Received: 22 September 2020 | Revised: 3 June 2021 | Accepted: 19 June 2021

DOI: 10.1002/int.22573

RESEARCH ARTICLE

WILEY

Value-based potentials: Exploiting quantitative information regularity patterns in probabilistic graphical models

Manuel Gómez-Olmedo[1] | **Rafael Cabañas**[2] | **Andrés Cano**[1] | **Serafín Moral**[1] | **Ofelia P. Retamero**[1]

[1]Department of Computer Science and Artificial Intelligence, University of Granada, Granada, Spain

[2]Dalle Molle Institute for Artificial Intelligence Studies, Lugano, Switzerland

Correspondence
Manuel Gómez-Olmedo, Department of Computer Science and Artificial Intelligence, University of Granada, ETSII, C/Periodista Daniel Saucedo Aranda s/n, Granada 18071, Spain.
Email: mgomez@decsai.ugr.es

Abstract

When dealing with complex models (i.e., models with many variables, a high degree of dependency between variables, or many states per variable), the efficient representation of quantitative information in probabilistic graphical models (PGMs) is a challenging task. To address this problem, this study introduces several new structures, aptly named *value-based potentials* (VBPs), which are based exclusively on the values. VBPs leverage repeated values to reduce memory requirements. In the present paper, they are compared with some common structures, like standard tables or unidimensional arrays, and probability trees (PT). Like VBPs, PTs are designed to reduce the memory space, but this is achieved only if value repetitions correspond to context-specific independence patterns (i.e., repeated values are related to consecutive indices or configurations). VBPs are devised to overcome this limitation. The goal of this study is to analyze the properties of VBPs. We provide a theoretical analysis of VBPs and use them to encode the quantitative information of a set of well-known Bayesian networks, measuring the access time to their content and the computational time required to perform some inference tasks.

KEYWORDS

Bayesian networks, inference algorithms, influence diagrams, probabilistic graphical models

1 | INTRODUCTION

Probabilistic graphical models (PGMs)[1–3] 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 modeled; second, a quantitative component consisting of a set of functions quantifying such dependencies. In PGMs over discrete domains, such as *Bayesian networks* (BN)[4,5] or *influence diagrams*,[6,7] these functions are traditionally represented with tables or unidimensional arrays (marginal or conditional probability tables and utility tables, 1DA in general).

The size of 1DAs increases exponentially with the number of variables in their domains. This property could restrict the ability to represent certain problems with large 1DAs (as memory size requirements can be prohibitive). Moreover, even if we are able to encode the model with 1DAs, problems may arise when performing inference computations. To make inferences, these 1DAs are managed for computing marginal or conditional probabilities for a certain variable, most probable explanation,[8,9] decision tables in the case of influence diagrams, and so on. Some inference algorithms[10–19] use basic operations on potentials: combination, restriction, and marginalization. Combination computes the product of two potentials $\phi_1(\mathbf{X})$ and $\phi_2(\mathbf{Y})$, yielding a new potential with higher dimension $\phi(\mathbf{X} \cup \mathbf{Y})$. 1DAs thus obtained as intermediate results can be very large and exceed the memory capacity of the computer.

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

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).[20–26] These structures can capture context-specific independencies[20] and save memory space when repeated values appear under certain circumstances. These representations also allow to obtain approximations through pruning operations: Assuming a loss of information, some contiguous values can be substituted by their average value to reduce memory space. There is previous work focused on improving the operations on potentials to alleviate the computational cost when dealing with complex models.[27] Therefore, the existence of efficient structures for storing and managing quantitative information is of interest in all those areas where PGM-based models can be applied (i.e., in any problem or system that requires the quantification of uncertainty or preferences.[28–34]).

Other data structures exploiting these independencies allow compiling a full PGM into a more compact representation. This is the case of algebraic decision diagrams (ADDs),[35,36] sequential decision diagrams (SDDs)[37,38] and recursive probability trees (RPTs).[39,40] 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 that minimizes the number of arithmetic operations during inference. Similarly, an SDD is a full binary tree for representing propositional knowledge bases (a.k.a. Boolean functions). This data structure allows encoding potentials which can then be conjoined to obtain an efficient SDD representation of a full model. RPTs provide efficient representations of joint distributions exploiting independencies between the variables, and therefore encoding the whole information of complete Bayesian networks.

