $\mathcal{S}$.

It is also a simple algorithm to run after running one of the other approximate algorithms to improve on the assignment they found. When the task of MAP inference has some pre-allocated time and another algorithm finished before this limit, it could be useful to use the remaining time on an algorithm such as this one to attempt an improvement.
The major steps of the algorithm are:

[3] An implementation improvement that was added to the code used on the experiments.

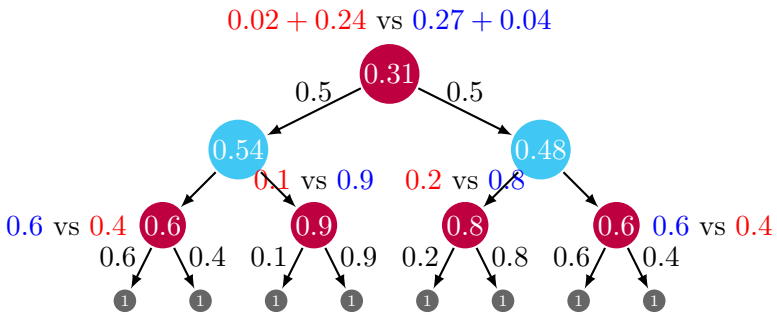


Figure 3.4: *Example of the ArgMax-Product Algorithm. The assignment found at the end is* $\{x_0, y_1\}$ *with value* 0.31 *at the root. The steps where values need to be recalculated on sub-SPNs are marked with* x *vs* y*.*

1. Evaluate the complete assignment $\boldsymbol{x}$ in $\mathcal{S}$ and store its value.
2. Choose a variable X_i whose values $val(X_i)$ have not been evaluated yet. If no such variables exist, jump to step 4.
3. For each $x_i' \in val(X_i)$, Evaluate the complete assignment $\boldsymbol{x}'$ where $\boldsymbol{x}' = (\boldsymbol{x} \setminus \{X_i\}) \times \{x_i'\}$. Store the complete assignment that reaches the highest value.
4. Return the complete assignment of highest value.

We now examine these steps in detail.

1. Evaluate the complete assignment $\boldsymbol{x}$ in $\mathcal{S}$ and store its value.

   The complete assignment $\boldsymbol{x}$ can be obtained in many ways, such as a random selection, or another approximate algorithm. The value $\mathcal{S}(\boldsymbol{x})$ is the starting point of the search and, as expected from an algorithm whose goal is to find the local maximum, will heavily influence the last solution found.

2. Choose a variable X_i whose values $val(X_i)$ have not been evaluated yet. If no such variables exist, jump to step 4.

   There are many heuristic choices to use when selecting the variable X_i. However, due to their similarity to the heuristics used in the faster implementations of the Max-Search algorithm, and how much more detailed they are in their implementation when first introduced [MJT18], we will postpone further discussions on these heuristics to their own section.

3. For each $x_i' \in val(X_i)$ Evaluate the complete assignment $\boldsymbol{x}'$ where $\boldsymbol{x}' = (\boldsymbol{x} \setminus \{X_i\}) \times \{x_i'\}$. Store the complete assignment that reaches the highest value.

   Each variable X_i chosen will have the possible values on the set of $val(X_i)$. The current best complete assignment (stored) will have the assignment to the variable X_i changed to one of the values in $val(X_i)$, their values recomputed, and compared with the previous one.

   Each time this step is executed, $\mathcal{S}$ will be evaluated (at most) $|val(X_i)|$ times, which results, if no optimizations are considered, in the total running time of $T(\mathcal{S}) \sum_{i=1}^{n} |val(X_i)|$, where $T(\mathcal{S})$ is the time to evaluate the SPN $\mathcal{S}$ and n is the number of variables.

4. Return the complete assignment of highest value.

   The complete assignment with largest value stored will be returned, and if there is an use for it, the value obtained by evaluating this configuration has already been obtained on a previous step of the algorithm and can be returned as well.

Local Search Pseudocode

Algorithm 4: Local Search

Data: An SPN $\mathcal{S}$, a set of Query Variables Q, an initial complete assignment of the variables in $sc(\mathcal{S})$, $\boldsymbol{x}$
Result: A complete assignment $\boldsymbol{e}$

```
best ← S(x);
e ← x;
foreach Variable X ∈ Q do
    foreach value x_i ∈ val(X) do
        e' ← the complete assignment e with the assignment to X changed to {x_i};
        if S(e') > best then
            best ← S(e');
            e ← e'
        end
    end
end
return e;
```

3.1.4 Beam Search

Beam search, as an approximate algorithm for MAP, was first presented in [MJT18] and described as an extension of the Hill Climbing algorithm for MAP inference in Bayesian Networks from [Par02].

[Dar03] introduced arithmetic circuits that encoded the network polynomial of a Bayesian Network, and proposed differential methods of obtaining posterior marginal probabilities, a result that was used in [MJT18] to develop an algorithm to obtain the derivatives of an SPN. Since SPNs were built on the same ideas [PD11], the extension of this algorithm to valid SPNs was straightforward.

This algorithm allowed the development of the Beam Search method, and also enabled the development of some heuristics for the Max-Search algorithm, presented in the same article.

Given a partial assignment $\boldsymbol{\mathcal{X}}$ over the variables in $\boldsymbol{X}$, and an SPN $\mathcal{S}$, we can choose one variable $X \in \boldsymbol{X}$ and a value $x \in val(X)$ such that $\{x\} \in \boldsymbol{\mathcal{X}}$ and calculate the following derivative:

$$\frac{\delta \mathcal{S}}{\delta x} = \mathcal{S}((\boldsymbol{\mathcal{X}} \setminus \{X\}) \times \{x\})$$

That is, the partial derivative of variable X on a value $x \in val(X)$ over an SPN $\mathcal{S}$ with a current set of partial evidences $\boldsymbol{\mathcal{X}}$ will calculate the value of the SPN on $\boldsymbol{\mathcal{X}}$ with the assignment of X changed to $\{x\}$. The advantage of this method is that this choice of $\{x\}$ can be made for all variables $X_i \in \boldsymbol{X}$, and all assignments to X_i, at the same time, with the time complexity of an evaluation on the structure of the SPN [MJT18].

The beam search algorithm needs a couple extra parameters: b, called the *beam size* that will control the size of the solution space to investigate, and $\boldsymbol{\mathcal{X}}$, an initial partial assignment.
The major steps of the algorithm are:

1. Obtain the derivatives for each value $x_{ij} \in val(X_i)$ for each X_i in $sc(\mathcal{S})$ given the current set of partial assignments $\boldsymbol{\mathcal{X}}$.
2. Select the b best assignments of variables to create b new sets of partial assignments.
3. Repeat steps 1 and 2 until there are no more marginalized variables to choose from.
4. Return the complete assignment with the highest value.

We now examine these steps in detail.

1. Obtain the derivatives for each value $x_{ij} \in val(X_i)$ for each X_i in $sc(\mathcal{S})$ given the current set of partial assignments $\boldsymbol{\mathcal{X}}$.

   This step runs in linear time on the size of the SPN, following the referenced algorithm for finding the derivatives described in [Dar03]. We start with a set of partial evidences $\boldsymbol{\mathcal{X}}$, having either single assignments for some variables, such as $\{x_i\}$, or having variables marginalized, such as $\{x_0, x_1, \ldots, x_n\}$.

   Each value x_{ij} for each variable X_i will generate a new partial assignment having $X_i = x_{ij}$ with the other variables having the same value (or values) as in $\boldsymbol{\mathcal{X}}$. The derivatives will result in the values for each of these generated partial assignments.

   For example, given an SPN over the variables X and Y where $val(X) = \{x_0, x_1\}$ and $val(Y) = \{y_0, y_1, y_2\}$, and starting with the partial assignments representing all possible values $\boldsymbol{\mathcal{X}} = \{val(X) \times val(Y)\}$, the algorithm for finding the derivatives will find 6 different values, one for each assignment of each variable, which represent the probabilities of that assignment with the other variables marginalized, that is: $\mathcal{S}(x_0)$, $\mathcal{S}(x_1)$, $\mathcal{S}(y_0)$, $\mathcal{S}(y_1)$, and $\mathcal{S}(y_2)$.[4]

2. Select the b best assignments of variables to create b new sets of partial assignments.

   Continuing from the last example, the algorithm has obtained 6 values for derivatives. In this step, we select the b variable assignments with the largest values in order to continue the algorithm.

3. Repeat steps 1 and 2 until there are no more marginalized variables to choose from.

   From this point the process will have b sets of partial assignments, each of them with at least one variable with a single value (instead of being marginalized). On the next steps, when referencing the use of the set of partial evidences, $\boldsymbol{\mathcal{X}}$, the algorithm should then use each of the b sets [5] when finding the derivatives.

