# Value-Based Potentials: Exploiting Quantitative Information Regularity Patterns in Probabilistic Graphical Models

Manuel Gómez-Olmedo[a], Rafael Cabañas[b], Andrés Cano[a], Serafín Moral[a], Ofelia Paula Retamero[a]

[a] *Department of Computer Science and Artificial Intelligence, University of Granada, Spain*
[b] *Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, Lugano, Switzerland*

**Abstract**

The efficient representation of quantitative information in probabilistic graphical models (PGMs) is a challenge for complex models (i.e. problems with many variables, high degree of dependence between them or many states per variable). In this work, several alternative structures are introduced for facing this problem. All of them are guided by the values and are named *Value-Based Potentials* (VBPs). VBPs try to take advantage of the regularity patterns founded in the data (repetitions of values), regardless of how they appear. These new structures are compared with respect to standard tables or unidimensional arrays representation (1DA) and probability trees (PTs). This last structure seeks for reducing memory space. But this goal can only be achieved if there are context specific independence patterns, that is, repeated values correspond to consecutive indexes. VBPs structures try to overcome this limitation. The objective of this work is to analyze the properties of these alternative representations. To do so, in addition to their theoretical analysis, they are tested to encode quantitative information, to access their content and to make inference with a set of well-known Bayesian networks.

Alternativo

The efficient representation of quantitative information in probabilistic graphical models (PGMs) us a challenge when dealing with complex models (i.e., models with many variables, high degree of dependence between them, or many states per variable). Several new structures are introduced in this work ti face this problem. They are based exclusively on the values and therefore are named *Value-Based Potentials*. VBPs try to take advantage of the presence of repeated values in order to reduce memory requirements. VBPs are compared respect to some common structures as standard tables or unidimensional arrays (1DA) and probability trees (PT). PT seeks for reducing memory space as well, but this is achieved only if repetitions of values correspond to context specific independence patterns (that is, repeated values are related to consecutive indexes or configurations). VBPs try to overcome this limitation. The objective of this work is to analyze VBPs properties. In addition to VBPs theoretical analysis, they

are used to encode the quantitative information of a set of well-known Bayesian networks, measuring access times to their content and the computational time required for performing some inference tasks.

---

## 1. Introduction

*Probabilistic graphical models* (PGMs) [31, 22, 20] are efficient representations for problems under uncertainty. PGMs encode joint probability or utility distributions and are defined by two parts: first, a qualitative component in the form of a graph that represents a set of dependencies among the variables (i.e., nodes) in the domain being modelled; secondly, a quantitative component consisting of a set of functions quantifying such dependencies. In PGMs over discrete domains, such as *Bayesian networks* (BN) [30, 32] or *influence diagrams* [28, 17], these functions are traditionally represented with tables or unidimensional arrays (marginal or conditional probability tables and utility tables, 1DA in general).

The sizes of 1DAs increase exponentially with the number of variables in their domains. This property may limit the ability to represent certain problems with large 1DAs (memory size requirements may be prohibitive). Moreover, even if the model can be encoded with 1DAs problems may arise when performing inference computations. In order to make inference, these 1DAs are managed for computing marginal or conditional probabilities for a certain variable, most probable explanation [11, 21], decision tables in the case of influence diagrams, etc. Some inference algorithms [10, 12, 18, 19, 24, 25, 26, 33, 39, 40] use basic operations on potentials: combination, restriction and marginalization. Combination computes the product of two potentials $_1(\mathbf{X})$ and $_2(\mathbf{Y})$ producing as result a new potential with higher dimension $(\mathbf{X} \lceil \mathbf{Y})$. In this way, 1DAs obtained as intermediate results can be so large that they exceed the memory capacity of the computer.

Therefore, in order to deal with complex problems, it is essential to use efficient representations of the quantitative information of the model. Usually 1DAs encoding probabilities or utilities contain repeated values. For example, some combinations of values are not allowed and are represented with 0's. An efficient representation should take advantage of all these repetitions in order to reduce memory space. Moreover, a useful representation should offer the capability of being approximated with a trade-off between precision and memory space. These features can allow the treatment of more complex models.

The importance of this problem is evidenced by previous attempts to obtain alternative structures to 1DAs. Two examples of alternative approaches are standard and binary probability trees (PTs and BPTs) [3, 7, 34, 14, 6, 4, 5]. These structures can capture context specific independencies [3] and save memory space when repeated values appear under certain circumstances. These representations also have the capability of obtaining approximations through

pruning operation: assuming loss of information, some contiguous values can be substituted by their average value in order to reduce memory space. There are also previous work focused on improving the operations on potentials to alleviate the computational cost when dealing with complex models [2]. Therefore,the existence of efficient structures for storing and managing quantitative information is of interest in all those areas in which PGMs models could be applied, that is, in any problem or system that needs to quantify uncertainty or preferences, [16, 15, 13, 44, 45, 23, 1].

Other data structures exploiting these independencies make possible to compile a full PGM into a more compact representation. This is the case of Algebraic Decision Diagrams (ADDs) [8, 35] or Sequential Decision Diagrams (SDDs) [9, 29]. The former is a graph representation of a function that maps instantiations of Boolean variables to real numbers. A model whose potentials are represented as ADDs can easily be transformed into an arithmetic circuit what minimizes the number of arithmetic operations during the inference. Similarly, a SDD is a full binary tree for representing propositional knowledge bases (a.k.a. Boolean functions). This data structure allows to encode potentials which can be then conjoint to obtain an efficient SDD representation of a full model.

In this work some new alternative representations for potentials are considered. They are based on the properties of the values themselves and not on the contexts in which they appear or the structure of the potential. That is why they have been called *value-based potentials* (VBPs). The paper defines these structures, making a theoretical analysis of their properties and showing concrete examples of how they encode the quantitative information of known and available Bayesian networks in *bnlearn* package repository [38, 37] as well as other networks used in inference *UAI* competitions [42, 43].

The major advantages of VBPs over other related data structures are the following. First, the memory requirements are noticeably reduced for some networks due to a better capacity of exploiting regularity patterns. Secondly, a VBP represents a single potential independently of the rest of parameters of the model. Thanks to that, many inference algorithms could be easily adapted for working with VBPs by simply adapting the basic operations over potentials (e.g., combination and marginalization). This is not the case of ADDs and SDDs, where a complex compilation process is done to obtain a compact representation of the full model.

The structure of the paper is as follows: Section 2 defines some basic concepts and notation and some usual representations for potentials as arrays and trees. Section 3 introduces basic concepts about memory requirement analysis. Section 4 introduces VBPs representations and how to categorize them. Section 5 introduces VBPs alternatives and their properties. Section 6 presents the empirical evaluation performed for testing VBPs capabilities. Finally Section 7 presents conclusions and lines for future research.

## 2. Basics

### *2.1. Definitions and notation*

Let us first introduce the basic notation. Upper-case *roman* letters will be used to denote random variables and lower-case for their values (or states). Thus, if $X_i$ is a random variable, $x_i$ will denote a generic value of $X_i$. The finite set of possible values of $X_i$ is called domain and denoted $\Omega_{X_i}$. For simplicity, we will consider variable values as integers starting with 0 and hence possible assignments will be $X_1 = 0$, $X_1 = 1$, $X_1 = 2$, etc. The cardinality of a variable, denoted $|\Omega_{X_i}|$, is the number of values in its domain. Similarly, we use bold-face upper-case *roman* letters to denote sets of variables, e.g. $\mathbf{X} := \{X_1, X_2, \ldots, X_N\}$ is a set of $N$ variables ($|\mathbf{X}| = N$). The Cartesian product $\bigtimes_{X_i \in \mathbf{X}} \Omega_{X_i}$ is denoted by $\Omega_{\mathbf{X}}$. The elements of $\Omega_{\mathbf{X}}$ are called configurations of $\mathbf{X}$ and will be represented by $\mathbf{x} := \{X_1 = x_1, X_2 = x_2, \ldots, X_N = x_N\}$ or simply $\mathbf{x} := \{x_1, x_2, \ldots, x_N\}$ if the variables are obvious from the context.

Formally, a PGM contains three elements $\langle \mathbf{X}, P, G \rangle$ where $\mathbf{X}$ is the set of variables in the problem with a joint probability distribution $P(\mathbf{X})$ and $G$ is a graph that represents the dependence (and independence) relations between the variables. A PGM allows to represent $P$, which is usually high-dimensional, as a factorisation of lower dimensional local functions. For instance, in case of BNs, these are conditional distributions represented as tables or conditional probability tables (arrays in general, 1DAs). However, we will use the term *potential* which is more general: a potential for $\mathbf{X}$ is a function of $\Omega_{\mathbf{X}}$ over $\mathbb{R}_0^+$. In other words, each configuration $\mathbf{x} \in \Omega_{\mathbf{X}}$ is associated to a real value. Thus, 1DAs or any other function encoding the quantitative information in PGMs can be seen as representations of potentials.

**Example 1.** *Let us consider the variables $X_1$, $X_2$ and $X_3$ with 2, 3 and 2 states respectively. Then $(X_1, X_2, X_3)$ is a potential defined on such variables with the values shown in Figure 1. Note that this potential represents the conditional distribution $P(X_3|X_1, X_2)$.*

The definition of structures for representing potentials requires the introduction of the following concept: *index of a configuration*. It is a unique numeric identifier representing each configuration in a given domain $|\Omega_{\mathbf{X}}|$. We will consider indexes starting with 0 (all the variables take their first value) and ending with $|\Omega_{\mathbf{X}}| - 1$. In the potential given in Figure 1, the index 0 is associated to $\{0, 0, 0\}$, the index 1 to $\{0, 0, 1\}$ and so on until the last one (11) which is associated to $\{1, 2, 1\}$ (indexes are contained in left most column).