In this study, some new alternative representations for potentials are considered. They are based on the properties of the values themselves, not on the contexts where they appear nor the structure of the potential. For this reason, they received the name *value-based potentials* (VBPs). This paper defines these structures, showing a theoretical analysis of their properties and specific examples of how they encode the quantitative information of known Bayesian networks, available in the *bnlearn* package repository,[41,42] as well as other networks used in *UAI*'s inference competitions.[43,44]

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. Second, a VBP represents a single potential, independently of the other parameters in the model. This allows easily adapting many inference algorithms for working with VBPs, by simply adjusting the basic operations over potentials (e.g., combination and marginalization). This is not the case for ADDs and SDDs, where a complex compilation process is done to obtain a compact representation of the full model. VBPs, on the other hand, do not include any information about variable dependencies and are based solely on the values, so they cannot be used to represent complete models.

The structure of this paper is as follows: Section 2 defines some basic concepts and notation, as well as some usual representations of potentials as arrays and trees. Section 3 introduces basic concepts about memory requirements analysis. Section 4 introduces VBP representations and how to categorize them. Section 5 introduces VBP alternatives and their properties. Section 6 presents the empirical evaluation performed for testing the features of VBPs. 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 will represent 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 Ω_{X_i}. For simplicity, we will consider variable values as integers, beginning with 0, and hence possible assignments will be $X_1 = 0$, $X_1 = 1, X_1 = 2$, and so on. 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, for example, $\mathbf{X} \coloneqq \{X_1, X_2, \ldots X_N\}$ is a set of N variables ($|\mathbf{X}| = N$). The Cartesian product $\prod_{X_i \in \mathbf{X}} \Omega_{X_i}$ is denoted by $\Omega_{\mathbf{X}}$. The elements of $\Omega_{\mathbf{X}}$ are called configurations of $\mathbf{X}$ and will be represented by $\mathbf{x} \coloneqq \{X_1 = x_1, X_2 = x_2, \ldots, X_N = x_N\}$, or simply $\mathbf{x} \coloneqq \{x_1, x_2, \ldots, x_N\}$ if the variables are obvious from the context.

Formally, a PGM contains three elements $\langle \mathbf{X}, P, \mathcal{G} \rangle$, where $\mathbf{X}$ is the set of variables of the problem with a joint probability distribution $P(\mathbf{X})$, and $\mathcal{G}$ is a graph that represents the dependency (and independence) relations between the variables. A PGM allows to represent P, which is usually high-dimensional, as a factorization of lower dimensional local functions. For instance, in the 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 ϕ 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 states 2, 3 and 2, respectively. Then $\phi(X_1, X_2, X_3)$ is a potential defined on such variables with the values shown in Figure 1. It should be noted 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 indices starting with 0 (all the variables take their first value) and ending with $|\Omega_{\mathbf{X}}| - 1$. In the potential given in Figure 1, index 0 is associated to $\{0, 0, 0\}$, index 1 is associated to $\{0, 0, 1\}$, and so on until the last one, 11, which is associated to $\{1, 2, 1\}$ (indices are shown in leftmost column).

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

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

For the potential considered in Example 1, the values of *weights* are: $w_3 = 1, w_2 = 2, w_1 = 6$. Therefore, the leftmost variable is the one with the highest *weight*. Therefore, the index of a certain configuration $\mathbf{x} := \{x_1, x_2, ..., x_N\}$ can be computed using the following expression:

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

Example 2. Let us consider the potential $\phi(X_1, X_2, X_3)$ given in Example 1, with $w_1 = 6, w_2 = 2$, and $w_3 = 1$. The indices for the domain configurations can be computed as follows:

$index$	x_1	x_2	x_3	$\phi(x_1, x_2, x_3)$
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 $\phi(X_1, X_2, X_3)$ as a mapping that assigns a numeric value to each configuration

$$\begin{aligned}
index(\{0,0,0\}) &= 0\cdot 6 + 0\cdot 2 + 0\cdot 1 = 0\\
index(\{0,0,1\}) &= 0\cdot 6 + 0\cdot 2 + 1\cdot 1 = 1\\
index(\{0,1,0\}) &= 0\cdot 6 + 1\cdot 2 + 0\cdot 1 = 2\\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
index(\{1,2,1\}) &= 1\cdot 6 + 2\cdot 2 + 1\cdot 1 = 11
\end{aligned}$$

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

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

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

Note that the association between indices and configurations requires an order of the variables in the domain. Any order is valid, but we will consider by default the order in which variables are written (e.g., for potential $\phi(X, Y, Z)$, the first variable would be X). In addition, we consider that the first variable has the highest weight (i.e., following *row-major order*). However, the opposite approach could be also considered: the first variable with weight 1 and the last one with the highest weight (*column-major order*).