   This kind of search will then find more partial assignments to be tested, but after finding the values for the derivatives of the b partial evidence sets, it will select only b of them, stopping a potential exponential grow of the number of partial assignments to evaluate.

   With each iteration, each set of partial evidences will have fixed one value for one additional variable, and therefore the maximum number of iterations this algorithm can have is $|sc(\mathcal{S})|$, the number of variables in this SPN.

4. Return the complete assignment with the highest value

   Once the best b partial assignments have all variables set to a single value, we return the one whose evaluation results in the largest value.

[4] We opted to use the SPN symbol $\mathcal{S}$ instead of the Probability function P, but at this point this exchange should be well understood since the SPN is calculating the same values.

[5] This comment may be an important consideration when implementing this algorithm. Given the way that SPNs can be computed, obtaining for example the sum of two evaluations of configurations $X = x_1$ and $X = x_2$ is faster with a data structure able to represent $X \in \{x_1, x_2\}$, since it will obtain this value with one downward pass on the SPN. This is also true for the algorithm that finds the derivatives of an assignment, which is why it works for the derivatives of partial evidences. However, if in one step there are incompatible partial evidences, such as $\{X = x_1, Y \in val(Y)\}$ and $\{X = x_0, Y = 0\}$, with $x_0 \neq x_1$, the algorithm for finding derivatives will need to be run multiple times.

Beam Search Pseudocode

Algorithm 5: Beam Search

Data: An SPN $\mathcal{S}$, a set of partial evidences $\boldsymbol{\mathcal{X}}$, an integer b
Result: A complete assignment $\boldsymbol{x}$

$P \leftarrow \{\boldsymbol{\mathcal{X}}\}$;
repeat
  foreach *set of partial evidences* $\boldsymbol{\mathcal{X}}$ *in* P **do**
    foreach X *having an assignment in* $\boldsymbol{\mathcal{X}}$ **do**
      foreach $x_i \in \boldsymbol{\mathcal{X}}[X]$ **do**
        Obtain the value of $\mathcal{S}(\boldsymbol{\mathcal{X}}_{X=x_i}) = \mathcal{S}((\boldsymbol{\mathcal{X}} \setminus \{X\}) \times \{x_i\})$;
      end
    end
    Remove $\boldsymbol{\mathcal{X}}$ from P;
  end
  foreach *of the* b *largest values* $\mathcal{S}(\boldsymbol{\mathcal{X}}_{X=x_i})$ **do**
    Add $\boldsymbol{\mathcal{X}}_{X=x_i}$ to P;
  end
until *Until only complete evidences remain in* P;
return *The complete assignment* $\boldsymbol{x} \in P$ *of largest value*;

3.1.5 K-Best Tree

The K-Best Tree algorithm was defined in [MJT18] and created after the empirical observation that the MAP configurations on some SPNs correspond to a complete assignment that is represented by one of its Parse Trees. However, not necessarily the Parse Tree with the largest value (these cases are already handled by the Max-Product algorithm).

In order to understand the association between complete assignments and Parse Trees, consider the example from the Max-Product algorithm shown in Figure 3.3. It shows only the nodes selected at the end of the algorithm run, and its indicators are limited to only one value per variable, representing a complete assignment for the problem. Removing the arc from the sum node to the indicator of $X = x_0$ and adding the arc from this same sum node to the Indicator of $X = x_1$ will generate a different Parse Tree, with a different complete assignment ($\{x_1, y_1\}$), and a different evaluated value at the root.

There are multiple Parse Trees that represent the same variable configurations [6], and the K-Best Tree algorithm ranks them by their value, selecting only the K largest ones and returning the one with the largest value when evaluated on the entire SPN.

The major steps of the algorithm are:

1. Change all sum nodes in $\mathcal{S}$ to max nodes.
2. Do a bottom-up evaluation of the SPN propagating the K best assignments upwards.
3. Select the best of the K propagated assignments at the root.

We now examine these steps in detail.

1. Change all sum nodes in $\mathcal{S}$ to max nodes.

   This is the same step done in Max-Product, and with the same desired effect.

[6] This property was named Ambiguity in [MJT18] and identified whenever a complete assignment on the SPN can be associated with more than one Parse Tree. The inverse of that property is called Non-Ambiguity or Selectivity, the property that allows Max-Product to obtain the exact MAP complete assignment.

2. Do a bottom-up evaluation of the SPN propagating the K best assignments upwards.

   The evaluation will now use max nodes instead of sum nodes to compute the results. More than that, given the type of the node, the results may vary:

   - Indicator nodes can only return value 1 indicating the variable on their scope assuming the value they represent. This is not changed on this algorithm.
   - Max nodes will now return a list of the K maximum values obtained by multiplying their children values with their respect weights.
   - Product nodes have disjoint scopes on each child, therefore they will receive a list of K separate assignments for each child. In order to obtain the value for this node, we need to use one assignment from each child node in order to create a complete assignment for the product node.

     For example, take a product node with 2 children, one returning $\{x_0, x_1\}$ (supposing $K = 2$) and the other $\{y_0, y_1\}$. There are 4 combinations of the two possible values on each variable, and so 4 values will be computed on this node, but returning only the $K = 2$ best ones.

     For more examples, and to consider the costs dependent on both the structure of the SPN and K:

     – Considering still $K = 2$ and 3 child nodes, it is easy to see that there will be 8 assignments to test. For 4 child nodes, there will be 16 assignments to test, or 2^4.
     – Considering $K = 3$, and 2 child nodes, and the child nodes returning the assignments $\{x_0, x_1, x_2\}$ and $\{y_0, y_1, y_2\}$, there will be 9 assignments to test. With 3 child nodes, there will be 27 assignments to test.

     It is now easy to see that the number of assignments to test is at most $K^{|ch(P)|}$,[7] although they are limited by the upper bound of the total amount of possible assignments on the scope of the node, that is $\prod_{X \in sc(P)} |val(X)|$.
     Still, implementation-wise, we can limit the number of assignments to check even further by considering our options with the number K, since we are only interested in the K best assignments returned by this node. The values will always be the product of what was obtained from the child nodes, and it is easy to see that, for example, we will not need the assignment obtained by merging all the lowest-valued assignments from the children (It will never be among the K best values).
     We can achieve a faster running-time for this algorithm by only examining the largest-valued assignments from each child. Consider a product node with 2 children, A and B, the first one returning its K best assignments as $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ and the second one as $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_k$, in order of the largest-valued to the lowest. We will always return the assignment resulting from the merging $\mathcal{A}_1 \times \mathcal{B}_1$. After that, we examine other combinations in order of largest to lowest such as $\mathcal{A}_1 \times \mathcal{B}_2$ and $\mathcal{A}_2 \times \mathcal{B}_1$, and so on. For each merging of configurations examined, $\mathcal{A}_i \times \mathcal{B}_j$, we examine in the next steps $\mathcal{A}_{i+1} \times \mathcal{B}_j$ and $\mathcal{A}_i \times \mathcal{B}_{j+1}$, on the ascending order of values.[8]

3. Select the best of the K propagated assignments at the root.

   At the root, the selected K complete assignments can have their pre-computed values (the ones used to sort the list of values) stored, but in order to obtain the correct value of each

[7] This is an important observation for the chapter on experiments, especially given that this Algorithm was first described and tested on [MJT18] where all tested SPNs had binary variables – meaning all sum nodes with only indicator children would return at most 2 configurations. This may not affect the overall tests, since the number of children per product node can still increase the number of configurations to test (and in the article's SPNs, the number of child nodes per product node was not limited).

[8] Following the original implementation in [MJT18], using a priority queue was enough to achieve a similar running time.

configuration on the SPN, each of them will have to be evaluated on the proper structure, that is, using sum nodes instead of max nodes to calculate the value of each of the K complete assignments. This can be done in time proportional to $K|\mathcal{S}|$.

K-Best Tree Pseudocode

Algorithm 6: K-Best Tree

Data: An SPN $\mathcal{S}$, an integer K
Result: A complete evidence $\boldsymbol{x}$

foreach *Node N in $\mathcal{S}$ in reverse topological order* **do**
  if *N is a sum node with weights $\boldsymbol{w}$ and child nodes $\boldsymbol{c}$* **then**
    $M_N \leftarrow$ the K best assignments from $\bigcup_{c \in ch(N)} M_c$;
  else if *N is a product node with child nodes $\{c_1, c_2, \ldots, c_n\}$* **then**
    $M_N \leftarrow \emptyset$;
    foreach *Combination of disjoint assignments in $\{M_{c_1}, M_{c_2}, \ldots, M_{c_n}\}$ called $M_{c'}$* **do**
      $M_N \leftarrow M_N \times M_{c'}$;
    end
    $M_N \leftarrow$ the K best assignments in M_N;
  else
    N `is an Indicator Node of variable` X `having value` x_i
    $M_N \leftarrow \{\{x_i\}\}$;
  end
end
return *The complete assignment $\boldsymbol{x} \in M_{\mathcal{S}}$ of largest value*;

3.2 Branch-and-Bound algorithms: Max Search

Given the lack of algorithms for an exact solution of this problem, a solution that is capable of surpassing the trivial solution is a good improvement. One approach of doing so is a Branch-and-Bound algorithm, which has yet to be bested after the experiments in [MJT18].