It is possible to set a correspondence between indexes and configurations based on the concept of *weight* (a.k.a. *stride* or *step size*). Let us suppose a domain $\mathbf{X} := \{X_1, X_2, \ldots, X_N\}$. Each variable $X_i$ has a weight $w_i$ computed as follows:

$$w_i = \begin{cases} 1 & \text{if i=N} \\ |\Omega_{X_{i+1}}| \; w_{i+1} & \text{otherwise} \end{cases} \tag{1}$$

| $index$ | $X_1$ | $X_2$ | $X_3$ | $(X_1; X_2; X_3)$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0:1 |
| 1 | 0 | 0 | 1 | 0:9 |
| 2 | 0 | 1 | 0 | 0:5 |
| 3 | 0 | 1 | 1 | 0:5 |
| 4 | 0 | 2 | 0 | 0:0 |
| 5 | 0 | 2 | 1 | 1 |
| 6 | 1 | 0 | 0 | 0:8 |
| 7 | 1 | 0 | 1 | 0:2 |
| 8 | 1 | 1 | 0 | 0:2 |
| 9 | 1 | 1 | 1 | 0:8 |
| 10 | 1 | 2 | 0 | 0:9 |
| 11 | 1 | 2 | 1 | 0:1 |

Figure 1: Representation of the potential $(X_1; X_2; X_3)$ as a mapping assigning a numeric value to each con guration

In the potential considered in Example 1 the values of *weights* are: $w_3 = 1; w_2 = 2; w_1 = 6$. Therefore the left most variable is the one with highest *weight*. Therefore, the index of a certain configuration $\mathbf{x} := fx_1; x_2; :::; x_N g$ can be computed with the following expression:

$$index(\mathbf{x}) = \sum_{i=1}^{N} x_i \quad w_i \tag{2}$$

**Example 2.** *Let us consider the potential* $(X_1; X_2; X_3)$ *given in Example 1, with* $w_1 = 6$, $w_2 = 2$ *and* $w_3 = 1$. *Then, the indexes of the configurations in the domain can be computed as follows.*

$$\begin{aligned}
index(f0; 0; 0g) &= 0 \quad 6+0 \quad 2+0 \quad 1 = 0 \\
index(f0; 0; 1g) &= 0 \quad 6+0 \quad 2+1 \quad 1 = 1 \\
index(f0; 1; 0g) &= 0 \quad 6+1 \quad 2+0 \quad 1 = 2 \\
&:::::::::::::::::::::::::::::::::::::::::::::::::: \\
index(f1; 2; 1g) &= 1 \quad 6+2 \quad 2+1 \quad 1 = 11
\end{aligned}$$

Given a certain index $k$, its *associated configuration* is denoted $\mathbf{x}^{(k)}$ and satisfies that $index(\mathbf{x}^{(k)}) = k$. Given a certain index $k$, the value assigned to each variable $X_i$ can be computed as:

$$x_i = (k == w_i)\%j\Omega_{X_i}j \tag{3}$$

where == denotes integer division and % the module of the division. This operation is necessary to allow a fast conversion between configurations and indexes.

Note that the association between indexes and configurations requires an order of the variables in the domain. Though any order is valid, we will consider by default the order in which variables are written. E.g., in the potential $(X; Y; Z)$, the first variable would be $X$. In addition, here we consider the first variable to have the highest weight (that is, following *row-major order*). However, the opposite approach could be also considered: the first variable has weight 1 whereas the last one has the highest (*column-major order*).

*2.2. Representation for potentials*

A potential is basically a multidimensional object with a dimension per variable. Thus, a standard method for storing it is to flatten it into a single 1D-array (1DA) in computer memory [20]. Thus, potential defined on a set of $N$ variables, can be represented by $A$ array as follows.

$$A := \left[ (0; 0; \ldots; 0); \ (0; 0; \ldots; 1); \ldots; \ (|\Omega_{X_1}| - 1; |\Omega_{X_2}| - 1; \ldots; |\Omega_{X_N}| - 1) \right] \quad (4)$$

The main advantage of the representation with 1DA consists in the fact that the position in which each value is stored coincides with the index of the corresponding configuration. This makes very efficient the access with indexes. The size of a 1DA, denoted $size(A)$, is the number of entries and is equal to the number of configurations in the potential.

**Example 3.** *The potential* $(X_1; X_2; X_3)$ *given in Example 1, can be represented as the following 1DA with 12 entries (see Figure 2).*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.1 | 0.9 | 0.5 | 0.5 | 0.0 | 1.0 | 0.8 | 0.2 | 0.2 | 0.8 | 0.9 | 0.1 |

Figure 2: $(X_1; X_2; X_3)$ as a 1DA

Some limitations of this representation are: the access to 1DA with configurations would require a transformation into indexes using Eq. 2; the coincidence between indexes and storage positions makes it necessary to store all values. Therefore, the storage of repeated values in consecutive positions cannot be avoided.

*Probability trees* (PTs) are an alternative structure that have been used for storing and operating with potentials in PGMs [7, 34, 6], both accurately and approximately. The tree storing a potential , $T$ , is a directed tree with two types of nodes: internal *nodes* that represent variables and *leaf* nodes representing the values of the potential. The internal nodes have outgoing arcs (one for each state of the associated variable). The size of a tree $T$, $size(T)$, is defined as the number of nodes it contains.

**Example 4.** *The same potential given in the previous example is presented in Figure 3 as a PT. This PT has a size of 21 nodes (*12 *leaves and* 9 *internal nodes).*

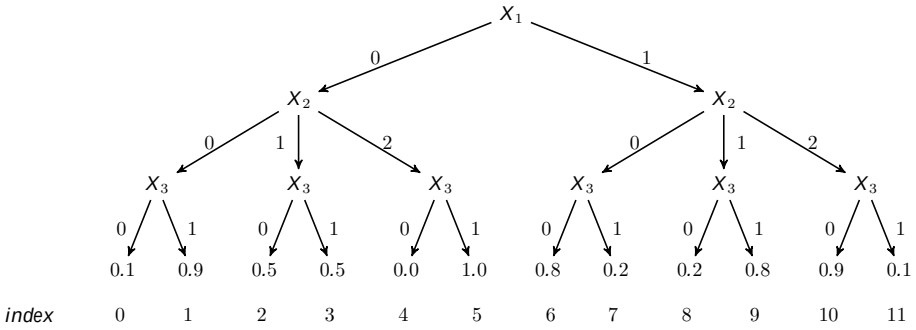


Figure 3: $(X_1; X_2; X_3)$ as PT

With PTs, the most efficient way of access is via configuration: the tree must be traversed from root to leaves selecting the corresponding branch for each variable until reaching a leaf node with its value. In addition, PTs can take advantage of context-specific independencies [3] so that many identical values can be grouped into a single one offering a compact storage. The operation of collapsing identical values is called *pruning*.

**Example 5.** *The potential in previous examples presents a context-specific independence that allows a reduction of its size: the value for* $X_1 = 0; X_2 = 1$ *is* 0:5 *regardless of the value of* $X_3$*. If the pruning is done, the result is a PPT (pruned PT) of size* 19 *shown in Figure 4.*

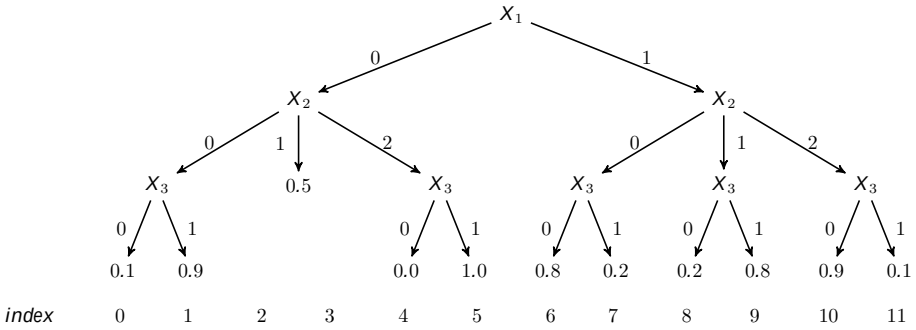


Figure 4: $(X_1; X_2; X_3)$ as PPT

However PTs can not take advantage of another repetition patterns. Let us consider the tree presented in Figure 3. The values for indexes 7 and 8 are 0:2

and consecutive as well, but they can not be pruned because they correspond to configurations varying both in $X_2$ and $X_3$ values.

A variant of PTs, called binary trees (BPTs) [6], can divide the domain of each variable into two subsets of states. This would allow finer grain context independencies to be exploited with respect to regular trees. For example, in the PT (left part of Figure 5), the values 0.4 in $c$ configuration (left subtree) can not be pruned. However, with BPTs grouping states 0 and 2 of $X_k$ would allow the value 0.4 to be represented with a single leaf node (as showed in the BPT of the right part of Figure 5). This reduces memory space for $c$ context, although it would require more nodes for the right subtree ($c'$ context). For this reason, only PTs and PPTs are considered for comparison in this work.

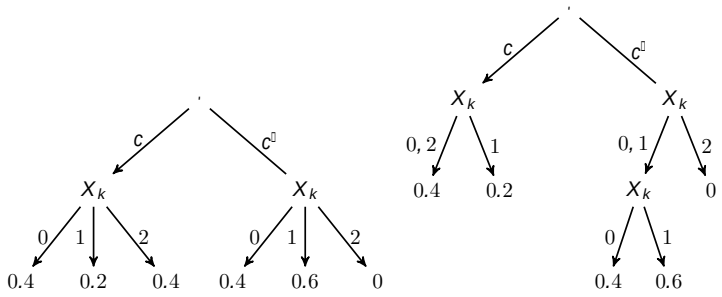


Figure 5: Binary tree representation

## 3. Memory requirements analysis

Even though the size of a representation gives an idea of its complexity, a more accurate analysis of its memory space requirements is needed: any representation will consume additional elements (e.g., pointers, meta-information, etc.) and each of them use different data types. For this analysis we will consider a potential  defined over a set of $N$ variables $\mathbf{X} := \{X_1, X_2, \ldots, X_N\}$. Additionally, the following notation related to the different memory sizes is defined.

- $s_f$ will be the memory size required for storing a float value.
- $s_i$ is the memory size for storing an index denoting a concrete configuration of $\Omega_{\mathbf{X}}$.
- $s_r$ denotes the size of a reference to an object.
- $s_v$ represents the memory space required for storing the information about a variable: name, cardinality and state names. As this depends on the names of variables and states, we will assume a fix value for all of them (in fact, this memory space will be negligible respect to the whole memory

used for storing a potential). Moreover we can define a standard way of coding variables with numerical identifiers and employ the same idea for their states.

- $s_s$ denotes the memory size of the data structure used for storing information (array, list, set or dictionary). Specific notation will be used for this term afterwards: $s_{arr}$, $s_{list}$, $s_{set}$ and $s_{dict}$ respectively.