3.2.1 The Branch-and-Bound paradigm

Branch-and-Bound is an algorithm design paradigm used to develop algorithms for solving integer and combinatorial problems. Currently, the only non-trivial, exact algorithm for solving the MAP problem in SPNs uses this paradigm, therefore it will be useful to know its general characteristics before delving into one of its implementations.

Branch-and-Bound has 3 main steps:

1. Setup the problem with a graph enumerating all the possible assignments to the variables, and an initial solution.
2. Choice of a node to expand and obtain a solution to the problem having additional restrictions based on the selected variable.
3. Prune the selected node if the solution is worse. Change the solution if a better one has been found.
4. Repeat steps 2 and 3 until no nodes can be expanded, and return the best found solution.

We now explain each step in detail.

- Setup the problem with a graph enumerating all the possible assignments to the variables, and an initial solution.

A simple example for this graph on two binary variables in figure 3.5. It is easy to see that, with more variables or variables with more than 2 values, this graph can grow exponentially.[9]

Each node represents a set of assignments made to a subset of the variables of the problem. On the figure, the root has no assignments (it is an empty set), the nodes on height 1 have an assignment on variable X, and the nodes on height 2 have the assignments of their parent node and an assignment of the variable Y. Each leaf contains a complete assignment to all the variables in the problem, and each of them is a potential solution.

The initial solution can be any assignment easily obtainable (such as a random assignment to all variables), that we can represent with $\boldsymbol{\mathcal{X}}$, and the value obtained with this configuration, represented by $\boldsymbol{x}^*$. The initial solution will also be called the first *bound* to the problem, and since the MAP problem is a maximization one, this Bound is a *lower bound* – we know the solution to the problem will never be lower than this bound. We set the current best solution as $\boldsymbol{\mathcal{X}}$.

The algorithm depends on a division between nodes: active or inactive. Inactive nodes will not be used on subsequent steps. The initial division is to set the root node as active and the rest as inactive.

- Choice of a node to expand and obtain a solution to the problem having additional restrictions based on the selected variable.

  By choosing a node to expand, we will have a restriction to the problem in the form of the set of assignments represented by that node. We can only choose active nodes to expand and, when the choice is made, the node becomes inactive for the rest of the execution.

  The solution found for the problem with the restriction encoded in this node may be feasible to the original problem, such as a complete assignment obtained from one of the leaves. So, we have three choices for what to do on the next step.

- Prune the selected node if the solution is worse. Change the solution if a better one has been found.

  – If the solution is feasible and is better (in our case, larger) than the bound, change the bound to this new value, and set the current best solution to this newly-found one.
  – If the solution is worse than the current bound, we prune this node, ignoring any node that may come as a child to this one (and they will never be active for the rest of the algorithm).
  – If the solution is not feasible, but the value found for this relaxation is larger than our bound, we activate the child nodes of the chosen node. For MAP problems, this value found for this node with a solution that is not feasible is called an upper bound.

- Repeat steps 2 and 3 until no nodes can be expanded, and return the best solution found.

  The algorithm will then repeat until no active nodes can be found on the graph. At this point, the current best solution will be the best solution, and thus returned, and the value associated with that solution is the best value found for the problem.

3.2.2 Max Search and Branch-and-Bound applications

MaxSearch($\mathcal{S}$, $\boldsymbol{\mathcal{X}}$) performs a recursive, depth-first, Branch-and-Bound search on the partial assignments $\boldsymbol{\mathcal{X}}$ in order to find the complete assignment of maximum probability $\boldsymbol{x}^*$ using the SPN $\mathcal{S}$ to find the value of each assignment. In other words, this algorithm finds $\boldsymbol{x}^*$ such that $\boldsymbol{x}^* = argmax_{\boldsymbol{x} \in \boldsymbol{\mathcal{X}}} \mathcal{S}(\boldsymbol{x})$.
The major steps of the algorithm are:

[9] It is also important to note that this graph will always be a tree, and may be referred to as a *search tree*

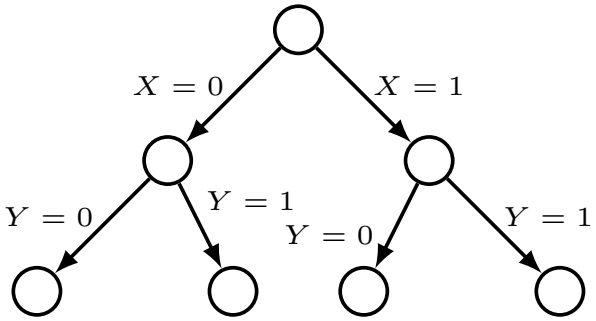


Figure 3.5: *Example of the enumeration created at the start of a Branch-and-Bound algorithm with binary variables.*

1. Obtain an initial assignment solution
2. Check for termination or select a variable to branch
3. Pruning of inferior search trees
4. Repeat steps 2 and 3 until termination is reached

We now examine these steps in detail.

1. Obtain an initial assignment solution.

   The initial solution will be considered the best solution found until this moment of the execution. If this is not a recursive call – meaning it can only be the first call to the algorithm – it is possible to set this solution to any $\boldsymbol{x} \in \boldsymbol{\mathcal{X}}$.

   On subsequent calls, the initial solution is passed on from the previous call. This solution is only changed when the algorithm reaches a leaf of the search tree.

   It is useful to store this solution as an assignment $\boldsymbol{x}^*$ and its value $\mathcal{S}(\boldsymbol{x}^*)$ for future comparisons since this value needs time $O(n)$ to be computed.

2. Check for termination or select a variable to branch

   Termination occurs when there is only one assignment for each variable in $\boldsymbol{\mathcal{X}}$, that is, a complete assignment. In this case, the algorithm does not have to compare its initial solution value because, given the restriction described above, the best solution should be inside the set $\boldsymbol{\mathcal{X}}$. Therefore, it just returns the assignment $\boldsymbol{x} \in \boldsymbol{\mathcal{X}}$.

   If there is more than one assignment in $\boldsymbol{\mathcal{X}}$, then there is at least one variable $X \in \mathit{scope}(\mathcal{S})$ being assigned different values in $\boldsymbol{\mathcal{X}}$. There are two main ways of making this choice:

   (a) Choose any variable that respects the precondition at random (a heuristic we call First Variable)

   (b) Choose the variable whose number of different values being assigned in $\boldsymbol{\mathcal{X}}$ is the lowest (The heuristic described in [MJT18] as Ordering).

3. Pruning of inferior search trees

   For each value x the variable X was assigned, we can obtain upper bounds to the remaining assignments in $\boldsymbol{\mathcal{X}}$ that follow the restriction $X = x$. By comparing these upper bounds with the best value we found so far (from the best solution $\boldsymbol{x}^*$) we can shrink the set $\boldsymbol{\mathcal{X}}$. The article [MJT18] gives two algorithms for doing this:

   (a) Marginal Checking: Compare the current best value $\mathcal{S}(\boldsymbol{x}^*)$ with $\mathcal{S}(\boldsymbol{\mathcal{X}})$. If the current best value is higher, we can remove all assignments from $\boldsymbol{\mathcal{X}}$.

It is easy to see that this algorithm performs the evaluation of the SPN marginalizing the variables that have not already been added to the solution on the lower bound, and it fits more easily with the definition of the Branch-and-Bound step shown previously.

(b) Forward Checking: Compare the current best value $\mathcal{S}(\boldsymbol{x}^*)$ with the derivatives of each individual variable assignment $\{x'\}$ in $\boldsymbol{\mathcal{X}}$. As described before with the Beam Search algorithm, all these values can be obtained at once with an algorithm that runs in linear time.

The derivatives in an SPN $\mathcal{S}$ give the following information: for each variable X' and set of assignments $\{x'_1, x'_2, \ldots, x'_n\}$ in $\boldsymbol{\mathcal{X}}$, we have the other variables $\boldsymbol{X} \setminus X'$ and their assignments marginalized,[10] and obtain the values of $\mathcal{S}((\boldsymbol{\mathcal{X}} \setminus X') \times \{x'_i\})$ for each $x'_i \in \{x'_1, x'_2, \ldots, x'_n\}$. We do not marginalize all the values that a variable can be assigned; only the variables for which there is a corresponding assignment in $\boldsymbol{\mathcal{X}}$.

After each of these comparisons, we may obtain many individual assignments $\{x'_i\}$ that we can remove from the set of assignments $\boldsymbol{\mathcal{X}}$. If we remove any of these, we must recheck the derivatives, since their values will have changed.