As it was mentioned before, the representation by means of 1D-array (1DA) has an important advantage: the values for configurations are stored consecutively. That is, the value in position $k$ corresponds to $k$-th configuration. In this way it is not necessary to store information about indexes. Therefore, its codification supposes an amount of memory given by the number of values to store, the size of the array data structure and the memory required for its variables:

**Proposition 1** (Memory space for an array representing a potential)**.** *Let $A$ be a 1DA representing $(\mathbf{X})$, $|\mathbf{X}| = N$. Then the amount of required memory is given by the following expression.*

$$memory(A) = N \cdot s_v + m \cdot s_f + s_s \tag{5}$$

*where $m = |\Omega_{\mathbf{X}}|$ is the number of entries in the array.*

**Example 6.** *From the previous examples, consider the potential $(X_1; X_2; X_3)$ and its codification as a 1D-array given in Example 3. Then the estimation of the memory size is:*

$$memory(A) = 3s_v + 12s_f + s_s \tag{6}$$

The tree representation (PT and PPT) is usually less efficient in terms of memory requirements as the full structure of the tree must be stored. Thus, the amount of memory depends on the number of internal nodes, denoted $n_I$, and the number of leaves $n_L$. Note that it holds that $n_I + n_L = size(T)$. Internal nodes store links (or references) to sub-trees (each for a state of the corresponding variable). These links are stored into an array. Then, it is relevant to consider the number of outgoing arcs: $n_I^{(j)}$ denotes the total number of internal nodes for variables with $j$ states.

**Proposition 2** (Memory space for a tree representing a potential)**.** *Let $T$ be a tree representing $(\mathbf{X})$, $|\mathbf{X}| = N$. Then the amount of required memory is estimated as follows.*

$$memory(T) = N \cdot s_v + n_L \cdot s_f + \sum_{j=2}^{K} n_I^{(j)} \cdot (s_v + s_s + j \cdot s_r) \tag{7}$$

*where $K = \max\{|\Omega_{X_i}| : X_i \in \mathbf{X}\}$ is the maximal cardinality among the variables in the potential.*

In case of a non-pruned tree, the number of leaf nodes will be equal to the number of configurations in the potential. As a consequence, the first two terms in Equations(5) and (7) are equal. Thus, the main factors in the size of trees are the data structure employed for storing links to subtrees and the repetition of variables information.

**Example 7.** *Let $T$ be the PT from Example 4 (Figure 3) containing* 7 *internal nodes for binary variables and* 2 *internal nodes for the ternary ones and* 12 *leaves. Similarly, let $T'$ be the PPT from Example 5 (Figure 4) with* 6 *internal nodes for binary variables,* 2 *internal nodes for ternary variables and* 11 *leaf nodes. Then, their memory cost can be computed with the following expressions:*

$$memory(T) = 3s_v + 12s_f + 7(s_v + s_s + 2s_r) + 2(s_v + s_s + 3s_r) \quad (8)$$

$$memory(T') = 3s_v + 11s_f + 6(s_v + s_s + 2s_r) + 2(s_v + s_s + 3s_r) \quad (9)$$

In the previous example, the pruning operation has reduced the number of internal nodes as well as the number of leaves, but anyway the cost is higher to the size of the table representation. A large number of repeated values are usually required for obtaining memory savings when using PTs.

For a concrete analysis, the following sizes for data types are assumed (the real sizes can depend on the machine architecture; in fact, real sizes are not relevant as long as the same sizes are used for all the comparisons):

- long: 4 bytes (for indexes).
- float: 8 bytes (for real values).
- pointer or reference: 8 bytes (memory addresses).
- variable: 50 bytes (this includes the space for storing name, state names, etc). That is, $s_v = 50$.
- the concrete value of $s_s$ will depend on the concrete data structure employed:
  - array and list ($s_{arr}$ and $s_{list}$ respectively): 16 bytes.
  - set ($s_{set}$): 32 bytes.
  - dictionary ($s_{dict}$): 64 bytes.

Using these sizes, the estimations of memory for the representations $A$, $T$ and $T'$ are 262, 1000 and 910 respectively.

## 4. Value-based representations

### *4.1. Motivation*

At this point, it is clear the necessity of having efficient mechanisms for handling quantitative information for the tasks of representation, inference and learning with PGMs. We have already considered that PTs allow to capture some patterns of repetition in very specific situations. The underlying idea in VBPs (Value-Based Potentials) is to let the representation process be guided by the values themselves and saving space from repetitions as much as possible. Therefore, the objectives of VBPs are:

- being able to take advantage of all repetition patterns, regardless of the order in which they appear.
- to allow efficient access to values and provide the necessary operations to perform inference tasks. In this work, basic implementations for combination and marginalization operations are provided to obtain a starting estimate of VBPs behavior when using inference algorithms.
- to facilitate the approximation task and the parallel management. These capabilities are not explored in this work but it is important to consider them for reaching a good design including these possibilities (to be explored as future research).

### *4.2. Alternatives categorization*

The proposed alternatives can be classified in two groups depending on how to make the queries:

- *driven by values* approaches, based on the use of dictionaries in which the keys will be values. Two representations belonging to this group are presented: *Value-Driven with Grains* (VDG) and *Value-Driven with Indexes* (VDI).
- *driven by indices* alternatives where keys are indexes. Into this group we present *Index-Driven with Indexes* (IDP) and *Index-Driven with Map* (IDM). Both alternatives use an array for values ($V$). IDP also uses a second array ($L$) that stores the indexes and the information needed to link indexes and values. IDM uses a map ($M$) with indexes as keys and the information to link with $V$ as values.

The particular features for all of them will be presented below. However, a common capability is outlined here. It must be clear that these representations require a search for managing the information. This can be exploited defining a default value to return when the search fails. This default value can be set to 0.0 or fixed after analyzing the values of the potential. In this case it would help to select as default value the most repeated one in order to reduce memory although it requires more computation time. This work considers the first alternative.

## 5. VBPs description

### *5.1. VDG: value-driven with grains*

Identical values in potentials will often appear in configurations with consecutive indexes. Consider for instance the potential given in Example 3, in which the value 0.5 appears in positions 2 and 3. Similar situation happens with value 0.2 in positions 7 and 8. As a consequence, a compact way of defining sets of configurations associated to the same value could be by means of intervals (i.e., grains). Formally, a grain can be defined as follows.

**Definition 1** (Grain)**.** *Let* $\mathbf{X}$ *be a set of variables and* $i$ *and* $j$ *indexes of valid configurations on* $\Omega_{\mathbf{X}}$*, with* $i \quad j$*. A grain* $g(i; j)$ *defines a sequence of consecutive indexes* $i; i+1; \ldots j$*. Grains will be used for representing sequences of repeated values in VBPs.*

In a VDG enconding a potential, each non-zero value will be associated with one or more grains defining all the indexes for which the potential takes this value. More formally, a VDG can be defined as follows.

**Definition 2** (Value-driven with grains)**.** *Let* $\quad$ *be a potential defined over* $\mathbf{X}$*. Then a value-driven with grains for* $\quad$*,* $VDG$ *, is a dictionary* $D$ *in which entries are defined as* $< v; L_v >$*, where* $v \; 2 \quad$ *(key) is a non-default value and* $L_v$ *is a list of grains that store its associated indexes. Therefore for each grain* $g(i; j) \; 2 \; L_v$ *all its indexes correspond to* $v$ *(that is, for all* $k = i; i+1; \ldots; j$ *then* $(\mathbf{x}_k) = v$*).*

**Example 8.** *The potential* $(X_1; X_2; X_3)$ *used in previous examples and presented in Figure 1 will be represented with VDG as showed in Figure 6.*

*The outermost rectangle with rounded corners represents the dictionary. The circular nodes indicate entry keys. Associated with each key, on the right, it is represented the related list of grains. It can be seen in Figure 6 that each value is stored only once. Some values appear only once in the potential. This is the case of* 1.0*, with* 5 *as starting and ending index in the grain. The rest of values appear several times. For example,* 0.1 *is the value for indexes* 0 *and* 11*. As these indexes are not consecutive, they must be stored in two different grains. Values* 0.2 *and* 0.5 *appear in consecutive indexes and their corresponding grains capture the sequences of repetitions.*

**Algorithm 1** (Access to index in VDG)**.** *Given* $VDG$ *, the algorithm for getting the value corresponding to a given index* $I$ *is described in Algorithm 1.*

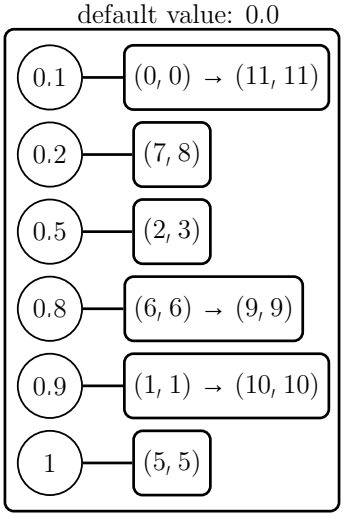


Figure 6: $(X_1; X_2; X_3)$ as VDG

**Algorithm 1** Access to $I$ index in $VDG$

```
1:  function access(VDG ; I)
2:      result = 0:0                          . sets the default value to result
3:      for each v (key) in D key set do      . loop on dictionary entries
4:          L_v   D(v)                        . list of grains for v
5:          for each g (grain) in L_v do
6:              if I 2 g then                 . I is included in g
7:                  result = v and stop iteration
8:              end if
9:          end for
10:     end for
11:     return result
12: end function
```

Assume 6 is the target index. The search progresses through dictionary entries until reaching key 0:8. The internal loop (line 5) iterates on the grains for this value. As the first one contains 6, then 0:8 would be the result of the search. When the search fails and the index is not found, then 0:0 (default value) is returned. This would be the case for index 4.

**Proposition 3** (Memory space for a VDG representing a potential)**.** *Let $VDG$ be the representation of $(\mathbf{X})$, with $j\mathbf{X}j = N$. Let assume $d$ represents the number of different values in the potential (discarding the default value). The number of grains for each value are denoted by $g_1 ::: g_d$. Then the amount of*

*memory required is estimated as follows.*

$$memory(VDG\ ) = N\ \ s_v + s_f + s_{dict} + d\ \ (s_f + s_{list}) + \sum_{j=1}^{X} 2\ \ g_j\ \ s_i \quad (10)$$

The terms in Equation (10) consider sizes for: variables; storage for default value; dictionary; values and lists; and grains with 2 indexes per grain. The number of grains for each value will depend of the sequences of repetitions. It will be lower as long as the sequences are longer. Therefore, the critical point in this representation is the required number of grains, given by $\sum_{j=1}^{d} g_j$.