If we did not obtain any assignment we can remove from $\boldsymbol{\mathcal{X}}$, that is, all values are equal to or greater than the current best value, we continue the algorithm with the remaining partial assignments.

Unlike the Marginal Checking algorithm, Forward Checking is capable of pruning many nodes of the search tree in one pass, which speeds up the checking for upper bounds[11].

4. Repeat steps 2 and 3 until termination is reached

If the modified partial assignments $\boldsymbol{\mathcal{X}}'$ obtained from step 3 is not empty, we can continue the search with the additional restriction $X = x$ of the variable obtained in step 2. We, then, call the algorithm again, passing forward the best solution (and its best value), and $\boldsymbol{\mathcal{X}}'$. This new set of assignments will have only one assignment for the variable X, and the next search will continue with a different variable, which ensures that, at some time, the algorithm will terminate.

At any time the algorithm can be stopped and return its current best solution, and therefore it is an Anytime Algorithm.

[10] If this use of the term "marginalized" is confusing, section 2.1.2 will explain it in more details.

[11] This is an important efficiency consideration for the Max-Search algorithm, and the reason why this checking algorithm performed the best on the original article. It is capable of not only obtaining multiple upper bounds given the variables not yet set, but it skips multiple steps of expanding nodes on the Branch-and-Bound search.

Max-Search Pseudocode

Algorithm 7: Max-Search

Data: An SPN $\mathcal{S}$, an initial complete assignment $\boldsymbol{x}$, a set of assignments to search $\boldsymbol{\mathcal{X}}$
Result: The MAP solution of $\mathcal{S}$: $\boldsymbol{x}^*$

```
1  lower ← S(x);
2  if X is a complete assignment then
3  |  return x, the only assignment in X
4  end
5  X ← one variable from sc(S) with more than one assignment in X;
6  foreach Assignment {x_i} in X for a variable X do
7  |  X' ← (X \ {X}) × {x_i};
8  |  X' ← CheckingFunction(X', {x});
9  |  if X' ≠ ∅ then
10 |  |  x ← Max-Search(S, x, X');
11 |  end
12 end
13 return The complete assignment x
```

Algorithm 8: Marginal Checking

Data: An SPN $\mathcal{S}$, an Assignment $\boldsymbol{x}$, a set of possible partial assignments $\boldsymbol{\mathcal{X}}$
Result: A modified set of partial assignments $\boldsymbol{\mathcal{X}}'$

```
1  if S(X) ≥ S(x) then
2  |  return X
3  else
4  |  return ∅
5  end
```

Algorithm 9: Forward Checking

Data: An SPN $\mathcal{S}$, an Assignment $\boldsymbol{x}$, a set of partial assignments $\boldsymbol{\mathcal{X}}$

```
1  repeat
2  |  foreach {x} in X for a variable X do
3  |  |  D_x ← ∂S/∂x (X);
4  |  end
5  |  foreach Variable X ∈ X do
6  |  |  foreach {x} in X[X] do
7  |  |  |  if S(x) ≥ D_x then
8  |  |  |  |  X ← (X[X] \ {x}) × X[X \ {X}];
9  |  |  |  end
10 |  |  end
11 |  end
12 until X is no longer changed;
13 return X
```

Heuristics for variable selection

On line 7 of the Max Search algorithm, we select one assignment of $\{x_i\}$ for some variable X that still has more than one value to be chosen. Which value should we choose? This question would, in this algorithm, have the same answer as "Which variable and assignment would make the algorithm finish faster?". With Max Search, convergence of the result is related to finding the lower

bound, and expecting the next found upper bounds to not surpass it. The original article for this algorithm [MJT18] described one heuristic for this choice, and left another one implicit:

- Ordering (the described heuristic): Choose the variable that has the least amount of values to choose. So, in $\boldsymbol{\mathcal{X}}$ there are the sets of values each variable X_i can assume, represented by $\boldsymbol{\mathcal{X}}[X_i]$. We choose the variable $X = argmin_{X_i}|\boldsymbol{\mathcal{X}}[X_i]|$.
- Random Selection (the implicit one): Either choose the variable that can be selected the fastest, or a random variable (if random selection is faster). This would be the heuristic to compare as a baseline, since it is supposed to take the least time, in comparison to any other.[12]

And in our work we have also developed and tested additional heuristics for comparison:

- Lowest (or Largest) Marginal: With the derivative calculation on the forward checking algorithm, it is possible to obtain the marginal probabilities of every value still to be checked. For example, if we still have not set the value of a variable X,[13] and we are still searching on the set of assignments $\boldsymbol{\mathcal{X}}$, and X can have the value of either x_1 or x_2, we are able to obtain both $P((\boldsymbol{\mathcal{X}} \setminus X) \times \{x_1\})$ and $P((\boldsymbol{\mathcal{X}} \setminus X) \times \{x_2\})$, which are the marginal values of these two assignments.[14] These heuristics will choose, from $\{x_1, x_2\}$, the assignment that has either the lowest or the largest marginal value.
- Lowest (or Largest) Entropy: We use the definition and, especially, the alternative view of entropy described in [KF09] in order to create a heuristic to select the variable and value. Considering a variable X with values $\{x_1, x_2, \ldots, x_n\}$ still to be set, the entropy of $X = x$ is $H_P(X = x) log \frac{1}{P(X=x)}$.[15]. This value can be interpreted as a measure of uncertainty about the assignment of $\{X = x\}$. These heuristics will select either the variable and value with the lowest or the largest entropy.

[12] In our implementation, the variables are stored in a set and we select the first element returned from an instantiated iterator over the set. Because of the implementation, this is equivalent as selecting a random variable from the set.

[13] That is, the algorithm has not run for long enough to prune all branches of this variable except for one $X = x_i$ that will be returned in the final answer.

[14] As explained before, and can be noted here, this marginalization is partial. It is common for some variable Y with $val(Y) = \{y_1, y_2, \ldots, y_n\}$ to be restricted to some values such as $\{y_1, y_2\}$ in $\boldsymbol{\mathcal{X}}$ because of some previous iteration of the pruning This behaviour is expected, but this clarification is helpful to understand this step.

[15] For completion, the entropy of X is $\sum_{x \in val(X)} H_P(X = x)$

Chapter 4

Experiments and Results

We have described the algorithms and the MAP problem we want to examine. Now we can begin describing the experiments we did in order to answer the questions at the introduction.

4.1 Experiment Setup

The SPNs created for these experiments, with one exception, have been learned with repeated executions of the Learn-SPN algorithm, each one testing a different combination of parameters, being $K \in \{2, 3, 4, 5\}$, the number of clusters, and $p \in \{0.0001, 0.001, 0.01, 0.1, 0.0002, 0.002, 0.02, 0.2, 0.0005, 0.005, 0.05, 0.5, 0.0007, 0.007, 0.07, 0.7, 0.0009, 0.009, 0.09, 0.9\}$ the p-value to be used on the test of independence. We performed a greedy-search on the learned SPNs by their log-likelihood, on each dataset. For the Optdigits dataset, we divided the training dataset into 10 parts, one for each digit, and learned 10 SPNs in the same way the other SPNs were learned. Afterwards, the main SPN was obtained by adding a Sum Node as a root having each of the learned SPNs as a child, with weight 0.1.

The table 4.1 shows relevant statistics of the SPNs used in the experiments. In order to understand the problem size, we need to point out that some SPNs are binary, and thus all their variables can be assigned either 0 or 1, and others are not, meaning that at least one variable can have 3 or more values. The size of the MAP problem, on an algorithm that lists all possible configurations for the $\boldsymbol{X}$ variables in an SPN will list $S = \prod_{X \in \boldsymbol{X}} |val(X)|$ configurations. This number gets exponentially large in some SPNs, and thus we decided to show $ln(S)$ instead, simplifying the comparisons between these measures.

It is important to note that, given the scenarios we have chosen, the problem size will change with the test. For example, in an SPN with 30 variables, we may choose 10 for the Query (Q), 10 for the Marginal, and 10 for the evidence. The problem size for this specific scenario will be $S' = \prod_{X \in Q} |val(X)|$. We have chosen 30 scenarios for each SPN such that each scenario would guarantee that the solution found by the Max-Product algorithm would not be optimal, and thus allowing for the other algorithms, given the allocated time, to find a better solution. For each scenario, we choose 1/3 of the variables for each set: Query, Marginalized, and Evidence. Where applicable, we have chosen these scenarios from the list of scenarios already chosen in [MJT18], but since we have additional SPNs in our experiments, we have generated our own scenarios for them.[1]

The running time limit varies with the experiment, being 10 minutes for the experiments comparing the results and running time of each algorithm, and 5 minutes for the others. When the time runs out, for the algorithms that can be stopped and still return a meaningful value (such as Local Search and Max Search), the best obtained result will be used, while for the others, the result is 0.