**Example 9.** *Let $VDG$ be the VDG from Example 8 with 6 different values to store in the dictionary and 0.0 as default value. Therefore, the dictionary stores 6 entries. The sequences of repetitions requires 9 grains. Then, the memory cost can be computed with the following expression:*

$$memory(VDG\ ) = 3s_v + s_f + s_{dict} + 6\ \ (s_f + s_{list}) + 9\ \ 2\ \ s_i \quad (11)$$

*Using the concrete memory sizes described in Section 3 the complete amount of memory is 438 bytes (sizes of 1DA, PT and PPT are 262, 1000 and 910 bytes respectively).*

### *5.2. VDI: value-driven with indexes*

Even though the previous structure with grains is a compact representation for potentials, it could encode unnecessary information when repeated values are not in consecutive positions. This is the case for 0.1 in Figure 6. Its entry includes 2 grains of length 1: $(0; 0)$ and $(11; 11)$. One way to avoid this repetition would be to associate values with the complete list of indexes in which they appear. In this example, 0.1 value will be related to the list $(0\ \ !\ \ 11)$. This alternative will be advantageous if the sequence of values of a potential does not contain large series of repetitions. Having this idea in mind, the following representation can be defined.

**Definition 3** (Value-driven with indexes)**.** *Let be a potential defined over $\mathbf{X}$, then a value-driven with indexes for , $VDI$ , is a dictionary $D$ in which each entry $< v; L_v >$ contains a value (as key) and a list of indexes $L_v$, such that $(\mathbf{x}_i) = v$ for each $l\ 2\ L_v$.*

**Example 10.** *The potential $(X_1; X_2; X_3)$ used before and described in Figure 1 will be represented as VDI as showed in Figure 7.*

*The outermost rectangle represents the dictionary: keys of entries are drawn as circles. Keys give access to lists of indexes (rectangles of rounded corners).*

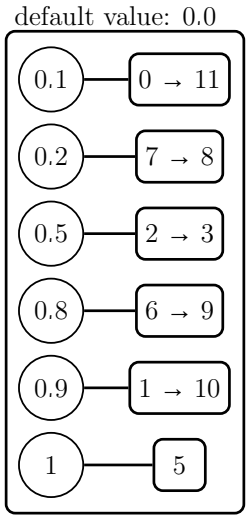


Figure 7: $(X_1; X_2; X_3)$ as VDI

**Algorithm 2** (Access to index in VDI)**.** *Given $VDI$ , the algorithm for getting the value corresponding to a given index $I$ is described in Algorithm 2.*

**Algorithm 2** Access to $I$ index in $VDI$

```
1:  function access(VDI ; I)
2:      result = 0:0                          . sets the default value to result
3:      for each v (key) in D key set do      . loop on dictionary entries
4:          L_v   D(v)                        . list of indexes for v
5:          for each p in L_V do              . loop on list of indexes
6:              if p == I then                . I is included in L_v
7:                  result = v and stop iteration
8:              end if
9:          end for
10:     end for
11:     return result
12: end function
```

Assume 6 is the target index. The search progress through dictionary entries until reaching key 0:8. The internal loop (line 5) iterates on the list. As it contains 6, then 0:8 would be the result. A failed search produces 0:0 as a result. This is the case for index 4.

**Proposition 4** (Memory space for a VDI representing a potential)**.** *Let $VDI$ be the representation of $(\mathbf{X})$ with $j\mathbf{X}j = N$. Let assume $d$ represents the number of different values in the potential (discarding the default value). The number*

*of indexes for each value are denoted by $i_1 ::: i_d$. Then the required amount of memory is estimated as follows.*

$$memory(V\,D\,I\ ) = N\ \ s_v + s_f + s_{dict} + d\ \ (s_f + s_{list}) + \sum_{j=1}^{\not{X}} i_j\ \ s_i \qquad (12)$$

The terms of Equation 12 consider sizes for: variables, default value, dictionary, values, lists and indexes. In VDI, the main source of savings comes from avoiding the storage of repeated values, since the indexes in which the significant values (non zero) appear must be explicitly stored.

**Example 11.** *Let $V\,D\,I$ be the VDI from Example 10 with* 6 *different values to store in the dictionary and* 0:0 *as default value. Therefore, the dictionary stores* 6 *pairs and* 6 *lists with* 11 *indexes. Then, the memory cost can be computed with the following expression:*

$$memory(V\,D\,I\ ) = 3s_v + s_f + s_{dict} + 6\ \ (s_f + s_{list}) + 11\ \ s_i \qquad (13)$$

*Using the memory sizes described in Section 3 the complete amount of memory is 410 bytes (a bit lower than VDG).*

### *5.3.* IDP: index-driven with pairs

The problem with the access to value-driven structures is the need to perform a double iteration. The search for a target index, $I$, requires iterating over the list of entries and over the associated lists (of grains or indexes). To avoid this double iteration and make the search more efficient, new structures are introduced in which the search is based on the indexes themselves. This is the case of IDP and IDM.

**Definition 4** (Index-driven with pairs)**.** *Let be a potential defined over* $\mathbf{X}$*. Then a structure index-driven with pairs (IDP) representing , $IDP$ , is a pair of arrays: $V$ and $L$. Non-repeated values in (excluding* 0:0 *as default value) are stored in $V := fv_0; v_1; ::: ; v_{d\ 1}g$. Let $nd$ represents the set of indexes storing non-default values. The array $L$ is defined as follows.*

$$L := f(i; j) :\ \ (\mathbf{x}_i) = v_j;\ \ i\ 2\ nd\ g \qquad (14)$$

That is, IDP is based on two components. First, an array storing the values (without repetitions, as before, and excluding 0:0 as default value). Secondly, an array of pairs (index in potential - index in array of values). The second index of the pair keeps the relation between indexes and values.

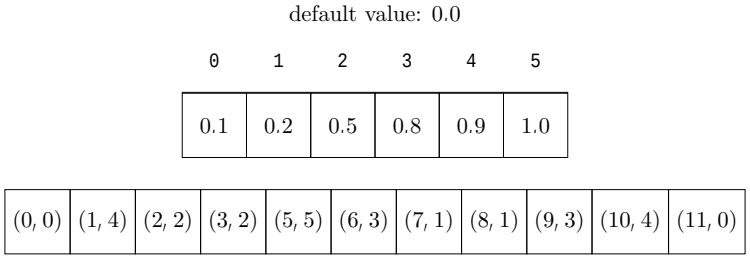


Figure 8: $(X_1; X_2; X_3)$ as IDP

**Example 12.** *The representation as IDP of the potential* $(X_1; X_2; X_3)$ *presented in Figure 1 is shown in Figure 8.*

*As explained before, IDP uses two coherent arrays.* $V$ *stores non repeated values (except the default value).* $L$ *contains pairs of indexes. Let us consider value* $0.5$ *in* $(X_1; X_2; X_3)$*, presented in indexes* $2$ *and* $3$*. Then, the array of pairs,* $L$ *(bottom part of Figure 8), requires two pairs for this relation:* $(2; 2)$ *and* $(3; 2)$*. Both indicate that potential indexes* $2$ *and* $3$ *stores the value in* $V(2)$*.*

**Algorithm 3** (Access to index in IDP)**.** *Given a* $IDP$ *, the algorithm for getting the value corresponding to a given index* $I$ *is described in Algorithm 3.*

**Algorithm 3** Access to $I$ index in $IDP$

```
1: function access(IDP ; I)
2:     result = 0.0                          . sets the default value to result
3:     for each pair t = (i ; i_V) in L do          . loop on L array pairs
4:         if i == I then                  . I is found; value stored in V(i_V)
5:             result = V(i_V) and stop iteration
6:         end if
7:     end for
8:     return result
9: end function
```

Assume 6 is the target index. The search progress through $L$ until reaching 6-th position (pair $(6; 3)$). The value to return is stored in $V(3) = 0.8$. If index 4 is searched, then $0.0$ will be returned (this index is not present in $L$).

**Proposition 5** (Memory space for an IDP representing a potential)**.** *Let* $IDP$ *be the structure representing* $(\mathbf{X}); |\mathbf{X}| = N$*. Let assume* $d$ *represents the number of different values in the potential (discarding the default value). The number of indexes corresponding to non-default value is* $p$*. Then the amount of memory is estimated as follows.*

$$memory(IDP) = N \cdot s_v + s_f + 2 \cdot s_{arr} + d \cdot s_f + 2 \cdot s_i \cdot p \qquad (15)$$

The terms in Equation 15 considers the sizes for: variables, default value, both arrays, values and pairs of indexes. This representation tries to use simple structures and favours the direct search on indices rather than on values.

**Example 13.** *Let $IDP$ be the IDP from Example 12 with 6 different values to store and 0.0 as default value. Therefore, $V$ stores 6 values and $L$ 11 pairs. Then, the memory cost can be computed with the following expression:*

$$memory(IDP) = 3s_v + s_f + 2 \; s_{arr} + 6 \; s_f + 11 \; 2 \; s_i \tag{16}$$

*Using the concrete memory sizes described in Section 3 the complete amount of memory is 326 bytes.*

### *5.4. IDM: index-driven with map*

This structure aims to take a further step in the idea of facilitating access to the structure, using a dictionary in which the keys are the indexes. Below is the definition of this new index-driven alternative.

**Definition 5** (Index-driven with map)**.** *Let be a potential defined over $\mathbf{X}$, then a structure index-driven with map (IDM) representing , $IDM$ consists of a dictionary $D$ and an array $V$. Non-repeated values in are stored in $V := \{v_0; v_1; \ldots; v_{d\ 1}\}$. $D$ entries $< i; j >$ are formed by indexes (keys) an $V$ indexes. Let $nd$ represents the set of indexes storing non-default values. Given an entry $< i; j >$, then: $(\mathbf{x}_i) = v_j$ and $i \ 2 \ nd$ .*

**Example 14.** *The representation as IDM of the potential $(X_1; X_2; X_3)$ presented in Figure 1 is shown in Figure 9.*

*The dictionary $D$ is represented in the left part of Figure 9 and the array of values $V$ in the right one. Keys in $D$ are drawn as circles and give access to $V$ indexes (boxes linked to keys).*

**Algorithm 4** (Access to index in IDM)**.** *Given $IDM$ , the algorithm for getting the value corresponding to a given index $I$ is described in Algorithm 4.*

**Algorithm 4** Access to $I$ index in $IDM$

```
1: function access(IDM ; I)
2:     result = 0.0                          . sets the default value to result
3:     entry(< I; j >)    D(I)               . search dictionary for I
4:     if entry != null then                 . dictionary contains I as key
5:         result    V(j)
6:     end if
7:     return result
8: end function
```

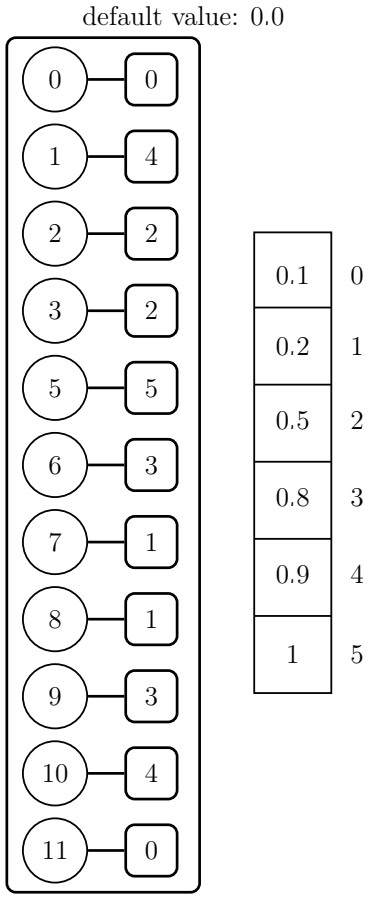


Figure 9: $(X_1; X_2; X_3)$ as IDM

Assume 6 is the target index. As this is a valid key, entry $< 6; 3 >$ is retrieved. The value to return is stored in $V(3) = 0.8$. If index 4 is searched, then 0.0 will be returned (this index is not present in $D$).

**Proposition 6** (Memory space for an IDM representing a potential)**.** *Let $IDM$ be the structure representing $(\mathbf{X})$; $|\mathbf{X}| = N$. Let assume $d$ represents the number of different values in the potential (discarding the default value) and $p$ the number of indexes storing non-default values. Then the amount of memory is estimated as follows.*

$$memory(IDM) = N \cdot s_v + s_f + s_{dict} + s_{arr} + d \cdot s_f + 2 \cdot p \cdot s_i \qquad (17)$$

The terms of the Equation (17) represents sizes for: variables, default value, dictionary, arrays of values, different values and indexes in dictionary entries.

**Example 15.** *Let $IDM$ be the VBP from Example 14 with* 6 *different values to store and* 0:0 *as default value. Then, the array of values has* 6 *elements. The dictionary contains* 11 *entries. Memory cost can be computed with the following expression:*

$$memory(IDM\ ) = 3s_v + s_f + s_{dict} + s_{arr} + 6\ \ s_f + 2\ \ 11\ \ s_i \qquad (18)$$

*Considering the concrete values as specified in Section 3 the complete amount of memory is 374 bytes.*

*5.5. Example of a extreme case*

Below it is included an example of the use of the representation structures under consideration for a potential with extreme features: only three different values (0, 0:5 and 1) and many repetitions of the default value (around 70% of the indexes are assigned to 0). The potential has 1024 possible values, with 5 variables in its domain, each with 4 states. The values are randomly generate. We have considered 10 random potentials with these features in order to get a reliable idea of the behavior of the representations. Results are presented in Table 1. $1DA$ and $PT$ representations do not depend on the concrete values and requires memory sizes of 8574 and 53816 respectively.

| iteration | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $PPT$ | 39090 | 38332 | 37658 | 38396 | 40946 | 38910 | 42610 | 38120 | 42706 | 38544 |
| $VDG$ | 1822 | 1814 | 1670 | 1774 | 1934 | 1854 | 1990 | 1734 | 2030 | 1694 |
| $VDI$ | 1206 | 1210 | 1110 | 1190 | 1310 | 1218 | 1346 | 1166 | 1374 | 1194 |
| $IPD$ | 1862 | 1870 | 1670 | 1830 | 2070 | 1886 | 2142 | 1782 | 2198 | 1838 |
| $IDM$ | 1910 | 1918 | 1718 | 1878 | 2118 | 1934 | 2190 | 1830 | 2246 | 1886 |
| values | 180 | 166 | 156 | 176 | 206 | 183 | 215 | 170 | 222 | 177 |

Table 1: Memory sizes for random potential representations

In all the random potentials the number of non-zero values ranges from 156 to 222 (last row in Table 1) and the longest sequence of repeated values is 2. Although better results could be obtained with longest sequences of repeated values (at least with VDG representation), the results show that the savings in memory consumption with respect $1DA$, $PT$ and $PPT$ are noticeable.

## 6. Empirical evaluation

Two sets of Bayesian networks are used for evaluating VBPs capabilities with respect to previous representations for potentials in PGMs: conditional probability tables (1DA) and trees (PT and PPT). The first set is taken from the *bnlearn* repository ([38, 37]) and the second one from *UAI* competitions ([42, 43]). The quantitative information of these models is represented with the structures previously defined (VDG, VDI, IDP and IDM). Experiments are organized in three different blocks: comparison of memory sizes, access times

and computation time of posterior distributions using the Variable Elimination (VE) algorithm [41, 46, 12].

The representations compared in experiments are:

- 1DA, PT, PPT, VDG, VDI, IDP and IDM for memory sizes and access times comparisons on *bnlearn* and *UAI* networks.
- 1DA, PT, VDI and IDM for posterior computations with *UAI* networks.

### *6.1. Bayesian networks features*

Some basic information about the Bayesian networks employed is gathered in Table 2 and Table 3: name of network; number of nodes, number of arcs; minimum, average and minimum number of states of variables; and complete number of parameters for quantifying networks uncertainty. Networks are ordered according the number of parameters.

| network | nodes | arcs | min. st. | avg. st. | max. st. | parameters |
|---|---|---|---|---|---|---|
| cancer | 5 | 4 | 2 | 2 | 2 | 20 |
| asia | 8 | 8 | 2 | 2 | 2 | 36 |
| survey | 6 | 6 | 2 | 2.33 | 3 | 37 |
| sachs | 11 | 17 | 3 | 3 | 3 | 267 |
| child | 20 | 25 | 2 | 3 | 6 | 344 |
| alarm | 37 | 46 | 2 | 2.83 | 4 | 752 |
| win95pts | 76 | 112 | 2 | 2 | 2 | 1148 |
| insurance | 27 | 52 | 2 | 3.29 | 5 | 1419 |
| hepar2 | 70 | 123 | 2 | 2.31 | 4 | 2139 |
| andes | 223 | 338 | 2 | 2 | 2 | 2314 |
| hail nder | 56 | 66 | 2 | 3.98 | 11 | 3741 |
| pigs | 441 | 592 | 3 | 3 | 3 | 8427 |
| water | 32 | 66 | 3 | 3.625 | 4 | 13484 |
| munin1 | 186 | 273 | 2 | 5.33 | 21 | 19226 |
| link | 724 | 1125 | 2 | 2.53 | 4 | 20502 |
| munin2 | 1003 | 1244 | 2 | 5.36 | 21 | 83920 |
| munin3 | 1041 | 1306 | 2 | 5.38 | 21 | 85615 |
| path nder | 109 | 195 | 2 | 4.11 | 63 | 97851 |
| munin4 | 1038 | 1388 | 2 | 5.44 | 21 | 97943 |
| munin | 1041 | 1397 | 2 | 5.43 | 21 | 98423 |
| barley | 48 | 84 | 2 | 8.77 | 67 | 130180 |
| diabetes | 413 | 602 | 3 | 11.34 | 21 | 461069 |
| mildew | 35 | 46 | 3 | 17.6 | 100 | 547158 |

Table 2: *bnlearn* Bayesian networks features

Observe that networks in the *UAI* set require more parameters than those from *bnlearn*.

| network | nodes | arcs | min. st. | avg. st. | max. st. | parameters |
|---|---|---|---|---|---|---|
| BN_76 | 2155 | 3686 | 2 | 7.01 | 36 | 627298 |
| BN_87 | 422 | 867 | 2 | 2 | 2 | 933776 |
| BN_29 | 24 | 30 | 10 | 10 | 10 | 1132080 |
| BN_125 | 50 | 375 | 2 | 2 | 2 | 2117680 |
| BN_115 | 50 | 375 | 2 | 2 | 2 | 2285616 |
| BN_119 | 50 | 375 | 2 | 2 | 2 | 2410544 |
| BN_121 | 50 | 375 | 2 | 2 | 2 | 2564144 |
| BN_123 | 50 | 375 | 2 | 2 | 2 | 3249200 |
| BN_27 | 3025 | 7040 | 3 | 6 | 10 | 3698565 |
| BN_117 | 50 | 375 | 2 | 2 | 2 | 4003888 |
| BN_22 | 2425 | 4239 | 2 | 18.743 | 91 | 4073904 |
| BN_111 | 50 | 375 | 2 | 2 | 2 | 4238512 |
| BN_109 | 50 | 375 | 2 | 2 | 2 | 4581936 |
| BN_113 | 50 | 375 | 2 | 2 | 2 | 4669488 |
| BN_20 | 2483 | 5272 | 2 | 18.92 | 91 | 5009364 |
| BN_107 | 50 | 375 | 2 | 2 | 2 | 5154864 |
| BN_105 | 50 | 375 | 2 | 2 | 2 | 6431792 |

Table 3: *UAI competition* Bayesian networks features

### *6.2. Memory sizes comparisons*

In order to compare the memory sizes required for each representation with respect to 1DA, PT and PPT, we proceed to convert all the potentials to VDG, VDI, IDP and IDM. The memory size used by 1DA is taken as a reference and does not appear in the table. Given a certain network, let $m_{1DA}$ be the memory space for 1DA representation and $m_{rep}$ the memory size for another one. Then the value $s$ included in table cells and representing the gain (or loose) for $rep$ is computed as:

$$s = \frac{m_{rep} \quad 100}{m_{1DA}} \quad 100 \tag{19}$$

Thus, a negative value for $s$ indicates that $rep$ requires less memory space than 1DA. On the contrary, positive values indicate higher memory consumption.

#### *6.2.1. Memory sizes for bnlearn networks*

The results for this set of networks are presented in Figure 10. Some comments about these results are included below.