For the algorithms that depend on parameters, such as Beam Search and K-Best Tree, we use $b = 3$ for beam size and $K = 3$ for the number of evaluated parse trees, both for the algorithm's efficiency (and thus being able to finish before the time limit) and in order to reduce memory usage.

[1] The SPNs and the scenarios used for the MAP problem can be found at https://gitlab.com/pgm-usp/learned-spns/

SPN	Variables	⊕	⊗	◎	Height	Problem Size
Nltcs	16	1435	388	32	27	11.09
Msnbc	17	9417	2155	34	23	11.78
House-Vote	17	139	27	34	7	11.78
Mushroom	23	1658	245	119	30	33.13
Hepatitis	20	99	27	366	8	37.29
Kdd	64	7606	1021	128	47	44.36
Plants	69	31379	5079	138	56	47.83
Baudio	100	44263	5667	200	104	69.31
Jester	100	22486	2390	200	140	69.31
Bnetflix	100	30169	5917	200	31	69.31
Accidents	111	26222	6264	222	57	76.94
Mushrooms	112	4089	833	224	29	77.63
Splice	61	77540	2145	243	26	84.28
Tretail	135	1565	323	270	34	93.57
Us_Census	68	140722	20752	644	58	107.98
Pumsb_Star	163	20998	4882	326	36	112.98
Dna	180	10182	1638	360	54	124.77
Kosarek	190	17363	1769	380	35	131.7
Optdigits	65	1406	216	1098	12	183.63
Msweb	294	9040	1115	588	74	203.79
Tmovie	500	50815	2369	1000	56	346.57
Book	500	48521	1972	1000	66	346.57
Nips	500	4362	704	1000	11	346.57
Digits	601	21655	1930	1210	27	418.19
Cwebkb	839	219005	3164	1678	270	581.55
Voting	1359	84391	5345	2718	75	941.99
Ad	1556	36619	1170	3112	43	1078.54

Table 4.1: *Statistics for the learned SPNs on the number of Variables, Nodes, Height, and problem size.*

For the Max Search algorithm, we follow the chosen heuristics at [MJT18] in order to reach somewhat comparable results to theirs. That is, we use ordering as the heuristic to choose the next variable and values to expand, and we use forward checking for pruning.

The experiments ran on a machine with the following configurations:[2]

- CPU: Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
- Memory: 64 GiB, L1 Cache: 384KiB, L2 Cache: 1536KiB, L3 Cache: 12MiB

We obtained the datasets from the publicly available repository of Density Estimation Benchmark Datasets (DEBD) [LD10] [VHD12] [BDC+15] [LM11][3] and from the UCI Machine Learning Repository [DG17].[4]

With this setup and the following benchmark, we look to answer the first question from the introduction: given a limited time-frame, which algorithms perform best in comparison with Max-Product?

4.2 Comparing Running time and found value

Table 4.2 shows the average running time of each algorithm (with their standard deviation) after the experiments on each of the 30 MAP scenarios. Table 4.3 shows how better is the value

[2]Time measures were made using the Standard Time Quantification library (Rust's std::stime).
[3]Available at https://github.com/arranger1044/DEBD
[4]Available at https://archive.ics.uci.edu

SPN	MP	MP + LS	AMP	AMP + LS	BS	KBT	Max Search
Nltcs	1.0 ±0.0ms	1.5 ±2.01ms	7.07 ±8.73ms	0.97 ±0.18ms	217.47 ±75.61ms	56.97 ±20.71ms	156.83 ±67.83ms
Msnbc	66.2 ±19.35ms	104.23 ±45.13ms	207.4 ±37.53ms	51.87 ±25.88ms	1.8 ±0.31s	443.37 ±82.59ms	1.2 ±0.38s
House-Vote	0.0 ±0.0ms	0.0 ±0.0ms	0.1 ±0.54ms	0.0 ±0.0ms	5.32 ±5.49ms	1.0 ±0.0ms	1.16 ±3.0ms
Mushroom	5.27 ±3.2ms	47.8 ±27.57ms	28.73 ±11.42ms	25.6 ±11.24ms	307.5 ±71.84ms	46.13 ±15.77ms	279.7 ±165.35ms
Hepatitis	0.0 ±0.0ms	1.55 ±1.43ms	1.0 ±0.0ms	1.0 ±0.0ms	26.84 ±9.38ms	0.74 ±3.95ms	42.16 ±34.17ms
Kdd	74.03 ±24.63ms	170.57 ±80.84ms	144.53 ±25.62ms	111.73 ±37.2ms	4.6 ±1.34s	805.9 ±260.79ms	94.9 ±176.29s
Plants	314.33 ±105.62ms	792.57 ±315.61ms	643.97 ±237.33ms	571.83 ±262.35ms	23.4 ±5.97s	2.7 ±0.86s	136.34 ±175.75s
Baudio	649.7 ±244.97ms	2.4 ±1.04s	1.75 ±0.54s	1.58 ±0.69s	68.88 ±10.57s	8.94 ±1.62s	10 ±0.01m
Jester	293.53 ±73.64ms	1.41 ±0.72s	889.7 ±303.48ms	649.2 ±210.24ms	37.62 ±8.91s	5.33 ±1.32s	9.85 ±0.81m
Bnetflix	507.17 ±124.43ms	1.67 ±0.92s	853.47 ±190.59ms	1.1 ±0.47s	30.86 ±6.06s	4.64 ±1.09s	10 ±0m
Accidents	402.7 ±124.86ms	1.33 ±0.59s	708.97 ±258.41ms	1.07 ±0.58s	33.43 ±7.85s	2.33 ±0.69s	9.09 ±2.76m
Mushrooms	77.03 ±22.46ms	137.34 ±38.34ms	102.24 ±32.21ms	124.0 ±42.84ms	5.98 ±1.71s	404.45 ±101.7ms	1.08 ±0.94s
Splice	471.1 ±157.55ms	1.44 ±0.38s	953.27 ±346.36ms	1.32 ±0.34s	16.18 ±2.24s	4.08 ±1s	10 ±0m
Tretail	11.47 ±8.71ms	61.97 ±17.01ms	23.2 ±6.83ms	37.23 ±10.32ms	1.62 ±0.65s	118.8 ±20.02ms	10 ±0m
Us_Census	362.67 ±106.91ms	3.66 ±1.01s	853.63 ±108.84ms	3.29 ±0.76s	52.87 ±2.4s	513.43 ±95.96ms	2.28 ±2.83m
Pumsb_Star	512.8 ±158.52ms	1.91 ±0.78s	717.33 ±211.83ms	1.11 ±0.29s	40.34 ±7.34s	1.81 ±0.51s	10 ±0m
Dna	175.57 ±58.25ms	845.13 ±665.32ms	309.37 ±97.17ms	595.5 ±546.81ms	22.46 ±4.52s	3.24 ±1.04s	10 ±0m
Kosarek	381.9 ±104.79ms	1.86 ±0.85s	549.33 ±108.82ms	1.10 ±0.46s	38.04 ±6.99s	5.15 ±1.24s	10 ±0m
Optdigits	28.87 ±7.27ms	611.03 ±334.48ms	65.6 ±17.85ms	429.97 ±173.82ms	1.03 ±0.39s	65.43 ±23.26ms	10 ±0m
Msweb	281.4 ±88.08ms	1.28 ±0.55s	381.17 ±105.35ms	623.93 ±214.05ms	38.15 ±6.82s	2.2 ±7.24s	10 ±0m
Tmovie	2.49 ±0.75s	13.47 ±4.7s	3.11 ±0.82s	8.91 ±3.42s	4.69 ±0.39m	33.07 ±6.58s	10 ±0m
Book	2.52 ±0.67s	15.33 ±5.8s	3.4 ±0.94s	10.05 ±4.07s	5.03 ±0.39m	47.19 ±8.81s	10 ±0m
Nips	232.24 ±63.46ms	1.24 ±0.5s	252.55 ±63.97ms	1.15 ±0.51s	34.63 ±6.27s	1.87 ±0.64s	10 ±0m
Digits	485.8 ±231.67ms	1.94 ±0.87s	513.9 ±206.11ms	1.45 ±0.7s	39.57 ±4.16s	2.9 ±0.8s	10 ±0m
Cwebkb	13.39 ±3.69s	73.54 ±26.24s	30.37 ±7.22s	56.97 ±27.34s	10 ±0m	5.37 ±0.68m	10 ±0.03m
Voting	10.46 ±2.93s	37.03 ±14.58s	11.55 ±3.04s	35.4 ±10.18s	10 ±0m	1.75 ±0.22m	10 ±0m
Ad	4.85 ±1.53s	15.55 ±3.12s	5.17 ±1.61s	15.31 ±3.19s	18.07 ±95.73s	40.86 ±8.79s	10 ±0m
Averages	1.44s	6.58s	2.35s	5.29s	84.97s	22.1s	6.85m
Medians	314.33ms	1.33s	549.33ms	1.07s	30.86s	2.33s	10m
Average Rankings	1.15	10.15	4.48	5.19	21.74	13.33	25.85
Standard Deviation Rankings	0.53	2.86	2.10	2.27	3.02	4.24	3.56

Table 4.2: *Average Running time with standard deviation for Approximate MAP algorithms. The time for the Local Search algorithms does not include the time used to find their initial solutions.*

found by other algorithms in comparison with Max-Product, that is, given the Max-Product value is v, the other columns show the value found by the other algorithms, v' as $\frac{v'}{v}$.

It is important to note that, in time computations, we do not sum the time needed to obtain the initial configuration on the algorithms that need one. Local Search and Max Search both can use an initial solution to start their search. In order to obtain the initial solution, we use either Max-Product or Argmax-Product for Local Search, depending on the column, and Max-Product for Max Search. This may change the conclusions to be taken from the data, since the Local Search algorithm could be faster than Max-Product (see the MSNBC line on the time table) when the total time taken to obtain that solution would be closer to the sum of both these algorithms' running times.

The rank is given by the relative "win" of the algorithm on each SPN, that is, having the largest value in proportion to the value found by Max-Product, or having the lowest running time. The algorithm that found the best value gets rank 1, and the algorithms that found the same value also get rank 1. The algorithm that finds the next best values get higher ranks, therefore the lower the rank value, the better. On the time table, the fastest algorithms get lower ranks accordingly.

The running time results were as expected, with Max-Product being overall faster than any other algorithm, and Max Search being the slowest. Argmax-Product being slightly slower than Max-Product conforms with the necessary reevaluations of smaller sub-SPNs.

The differences in both Local Searches are not accounted by the algorithm, and the difference in their averages is of about 1.3 seconds. In comparison to the other algorithms, they are close enough to dismiss this difference.

We conclude that, given the allocated time of ten minutes, running the Argmax-Product algorithm should be the best, and if more time can be allocated, using the found solution to run the Local Search algorithm looks, on average, to be the best way of obtaining a good solution. If the SPN is small enough for the Max Search algorithm to run until completion, then there is no reason to not use it, but few of our instances met this requirement.[5]

[5] A result that was also found in [MJT18], since Max Search resulted in various timeouts on their experiments as well.

SPN	MP + LS	AMP	AMP + LS	BS	KBT	Max Search
Nltcs	1.61 ±0.65	1.65 ±0.65	**1.66 ±0.64**	**1.66 ±0.64**	1.00 ±0.0	**1.66 ±0.64**
Msnbc	1.27 ±0.58	**1.51 ±0.71**	**1.51 ±0.71**	**1.51 ±0.71**	1.00 ±0.0	**1.51 ±0.71**
House-Vote	1.00 ±0.0	**1.02 ±0.11**	**1.02 ±0.11**	1.01 ±0.12	1.00 ±0.0	**1.02 ±0.11**
Mushroom	1.12 ±0.16	**1.22 ±0.16**	**1.22 ±0.17**	**1.22 ±0.17**	1.00 ±0.0	**1.22 ±0.17**
Hepatitis	1.15 ±0.39	**1.16 ±0.39**	**1.16 ±0.39**	**1.16 ±0.39**	1.00 ±0.0	**1.16 ±0.39**
Kdd	1.93 ±2.05	4.76 ±5.38	**4.77 ±5.38**	4.75 ±5.39	1.00 ±0.03	4.69 ±5.43
Plants	1.98 ±3.13	**3.16 ±3.78**	**3.16 ±3.78**	2.74 ±3.84	1.00 ±0.02	3.00 ±3.77
Baudio	2.28 ±2.41	1.03 ±3.12 E+01	1.03 ±3.11 E+01	**9.92 ±31.2**	1.00 ±0.03	1.00 ±0.0
Jester	3.11 ±2.65	5.48 ±5.29	**5.54 ±5.31**	4.91 ±5.11	9.92 ±0.42 E-01	1.01 ±0.03
Bnetflix	1.80 ±1.89	4.46 ±5.84	**4.49 ±5.83**	4.18 ±5.76	9.90 ±0.44 E-01	1.00 ±0.0
Accidents	2.29 ±4.44	4.77 ±5.94	**4.81 ±5.93**	4.76 ±5.96	1.00 ±0.0	1.06 ±0.21
Mushrooms	1.00 ±0.0	1.01 ±0.06	1.01 ±0.06	**1.02 ±0.11**	1.00 ±0.0	**1.02 ±0.11**
Splice	1.09 ±0.48	**6.14 ±12.4**	**6.14 ±12.4**	3.95 ±12.1	1.00 ±0.0	1.00 ±0.0
Tretail	3.20 ±3.34	**3.35 ±3.26**	**3.35 ±3.26**	**3.35 ±3.26**	1.00 ±0.0	1.00 ±0.0
Us_Census	1.04 ±0.15	**1.51 ±0.63**	**1.51 ±0.64**	1.49 ±0.66	1.00 ±0.0	1.46 ±0.59
Pumsb_Star	1.57 ±0.82	**6.36 ±14.4**	**6.36 ±14.4**	5.95 ±14.3	1.00 ±0.01	1.00 ±0.0
Dna	6.58 ±9.38	3.18 ±5.91 E+01	**3.20 ±5.9 E+01**	2.64 ±5.49 E+01	1.02 ±0.07	1.00 ±0.0
Kosarek	1.40 ±2.73 E+02	**2.38 ±5.59 E+02**	**2.38 ±5.59 E+02**	1.99 ±5.33 E+02	1.15 ±0.5	1.00 ±0.0
Optdigits	6.33 ±20.3	**7.96 ±21.0 E+01**	**7.96 ±21.0 E+01**	7.47 ±18.8 E+01	9.97 ±0.06 E-01	1.00 ±0.0
Msweb	5.38 ±5.1	5.88 ±5.04	5.88 ±5.04	**5.89 ±5.03**	1.38 ±0.7	1.00 ±0.0
Tmovie	1.15 ±5.91 E+05	**1.60 ±6.37 E+05**	**1.60 ±6.37 E+05**	1.15 ±5.9 E+05	1.03 ±0.1	1.00 ±0.0
Book	6.20 ±33.9 E+04	**4.38 ±23.9 E+05**	**4.38 ±23.9 E+05**	4.37 ±23.9 E+05	1.45 ±1.31	1.00 ±0.0
Nips	1.13 ±0.12	**1.06 ±4.47 E+03**	**1.06 ±4.47 E+03**	9.67 ±44.8 E+02	9.84 ±0.64 E-01	1.00 ±0.0
Digits	1.03 ±0.04	**1.81 ±6.77 E+02**	**1.81 ±6.77 E+02**	1.74 ±6.78 E+02	1.00 ±0.0	1.00 ±0.0
Cwebkb	6.60 ±26.8 E+06	**9.05 ±40.3 E+07**	**9.05 ±40.3 E+07**	—	2.61 ±5.9	1.00 ±0.0
Voting	1.67 ±3.63	**1.36 ±5.96 E+03**	**1.36 ±5.96 E+03**	—	1.00 ±0.0	1.00 ±0.0
Ad	**6.67 ±0.0 E-02**	**6.67 ±0.0 E-02**	**6.67 ±0.0 E-02**	—	**6.67 ±0.0 E-02**	**6.67 ±0.0 E-02**
Averages	2.51E+05	**3.37E+06**	**3.37E+06**	2.05E+04	1.06	1.26
Medians	1.80	5.48	**5.54**	4.18	1.00	1.00
Average Rankings	9.56	3.11	**1.37**	6.93	15.85	12.04
Standard Deviation Rankings	2.83	1.87	1.15	6.93	4.57	7.78

Table 4.3: *Average of values found by Approximate Algorithms in proportion to the Max-Product value.*

4.3 Testing Heuristics for Max Search

From the test results on the last section, we know that Max Search is the slowest algorithm on average. However, Max Search is an Anytime Algorithm, and therefore we can stop it at anytime and obtain a usable solution. The question then becomes the second questions listed at the introduction: Which heuristics in Max Search find good solutions faster?

The ways we have to test this is by changing the heuristics for choosing the next variable to be expanded, and for the nodes to be pruned. Table 4.4 shows the results of our experiments with the following heuristics: First Variable (FV), Lowest and Largest Marginals (LoM and LaM), Lowest and Largest Entropy (LoE and LaE), Ordering (Ord), Marginal Checking (MC), and Forward Checking (FC). Each "win" marked on the table signifies an algorithm that reached, in the time limit of 5 minutes, the best value in the least amount of time, that is, given their final value, we give the win to the combination of heuristics that reached such value the soonest. If two or more heuristics reached the same value at approximately the same time (with a margin of error of ten milliseconds).