- PTs and PPTs always require more memory space than 1DA. Both of them are quite similar except for **win95pts**. This network contains several potentials with repeated values where the prune operation substantially reduces the number of leaf nodes and consequently memory size.

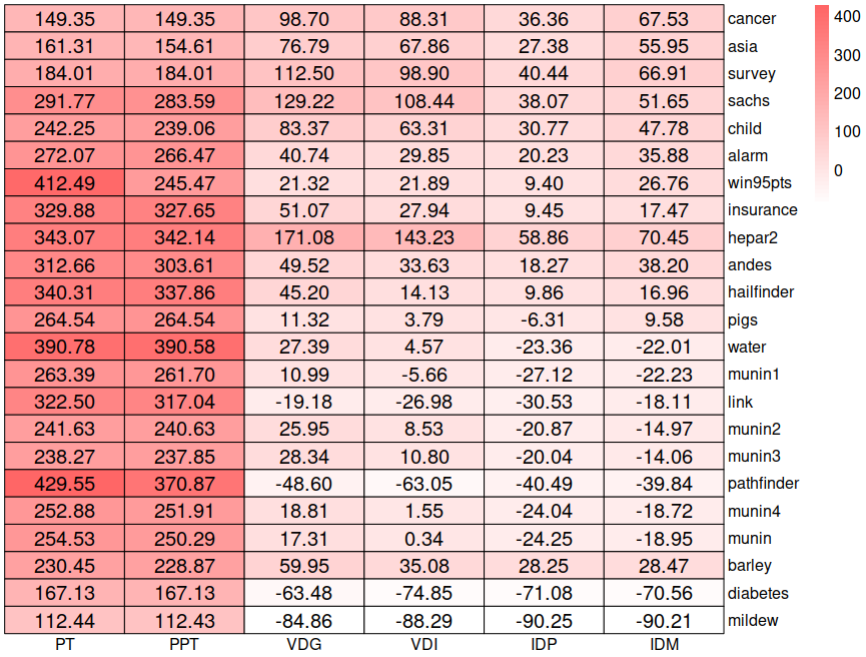


Figure 10: Memory sizes for *bnlearn* networks

- In almost all networks, the most competitive representation is IDP. In networks with a number of parameters under 8427 (from **cancer** to **hailfinder**), IDP requires more memory than 1DA. But for the rest of networks IDP offers memory savings ranging from 6.31% until 90.25%; **barley** network is an exception. The potentials in this network have short sequences of repeated values including few indices. For example, the potential for **jordn** variable has 4752 possible values, but only 4 different ones. However, the sequences are arranged in such a way that they cannot be exploited by PPT. This is the reason why VDG uses many grains. Moreover, potentials contain few zeros, which means that there are many indexes to store. In these cases, it would be appropriate to make the default value be the most repeated one, although this would complicate the way of performing combination and marginalization operations.

- More important savings correspond to **diabetes** and **mildew**. In these two networks there are several potentials with a high number of repeated values. For **diabetes** there are 25 potentials with 7056 parameters but only 44 different values. In **mildew** the same circumstance arises (a potential with 39040 possible values but only 1756 different ones; another one with 201300 parameters and 4508 different values are two examples). In these potentials repeated values can not be collapsed when using PPTs.

- VDI is the best representation for **pathfinder** and **diabetes**. This is

explained by the reduced number of different values respect to the number of parameters. In **pathfinder** a potential with 8064 parameters needs only 29 different values.

*6.2.2. Memory sizes for UAI networks*

The results for these networks are presented in Figure 11.

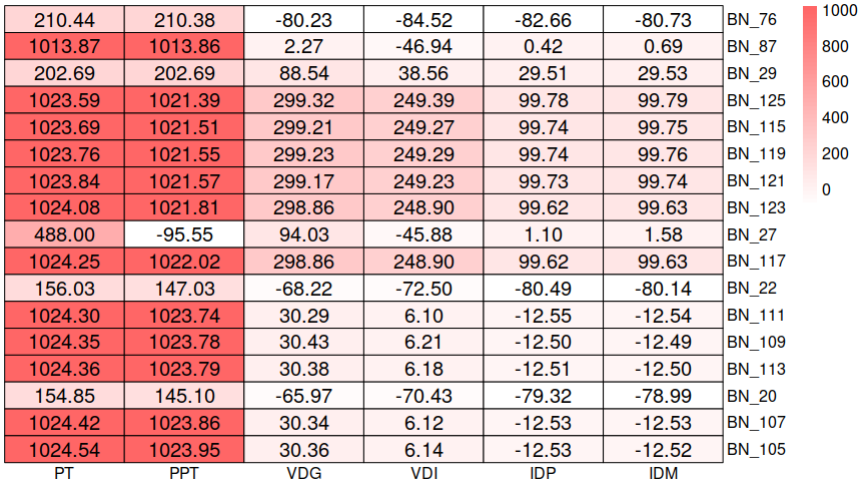


Figure 11: Memory sizes for *UAI* networks

The following conclusions can be outlined from these results:

- Percentages for PTs and PPTs are always over 200% except for **BN_27**. In this network there is an important difference between PT (488%) and PPT ( 95:55%). Furthermore, PPT is the best representation, but VDG offers a similar saving. This network presents 1005 potentials with 3645 parameters but a single value (therefore PPTs contain a single leaf node).
- IDP is the best representation for most of the networks, with substantial savings for **BN_76**, **BN_22** and **BN_20**; moderated savings for **BN_111**, **BN_109**, **BN_113**, **BN_107** and **BN_106**. For the rest of network, the percentages of increase respect to 1DA is the lowest one.
- All the proposed representations offer a competitive alternative respect to PTs and PPTs. In general, the significant savings are produced in networks with a very high number of parameters, where it is really necessary to have efficient representations to be able to apply inference algorithms.

*6.3. Access time*

Although the treatment of complex models implies the need of alternative models that imply worse calculation times (more calculation time is assumed in exchange for saving memory space), it is important to take into account the

speed of access to potential values and the efficiency of the operations necessary for the inference tasks. It would be completely useless to be able to represent complex models but having excessively long computation times.

Therefore, the efficiency of access to potential values is an indispensable requirement. For this reason, part of the experimentation is focused on testing this operation. This experiment is based on: a) random selection of 10000 pairs (potential, index). This set will be used for all the representations of each network. Tables including the results show times in milliseconds.

For the reliable estimation access times, the **Scalameter** library [36] has been used. This tool allows to configure the time-taking experiments ensuring that the machine has reached a steady state (after a warm-up phase); and after that, repeats several times the procedure of interest and finally offers the average time.

*6.3.1. Access times for bnlearn networks*

Access times for *bnlearn* networks are presented in Figure 12. These times show that IDM representation is very competitive, with similar times that 1DA. Times for the rest of alternatives are in most cases shorter than those required for PT and PPT, except for **barley** and **mildew**. In these networks, savings in memory space entail a more complex structure that requires longer access times.

*6.3.2. Access times for UAI networks*

Times for *UAI* networks are shown in Figure 13. In this set, $IDM$ representation is the most advantageous, with times similar to those required for $1DA$. In these complex networks, where VBPs offer significant reductions in memory space, the resulting structures for $VDG$, $VDI$ and $IDP$ lead to worse access times. This is especially relevant for structures where the search is value-driven ($VDG$ and $VDI$).

*6.4. Posterior computation*

Since the objective of this work is to investigate the possibilities of using VBPs in inference algorithms with Bayesian networks it is required to define marginalization and combination operations on these structures.

In general if is a potential defined on $\mathbf{X}$ and $\mathbf{Z}$ $\mathbf{X}$, then the marginalization of in $\mathbf{Z}$ is computed by:

$$^{\#\mathbf{Z}}(\mathbf{z}) = \sum_{\mathbf{x}^{\downarrow\mathbf{Z}}=\mathbf{z}} (\mathbf{x}); \quad 8\mathbf{z} \; 2 \; \Omega_{\mathbf{Z}} \tag{20}$$

where $\mathbf{x}^{\#\mathbf{Z}}$ denotes the projection of $\mathbf{x}$ configuration on $\mathbf{Z}$. This operation can be done by iteratively marginalizing out each variable $Y \; 2 \; \mathbf{X} \, n \, \mathbf{Z}$.