The results vary depending on the SPN. For a small SPN such as NLTCS, the algorithm runs so fast that most heuristics will receive wins, but the heuristics that involve less calculations (such as getting the first variable, or simply calculating the marginals) reach the best solutions first.

One surprising result was the heuristics that we chose for the experiments in the previous section, and the heuristics used in [MJT18] perform, on average and in total, worse than most other combinations of heuristics. The variable-choice heuristics that require less calculations (First Variable, Lowest Marginal, Largest Marginal) tend to perform better than the others. For the pruning heuristic, Marginal Checking still receives more wins in both total and average.

4.4 Missing Instances on Classifiers

Another use of the MAP algorithms, as has been explained before, is auto-completion. We can accomplish that by seeing the variables in the Query set as missing variables, and those in the

SPN	FV + MC	LoM + MC	LaM + MC	LoE + MC	LaE + MC	Ord + MC	FV + FC	LoM + FC	LaM + FC	LoE + FC	LaE + FC	Ord + FC
Nltcs	20	20	7	4	2	4	6	10	12	14	12	14
Msnbc	10	10	0	0	0	0	0	0	1	1	11	8
House-Vote	24	19	2	1	1	3	3	3	27	26	10	11
Mushroom	9	16	0	0	0	0	5	3	0	0	0	0
Hepatitis	7	7	0	0	4	5	0	0	20	18	1	0
Kdd	2	4	0	0	0	0	0	0	13	14	0	0
Plants	3	9	0	0	0	0	0	0	12	6	0	1
Baudio	4	3	4	6	4	1	2	2	3	2	0	0
Jester	6	7	4	3	3	3	3	7	2	2	2	2
Bnetflix	5	1	5	4	3	2	4	1	4	5	0	0
Accidents	6	6	6	4	5	0	7	2	1	1	0	0
Mushrooms	3	9	0	0	0	0	0	0	9	9	0	0
Splice	0	4	3	4	2	3	4	2	1	2	2	4
Tretail	19	21	14	11	13	13	13	5	8	4	6	4
Us_Census	3	3	2	0	4	2	4	1	3	1	3	4
Pumsb_Star	4	6	3	0	5	3	1	0	7	4	0	0
Dna	19	13	11	6	5	6	8	7	10	10	8	8
Kosarek	6	8	3	7	3	3	2	3	1	2	1	2
Optdigits	8	2	7	7	5	1	4	1	0	0	0	0
Msweb	2	1	13	1	9	9	7	5	1	1	1	1
Tmovie	8	8	4	4	3	1	3	2	0	1	1	0
Book	2	2	7	3	5	5	5	4	1	1	0	0
Nips	4	12	5	1	1	1	0	1	5	5	0	0
Digits	2	7	1	4	2	1	0	1	5	5	1	2
Cwebkb	1	1	8	1	1	4	3	1	0	4	6	1
Voting	7	3	3	2	0	6	3	1	1	3	4	1
Ad	5	9	6	6	4	5	8	4	11	11	7	3
Total	189	211	118	79	84	81	95	66	158	152	76	66
Average	7.00	7.81	4.37	2.92	3.11	3.00	3.51	2.44	5.85	5.62	2.81	2.44

Table 4.4: *Number of winning counts by heuristics on Max Search.*

Evidence set as given, fixed variables. This is a different way of evaluating the algorithms since both the obtained value and the complete assignment are important in order to compare the solution's quality.

We measured three classifiers: A Support-Vector Machine, a Random Forest, and a Decision Tree, all from the scikit-learn library [PVG+11] with their default parameters. We have chosen the Digits dataset, a small dataset (229 test instances) of binary images of digits for classification, and the Optdigits dataset, a larger dataset (1797 test instances) with the same purpose but with values ranging from 0 to 16.[6] We also learn an SPN from the same training dataset, and use it in order to auto-complete selected test instances with values removed.

In order to formulate a MAP problem with these instances, we remove the first $\frac{1}{3}$ values and add the variables corresponding to them to the query set. We then remove the next $\frac{1}{3}$ values and add the variables to the marginalized set. The rest of the variables, including the classification label, are given as evidence.

Table 4.5 shows the base accuracy of each tested classifier, that is, the accuracy of classification with no missing instances and no auto-completion. We can consider this an upper limit to the classification accuracy on the auto-completed instances.

Besides the Max-Product and Argmax-Product algorithms we described before, we also have compared:

- Random Completion: we complete the missing values with a random value, chosen uniformly from the set of values from each variable.

- Mode Completion: we complete the missing values with the value that occurs the most on the training dataset for that variable.

On tables 4.6 and 4.7 we have the accuracy as the percentage of correctly-classified images for each classifier, using each algorithm for auto-completion.

We can see that using the SPN and one of the MAP algorithms is, in the majority of scenarios, better than using random values or the mode for completing instances with very few exceptions. A smaller but inconsistent difference is in the accuracy provided by Argmax-Product. It shows that,

[6] We have chosen these two datasets because of their purpose for classification tasks (the ideal scenario for this test), and because they complement each other in terms of the size of the MAP problem: Optdigits has less pixels, but more possible values per pixel, while Digits has a larger problem size.

Classifier	Digits	Optdigits
Random Forest	96%	80%
SVM	96%	96%
Decision Tree	89%	85%
SPN	89%	35%

Table 4.5: *Classification accuracy using the 4 classifiers with no missing data for the tested Datasets.*

Classifier	Algorithm	Accuracy
Random Forest	Random Completion	33%
SVM		9%
Decision Tree		18%
Random Forest	Mode Completion	35%
SVM		9%
Decision Tree		16%
Random Forest	Max-Product	69%
SVM		9%
Decision Tree		31%
Random Forest	Argmax-Product	68%
SVM		9%
Decision Tree		31%

Table 4.6: *Accuracy Results for auto-completion tasks on the Digits Dataset.*

Classifier	Algorithm	Accuracy
Random Forest	Random Completion	28%
SVM		10%
Decision Tree		18%
Random Forest	Mode Completion	28%
SVM		28%
Decision Tree		18%
Random Forest	Max-Product	45%
SVM		57%
Decision Tree		27%
Random Forest	Argmax-Product	49%
SVM		67%
Decision Tree		31%

Table 4.7: *Accuracy Results for auto-completion tasks on the Optdigits Dataset.*

sometimes on the Digits dataset, we may get a lower accuracy rate. On the Optdigits dataset, it is consistently higher accuracy.

We know from the experiments on the previous sections that the values obtained by Argmax-Product are, on average, higher than the values obtained by Max-Product, but it is hard to see if there is any relation between this higher value and the task of classification. Because of this problem, we have compared the log-values[7] of each solution generated by Max-Product and Argmax-Product against the mean square error of the auto-completed solution (which is a rough measure of how far from the original image the auto-completion is). The results are in figure 4.1. We use blue points on the chart to show instances where the difference in log-values is zero, and red points to show where the Argmax-Product reached a higher value.

On the digits dataset we cannot see a clear relation between the value being better and a lower error (in fact, sometimes the solution is much better but the error is also higher). However, we can see a relation between the log-value and the lower error rates for optdigits, which could partially explain the increase in accuracy we have seen before on table 4.7.

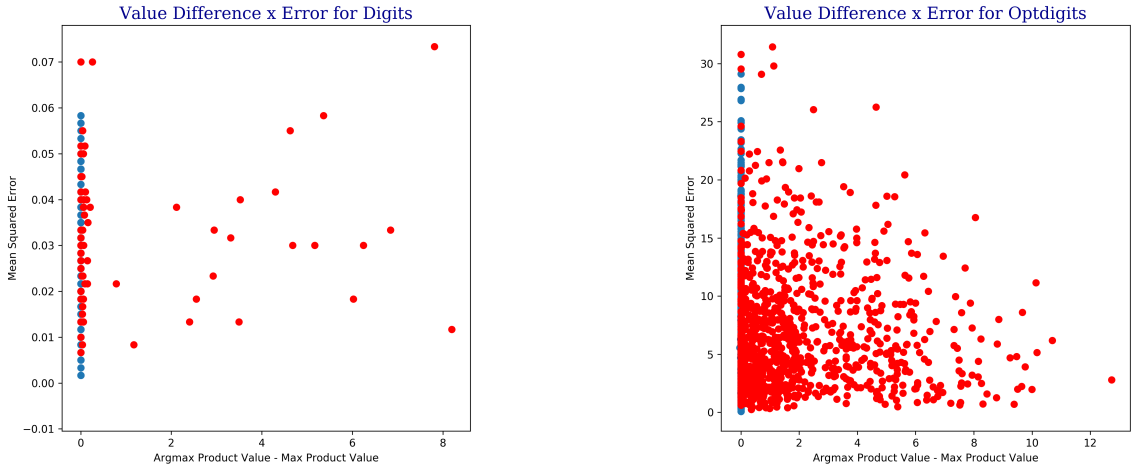


Figure 4.1: *Difference of log-values obtained by Max-Product and Argmax-Product when obtaining auto-completed instances of the digits and optdigits test datasets, and measure of mean square error. When the difference is* 0*, the points on the chart are blue, otherwise they are red.*

[7] Comparing the raw values was problematic because, frequently, some values would be too small (to the order of 10^{-200}) and thus cause problems to the statistical libraries.