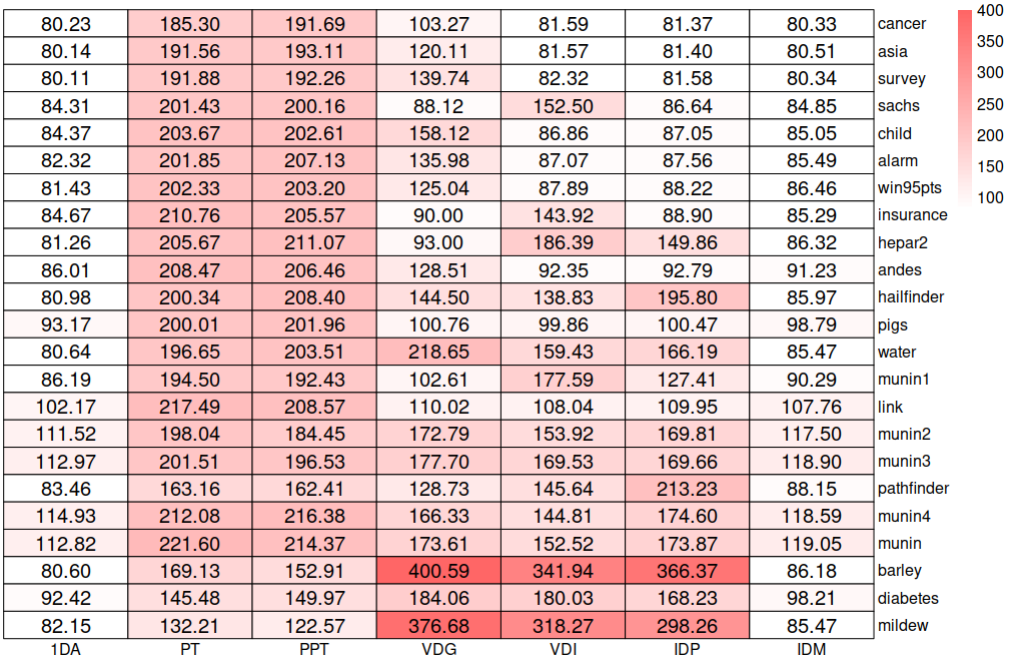


Figure 12: Access times for *bnlearn* networks

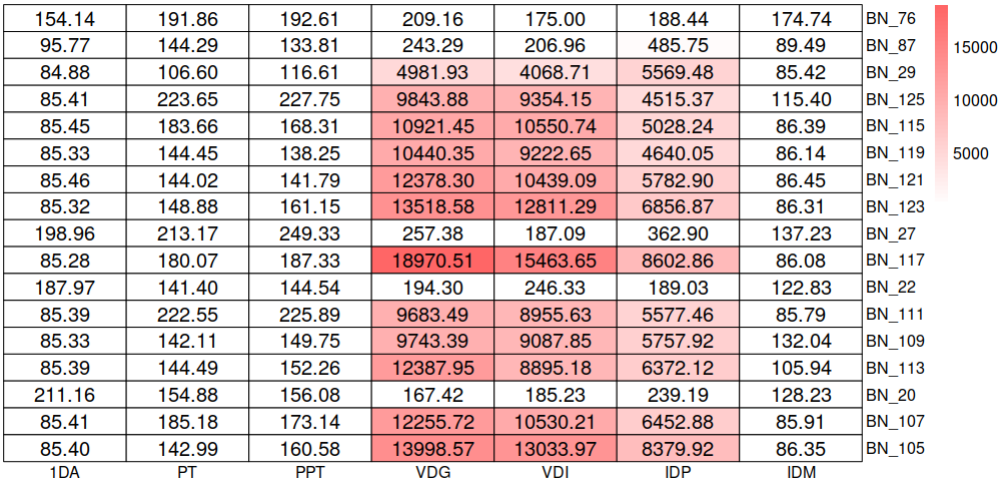


Figure 13: Access times for *UAI* networks

In the case of combination operation, given two potentials $_1(\mathbf{X})$ and $_2(\mathbf{Y})$ then the combination of $_1$ and $_2$ is the potential denoted $_1$ $_2$ defined on $\mathbf{Z} = \mathbf{X}$ *[* $\mathbf{Y}$ by means of pointwise multiplication:

$$_1 \quad _2(\mathbf{z}) = \ _1(\mathbf{z}^{\#\mathbf{X}}) \quad _2(\mathbf{z}^{\#\mathbf{Y}}); \quad 8\mathbf{z}\ 2\ \mathbf{X}\ [\ \mathbf{Y} \tag{21}$$

We have developed simple and direct algorithms for marginalization and combination operations (see Algorithms 5 and 6). These algorithms are based on accessing the values of the potentials (operations defined in Algorithms 1, 2, 3 and 4). This makes the access operation so important. As there is a one to one correspondence between indexes and configurations we refer to them interchangeably (as en example, $I$ index on $\mathbf{Z}$ corresponds to $\mathbf{z}_I$ configuration).

**Algorithm 5** (Marginalization in VBPs)**.** *Given* $VBP\ (\mathbf{X})$ *a potential defined over* $\mathbf{X}$ *and* $Y\ 2\ \mathbf{X}$*. The method for marginalizing out* $Y$ *from* $VBP$ *is described in Algorithm 5. Observe that this algorithm can be employed for all VBP alternatives.*

**Algorithm 5** Marginalization of $Y$ from $VBP\ (\mathbf{X})$

```
1:  function marginalize(VBP ; Y )
2:      Z = X n Y                                   . make result domain
3:      creates VBP_r(Z)                            . empty result potential
4:      for each I = f0:::kg; k = jΩ_Z j  1 do      . loop on VBP (Z) indexes
5:          for each y 2 Ω_Y do
6:              x_Iy   z_I^{"Y=y}                   . get x_Iy configuration compatible with z_I
7:              v_Iy   VBP (x_Iy)                   . value in VBP (X) for x_Iy
8:          end for
9:          VBP_r(z_I)   P_{y2Ω_y} v_Iy
10:     end for
11:     return VBP_r(Z)
12: end function
```

Algorithm 5 removes a variable $Y$ from $VBP\ (\mathbf{X})$. In lines 2 and 3 the final potential domain $\mathbf{Z} = \mathbf{X}\ n\ Y$ is used to create the resulting potential, which will initially be empty. Line 4 iterates over $VBP\ _r(\mathbf{Z})$ indexes. Let assume $I$ corresponds to a specific configuration of variables in $\mathbf{Z}$ (denoted by $\mathbf{z}_I$). Compatible indices $\mathbf{x}_I$ refer to configurations produced completing $\mathbf{z}_I$ with the possible values of $Y$. This operation is noted as $\mathbf{z}_I^{"Y}$. Internal loop (lines 5 to 8) iterates on $Y$ values. The sum of $\mathbf{v}_{Iy}$ values is assigned to the resulting potential (line 9).

**Algorithm 6** (Combination in VBPs)**.** *Given* $VBP\ _1(\mathbf{X})$ *and* $VBP\ _2(\mathbf{Y})$ *two potentials defined over* $\mathbf{X}$ *and* $\mathbf{Y}$*. The method for combining both potentials is presented in Algorithm 6. As it happens with Algorithm 5, this is a general method applicable to all the alternatives previously described: VDG, VDI, IDP and IDM.*

Algorithm 6 combines two potentials $VBP\ _1(\mathbf{X})$ and $VBP\ _2(\mathbf{Y})$. Line 2 produces the domain $\mathbf{Z}$ as $\mathbf{Z} = \mathbf{X}\ [\ \mathbf{Y}$. This is the domain of the potential

**Algorithm 6** Combination of $VBP_1(\mathbf{X})$ and $VBP_2(\mathbf{Y})$

```
1: function combine(VBP_1(X); VBP_2(Y))
2:     Z = X ∪ Y                                   . make result domain
3:     creates VBP_r(Z)                            . empty result potential
4:     for each I = {0, ..., k, k = |Ω_Z| − 1} do  . loop on VBP_r(Z) indexes
5:         v_I ← 0.0
6:         x_I ← z_I^{#X}                          . project z_I index on X
7:         v_1 ← VBP_1(x_I)                        . value in VBP_1(X) for x_I
8:         if v_1 ≠ 0.0 (default value) then
9:             y_I ← z_I^{#Y}                      . project z_I index on Y
10:            v_2 ← VBP_2(y_I)                    . value in VBP_2(Y) for y_I
11:            if v_2 ≠ 0.0 then
12:                v_I = v_1 ⊗ v_2
13:            end if
14:        end if
15:        VBP_r(z_I) ← v_I
16:    end for
17:    return VBP_r(Z)
18: end function
```

$VBP_r(\mathbf{Z})$ to return. Line 4 iterates over $VBP_r(\mathbf{Z})$ indexes. Let be $I$ the index under consideration (it corresponds to a given configuration $\mathbf{z}_I$). The value to assign to $I$ is initialized to 0.0 (line 5). The configuration $\mathbf{z}_I$ must be projected into $VBP_1(\mathbf{X})$ (line 6) and $VBP_2(\mathbf{Y})$ (line 9). These operation are termed $\mathbf{z}_I^{\#\mathbf{X}}$ and $\mathbf{z}_I^{\#\mathbf{Y}}$ and consists of removing from $\mathbf{z}_I$ the values of the variables that do not belong to $\mathbf{X}$ and $\mathbf{Y}$ respectively. The index of configuration $\mathbf{x}_I$ is employed for getting $v_1$ (line 7). If $v_1$ is 0.0 then for sure $v_I = 0.0$ and no more operations are required. Otherwise, it is also necessary to access $VBP_2(\mathbf{y}_I)$ (see line 10). Finally $v_I$ is assigned to $VBP_r(\mathbf{Z})$ in line 15.

This section presents the computation times required for obtaining the posterior on 10 randomly selected variables on each *bnlearn* network with Variable Elimination algorithm and using the algorithms for marginalization and combination previously described. Experiments have been limited *bnlearn* networks because for most of *UAI* networks computations with 1DA, PT and PPT overcomes the memory capacity of the computer used for the experimental work.

In any case, it is important to highlight that these experiments aim to obtain a first idea of the behavior of VBPs structures. A future line of work will be to carry out specific implementations for marginalization and combination operations, which take into account the special properties of each representation.

We have selected one alternative for each category: VDI for value-driven approximation and IDM for index-driven approach. These two representations are the ones that show the best compromise between memory use and access times within their category. They are compared respect to 1DA and PT (when

using trees there is not much difference between PT and PPT in general). We have also employed **Scalameter** library for measuring computation times. The results for this section are presented in Figure 14.

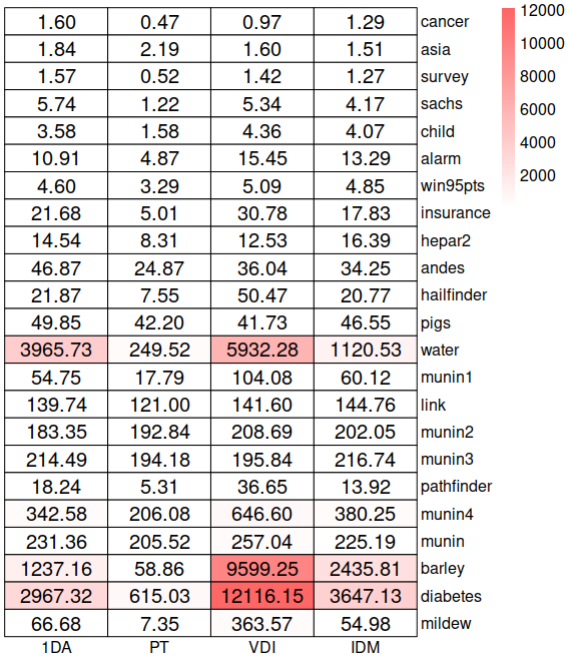


Figure 14: VE times for *bnlearn* networks

Regarding these results, it is observed that the inference with PT is the most efficient one. This is because PT has specific implementations of marginalization and combination operations (see [34]). The implementation of these methods is recursive. It should be noted that in some of *UAI* networks, the execution of the algorithm produces a stack overflow error when PTs with many variables are produced. In these cases the evaluation with 1DA also fails producing out of memory errors.

It is also observed that, in most cases, times for IDM are similar to those for 1DA. This is remarkable and it is possible to think that more refined implementations of marginalize and combine operations, which take into account their structure, will improve current computation times. More efficient implementations of these operations should try to avoid iterating over all indexes of the resulting potential, limiting it to those stored. This, in some of the networks, can suppose significant reductions of time.

## 7. Conclusion

Regarding the use of memory space, it is observed that all the alternatives proposed offer competitive results with respect to 1DA, PT and PPT. In most

of the networks, VDI (value-driven) and IDP (index-driven) alternatives stand out. In terms of access times, the best alternative is IDM. For this reason, this representation was selected as an alternative for VE tests.

The basic versions of marginalization and combination allows to observe that VDI and IDM also offer reasonable execution times, similar to those necessary for 1DA. We think that these results are promising and that more efficient implementations will produce better results. This task will be the subject for a line of future research.

Although it has not been used in this work, it should be noted that another important feature of VBPs lies in its ability to be approximated. The approximation operation assumes loss of information. Intuitively, the idea is to group nearby values and replace them with their average (or some other measure), so that repetition patterns are expanded and, therefore, the number of values to be stored is reduced. With this operation any algorithm that involves the use of approximated potentials will become approximate as well and will ultimately offer non-exact solutions. In very complex problems it is always better to have at least one approximate solution (see [14, 6, 4, 5] as examples of approximation with PTs).

The software used in this paper is implemented with **Scala** programming language and it is available at `https://github.com/mgomez-olmedo/bnetSbtV2` with explanations about how to reproduce the experiments.The functional programming paradigm (combined with object orientation) offered by **Scala** can be exploited for getting well defined operations easily converted into parallel ones on multi-core CPUs when possible. Some of these benefits are investigated in [27].

*Acknowledgements*

This research was jointly supported by the Spanish Ministry of Education and Science under projects PID2019-106758GB-C31 and TIN2016-77902-C3-2-P and the European Regional Development Fund (FEDER).

## References

[1] A. M. Alaa and M. van der Schaar. Bayesian Nonparametric Causal Inference: Information Rates and Learning Algorithms. *IEEE Journal of Selected Topics in Signal Processing*, **2018**, doi: 10.1109/JSTSP.2018.2848230.

[2] M. Arias, F. Díez. Operating with potentials of discrete variables. *International Journal of Approximate Reasoning*, **2007**, *46 (1)*, pp. 166–187.

[3] C. Boutilier, N. Friedman, M. Goldszmidt, D. Koller. Context-specific independence in Bayesian networks. *In: Proceedings of the 12th International Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann Publishers Inc.*, **1996**, pp. 115–123.

[4] R. Cabañas, M. Gómez, A. Cano. Approximate inference in influence diagrams using binary trees. *In: Proceedings of the Sixth European Workshop on Probabilistic Graphical Models (PGM-12)*, **2012**.

[5] R. Cabañas, M. Gómez-Olmedo, A. Cano. Using binary trees for the evaluation of influence diagrams. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, **2016**, *24 (01)*, pp. 59–89.

[6] A. Cano, M. Gómez-Olmedo, S. Moral. Approximate inference in Bayesian networks using binary probability trees. *International Journal of Approximate Reasoning*, **2011**, *52 (1)*, pp. 49–62.

[7] A. Cano, S. Moral, A. Salmerón, Penniless propagation in join trees. *International Journal of Intelligent Systems*, **2000**, *15 (11)*, pp. 1027–1059.

[8] M. Chavira, A. Darwiche. Compiling Bayesian networks using variable elimination. *In: Proceedings of the 20th International Joint Conference on Artifical Intelligence*, **2007**, pp. 2443–2449.

[9] A. Choi, D. Kisa, A. Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. *In: European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty, Springer*, **2013**, pp. 121–132.

[10] P. Dagum, M. Luby, An optimal approximation algorithm for Bayesian inference. *Artificial Intelligence*, **1997**, *93 (1)*, pp. 1–27.

[11] A. P. David, Applications of a general propagation algorithm for probabilistic expert systems. *Statistics and Computing*, **1992**, doi:10.1007/BF01890546.

[12] R. Dechter, Bucket elimination: A unifying framework for probabilistic inference. *In: Learning in graphical models, Springer*, **1998**, pp. 75–104.

[13] F.J. Díez, M. Luque, M. Arias, J. Pérez-Martín. Cost-Effectiveness Analysis with Unordered Decisions. *Artifical Intelligence in Medicine*, **2021**, https://doi.org/10.1016/j.artmed.2021.102064.

[14] M. Gómez-Olmedo, A. Cano. Applying numerical trees to evaluate asymmetric decision problems. *In: T. Nielsen, N. Zhang (Eds.), Symbolic and Quantitative Approaches to Reasoning with Uncertainty, Vol. 2711 of Lecture Notes in Computer Science, Springer Berlin Heidelberg*, **2003**, doi:10.1007/978-3-540-45062-7_16.

[15] M. Gómez-Olmedo. Real-World Applications of Influence Diagrams. *In: J.A. Gámez, S. Moral and A. Salmerón (eds) Advances in Bayesian Networks. Studies in Fuzziness and Soft Computing, Springer, Berlin, Heidelberg*, **2004**, doi:10.1007/978-3-540-39879-0_9.

[16] D. Heckerman, A. Mamdani, M.P. Wellman. Real-World Applications of Bayesian Networks. *In: Association for Computing Machinery*, **1995**, doi: 10.1145/203330.203334.

[17] R. A. Howard, J. E. Matheson. Influence diagram retrospective. *Decision Analysis*, **2005**, *2 (3)*, 144–147.

[18] C. S. Jensen, U. Kjærulff, A. Kong, Blocking Gibbs sampling in very large probabilistic expert systems. *International Journal of Human-Computer Studies*, **1995**, *42 (6)*, pp. 647–666.

[19] F. V. Jensen, T. D. Nielsen, Bayesian networks and decision graphs. *Springer Verlag*, **2007**.

[20] D. Koller, N. Friedman. Probabilistic graphical models: principles and techniques. *MIT Press*, **2009**.

[21] J. Kwisthout. Most probable explanations in Bayesian networks: Complexity and tractability. *Int. J. Approx. Reasoning*, **2011** doi:10.1016/j.ijar.2011.08.003.

[22] S. L. Lauritzen. Graphical Models. *Oxford University Press*, **1996**.

[23] W. Lee and N. Zabaras. Parallel probabilistic graphical model approach for nonparametric Bayesian inference. *Journal of Computational Physics*, **2018**, doi: 10.1016/j.jcp.2018.06.057.

[24] Z. Li, B. D'Ambrosio, Efficient inference in Bayes networks as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, **1994**, *11 (1)*, pp. 55–81.

[25] A. L. Madsen, F. V. Jensen, Lazy evaluation of symmetric Bayesian decision problems. *In: Proceedings of the 15th Conference on Uncertainty in AI, Morgan Kaufmann Publishers Inc.*, **1999**, pp. 382–390.

[26] A. L. Madsen, F. V. Jensen, Lazy propagation: a junction tree inference algorithm based on lazy evaluation. *Artificial Intelligence*, **2004**, *113 (1-2)*, pp. 203–245.

[27] A. R. Masegosa, A. M. Martinez, H. Borchani. Probabilistic graphical models on multi-core cpus using Java 8. *IEEE Computational Intelligence Magazine*, **2016**, *11 (2)*, pp. 41–54.

[28] S. M. Olmsted. On Representing an Solving Decision Problems. *PhD Thesis, Department of Engineering-Economic Systems, Stanford University*, **1983**.

[29] U. Oztok, A. Darwiche. A top-down compiler for sentential decision diagrams. *In: Proceedings of the 24th International Conference on Artificial Intelligence*, **2015**, pp. 3141–3148.

[30] J. Pearl. Bayesian networks: A model of self-activated memory for evidential reasoning. University of California (Los Angeles). Computer Science Department, **1985**.

[31] J. Pearl. Probabilistic reasoning in Intelligent Systems: Networks of Plausible Inference. *Morgan Kaufmann*, **1988**.

[32] J. Pearl, S. Russell. Bayesian networks. *Computer Science Department, University of California*, **1998**.

[33] J. Pearl, Evidential reasoning using stochastic simulation of causal models. *Artificial Intelligence*, **1987**, *32 (2)*, pp. 245–257.

[34] A. Salmerón, A. Cano, S. Moral, Importance sampling in Bayesian networks using probability trees, Computational Statistics & Data Analysis 34 (4) (2000) 387–413.

[35] S. Sanner, D. McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. *In: Proceedings of the 19th International Joint Conference on Artificial Intelligence*, **2005**, pp. 1384–1390.

[36] Scalameter: automate your performance testing today, **2021**, `http://https://scalameter.github.io/`.

[37] M. Scutari. Learning Bayesian networks with the bnlearn R package. *Journal of Statistical Software*, **2010**, doi:10.18637/jss.v035.i03.

[38] M. Scutari. Bayesian network constraint-based structure learning algorithms: Parallel and optimized implementations in the bnlearn R package. *Journal of Statistical Software*, **2017**, doi:10.18637/jss.v077.i02.

[39] R. D. Shachter, Evaluating influence diagrams. *Operations Research*, **1986**, *34 (6)*, pp. 871–882.

[40] R. D. Shachter, B. D'Ambrosio, B. Del Favero, Symbolic probabilistic inference in belief networks. *In: AAAI Proceedings*, **1990**, pp. 126–131.

[41] P. Shenoy, G. R. Shafer. Axioms for probability and belief-function propagation. *In: R. D. Shachter, T. S. Levitt, L. N. Kanal, J. F. Lemmer (Eds.), Uncertainty in Artificial Intelligence, Vol. 9 of Machine Intelligence and Pattern Recognition*, **1990**, doi.org/10.1016/B978-0-444-88650-7.50019-6.

[42] UAI 2016 Inference Competition, **2016**, `http://www.hlt.utdallas.edu/~vgogate/uai16-evaluation/`

[43] UAI 2014 Inference Competition, **2014**, `http://www.hlt.utdallas.edu/~vgogate/uai14-competition/index.html`

[44] T. L. Wong, H. Xie, W. Lam, and F. L. Wang. A learning framework for information block search based on probabilistic graphical models and Fisher Kernel. *International Journal on Machine Learning and Cybernetics*, **2018**, doi: 10.1007/s13042-017-0657-9.

[45] L. Yang and Y. Guo. Combining pre- and post-model information in the uncertainty quantification of non-deterministic models using an extended Bayesian melding approach. *Information Sciences*, **2019**, doi: 10.1016/j.ins.2019.06.029.

[46] N. L. Zhang, D. Poole. Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, **1996**, *5*, pp. 301–328.