Chapter 5

Conclusions and Future Works

5.1 Final Considerations

We have developed a benchmark to test MAP algorithms for SPNs and executed it on a collection of them, obtaining running time and average value comparisons. Additionally, we have provided and tested additional heuristics for variable selection in order to improve the speed of Max Search algorithms in terms of how quick they reach their solutions.

At the introduction, we have listed three questions, and now we can expand on them:

1. Given a limited time-frame, which algorithms perform best in comparison with Max-Product?

   Depending on the SPN, and on the specific MAP problem scenario being tested, some algorithms will invariably perform better. The application depending on this calculation will dictate which algorithm should be used. Max Search is optimal when the SPN is small or its heuristics are able to reach the final values faster, and in these situations it would be preferred.

   However, either the Argmax-Product or a combination of a solution from Argmax-Product with Local Search performed better on average, for small or large SPNs, and thus it should be the ideal choice when Max Search cannot be used.

2. Which heuristics in Max Search find good solutions faster?

   On average, we have seen that choosing either the first variable, or the one with lowest marginal, and using the marginal checking algorithm yielded the best results.

   However, there may not have been enough test instances, and they have not reached the end of the algorithms execution, so the other heuristics could, at the end, obtain the final value (the solution to the MAP problem) faster than the ones we have listed.

3. Does a larger value for a MAP solution provide any gain in a downstream task (such as classification) that depends on that solution?

   We do not have a conclusive answer for this, and thus more experiments are needed. On the digits dataset, the accuracy did not increase, but the opposite happened on the optdigits.

5.2 Suggestions for future work

There is still a space for discovering and implementing an algorithm capable of surpassing Max Search in terms of speed while still obtaining the exact solution to the problem. SPNs still lack a fast algorithm for solving this problem, but in order to develop something that surpasses Max Search, these are the things to be considered:

- For an algorithm that works with bounds, such as the Branch-and-Bound, it needs a better way of obtaining these bounds. Forward Checking on Max Search can obtain as many upper bounds as there are variables and values still to be tested (and is able to prune them at the

same time they are checked). Obtaining better bounds, lower than the marginalized values, would be enough for this. There is still space for experimenting with an algorithm that finds less bounds, but with better quality (higher lower bounds, or lower upper bounds).

- The algorithm was not developed to take advantage of parallelism, and this may be an advantage that could be explored. Splitting the partial assignments set $\boldsymbol{\mathcal{X}}$ into two, and restricting the search on each of these splits to be conducted on different threads could speed up the algorithm, but it clashes with the effect of Forward Checking (that is capable of pruning from the entire $\boldsymbol{\mathcal{X}}$) and therefore may not be ideal.

The effect of dataset differences must be given some attention. On [MJT18] they have shown that developing an algorithm more specialized for the binary SPNs they used works to reach faster executions of the Max Search algorithm. Further, our experiments show that, depending on the SPN, the MAP values will vary much more in proportion to the Max-Product value. SPNs such as CWebkb have achieved larger values on other algorithms to the order of 10^7, while SPNs such as Ad, 10^{-2}. It is fair to conclude that a study on specific datasets, and the way each algorithm behaves on it, may provide more conclusive information for the MAP problem on that specific application than a generalized study using unrelated SPNs.

The technique of changing (removing nodes from) the SPN during an algorithm execution presented on that same article should also be investigated, even if their results were not consistently good, and their conditions for applying not fully described.

Added to the differences of datasets, the differences in learning algorithms that generates these SPNs completes the full scenario of what can change the relevant attributes for these algorithms: Structure and parameters. More information about the effect of these two variables should be useful for the next studies.

Bibliography

[AT15] Mohamed R Amer and Sinisa Todorovic. Sum product networks for activity recognition. *IEEE transactions on pattern analysis and machine intelligence*, 38(4):800–813, 2015. 1

[BDC+15] Jessa Bekker, Jesse Davis, Arthur Choi, Adnan Darwiche, and Guy Van den Broeck. Tractable learning for complex probability queries. *Proceedings of Advances in Neural Information Processing Systems 28 (NIPS)*, 28:2233–2241, 2015. 28

[BVZ98] Yuri Boykov, Olga Veksler, and Ramin Zabih. Markov random fields with efficient approximations. In *Proceedings. 1998 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No. 98CB36231)*, pages 648–655. IEEE, 1998. 2

[CKP+14] Wei-Chen Cheng, Stanley Kok, Hoai Vu Pham, Hai Leong Chieu, and Kian Ming A Chai. Language modeling with sum-product networks. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014. 1

[CMdC17] Diarmaid Conaty, Denis D Mauá, and Cassio P de Campos. Approximation complexity of maximum a posteriori inference in sum-product networks. *arXiv preprint arXiv:1703.06045*, 2017. 2, 3, 9, 13

[Dar03] Adnan Darwiche. A differential approach to inference in bayesian networks. *Journal of the ACM (JACM)*, 50(3):280–305, 2003. 16, 17

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. 28

[GG84] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, PAMI-6(6):721–741, 1984. 2, 8

[GP13] Robert Gens and Domingos Pedro. Learning the structure of sum-product networks. In *International conference on machine learning*, pages 873–880, 2013. 5, 7

[Hoe12] Jesse Hoey. The two-way likelihood ratio (g) test and comparison to two-way chi squared test. *arXiv preprint arXiv:1206.4881*, 2012. 8

[KF09] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009. 25

[KRC+10] Terry Koo, Alexander M Rush, Michael Collins, Tommi Jaakkola, and David Sontag. Dual decomposition for parsing with non-projective head automata. Association for Computational Linguistics, 2010. 2

[LD10] Daniel Lowd and Jesse Davis. Learning markov network structure with decision trees. In *2010 IEEE International Conference on Data Mining*, pages 334–343. IEEE, 2010. 28

[LM11] Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 29–37. JMLR Workshop and Conference Proceedings, 2011. 28

[MJT18] Jun Mei, Yong Jiang, and Kewei Tu. Maximum a posteriori inference in sum-product networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. 2, 12, 15, 16, 18, 19, 20, 22, 25, 27, 28, 29, 30, 36

[MRKA20] Denis Deratani Mauá, Heitor Ribeiro Reis, Gustavo Perez Katague, and Alessandro Antonucci. Two reformulation approaches to maximum-a-posteriori inference in sum-product networks. In *International Conference on Probabilistic Graphical Models*, pages 293–304. PMLR, 2020. 2

[Par02] James D Park. Map complexity results and approximation methods. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, pages 388–396. Morgan Kaufmann Publishers Inc., 2002. 16

[PD11] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690. IEEE, 2011. 1, 2, 5, 6, 12, 16

[Peh15] Robert Peharz. *Foundations of sum-product networks for probabilistic modeling*. PhD thesis, PhD thesis, Medical University of Graz, 2015. 2, 12

[PGD14] Robert Peharz, Robert Gens, and Pedro Domingos. Learning selective sum-product networks. In *LTPM workshop*, 2014. 9, 12

[PGPD16] Robert Peharz, Robert Gens, Franz Pernkopf, and Pedro Domingos. On the latent variable interpretation in sum-product networks. *IEEE transactions on pattern analysis and machine intelligence*, 39(10):2030–2044, 2016. 12

[PSCD20] Iago París, Raquel Sánchez-Cauce, and Francisco Javier Díez. Sum-product networks: A survey. *arXiv preprint arXiv:2004.01167*, 2020. 2

[PTPD15] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. On theoretical properties of sum-product networks. In *Artificial Intelligence and Statistics*, pages 744–752, 2015. 6

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 31

[SC16] B. M. Sguerra and F. G. Cozman. Image classification using sum-product networks for autonomous flight of micro aerial vehicles. In *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 139–144, 2016. 1

[VHD12] Jan Van Haaren and Jesse Davis. Markov network structure learning: A randomized feature generation approach. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, 2012. 28

[WS11] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011. 9

[ZNSS11] Cäcilia Zirn, Mathias Niepert, Heiner Stuckenschmidt, and Michael Strube. Fine-grained sentiment analysis with structural features. In *Proceedings of 5th International Joint Conference on Natural Language Processing*, pages 336–344, 2011. 2

[ZPG16] Han Zhao, Pascal Poupart, and Geoffrey J Gordon. A unified approach for learning the parameters of sum-product networks. In *Advances in neural information processing systems*, pages 433–441, 2016. 12