2.2 | Representation for potentials

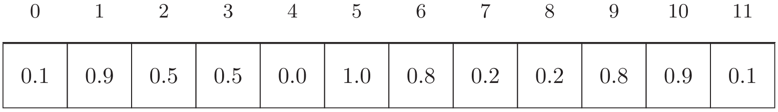
In essence, a potential is a multidimensional object with a dimension per variable. Thus, a standard method for storing it is to flatten this object into a 1DA in computer memory.[3] Thus, potential ϕ defined on a set of N variables can be represented by array $\mathcal{A}_\phi$ as follows:

$$\mathcal{A}_\phi := [\phi(0,0,\ldots,0), \phi(0,0,\ldots,1), \ldots, \phi(|\Omega_{X_1}| - 1, |\Omega_{X_2}| - 1, \ldots, |\Omega_{X_N}| - 1)] \tag{4}$$

The main advantage of the representation with 1DA consists in the fact that the position where each value is stored matches the index of the corresponding configuration. This allows a very efficient access through indices. The size of a 1DA, denoted by $size(\mathcal{A}_\phi)$, is the number of entries, which is equal to the number of configurations in the potential.

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

This representation has limitations. The access to 1DA through configurations would require a transformation into indices using Equation (2); the coincidence between indices and storage positions requires storing all values. Therefore, the storage of repeated values in consecutive positions cannot be avoided.



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 $\phi(X_1, X_2, X_3)$ as a 1DA

A *probability tree* (PT) is an alternative structure that has been used to store potentials in PGMs and operate them,[21,22,24] both accurately and approximately. The tree storing a potential ϕ, $\mathcal{T}_\phi$, 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 $\mathcal{T}$, $size(\mathcal{T})$, 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).

With PTs, the most efficient access method is by means of configurations: the tree must be traversed from root to leaves, selecting the corresponding branch for each variable until a leaf node with its value is reached. In addition, PTs can benefit from context-specific independencies,[20] as many identical values can be grouped into a single value, thus providing a compact storage. The operation that collapses identical values is called *pruning*.

Example 5. The potential in previous examples presents a context-specific independence that allows reducing its size: The value for $X_1 = 0, X_2 = 1$ is 0.5, regardless of the value of X_3. Once the pruning is complete, the result is a pruned PT (PPT) of size 19, shown in Figure 4.

However, PTs cannot leverage other repetition patterns. Let us consider the tree presented in Figure 3. The values for indices 7 and 8 are 0.2, and they are consecutive, but cannot be pruned because they correspond to configurations for which values X_2 and X_3 vary.

A variant of PTs, called binary trees (BPTs),[24] can divide the domain of each variable into two subsets of states. Compared to regular trees, this feature of BPTs allows exploiting finer-grained context independencies. For instance, in the PT (left-side of Figure 5), values 0.4 in configuration c (left subtree) cannot be pruned. However, with BPTs, grouping states 0 and 2 of X_k would allow representing value 0.4 with a single leaf node (as shown in the BPT at the right-side of Figure 5). This reduces memory space for context c, but it would require more nodes for the subtree at the right-side (context c'). For this reason, only PTs and PPTs are considered for comparison in this study.

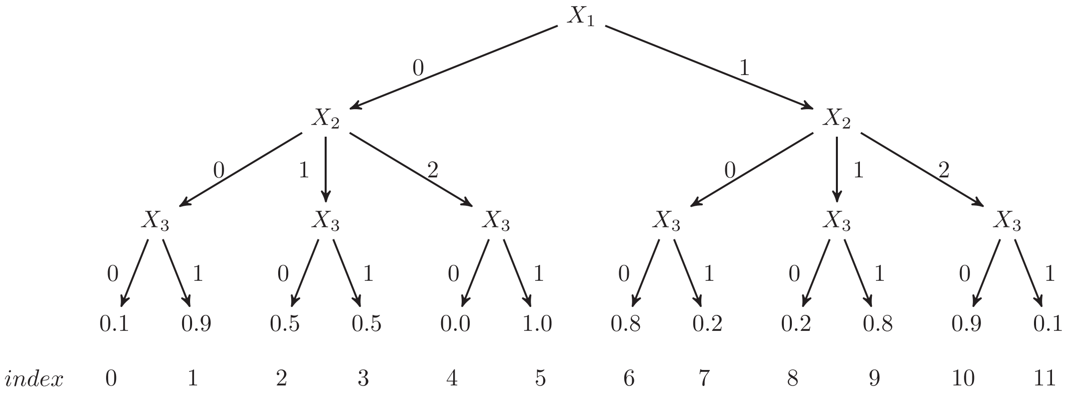


FIGURE 3 $\phi(X_1, X_2, X_3)$ as probability tree

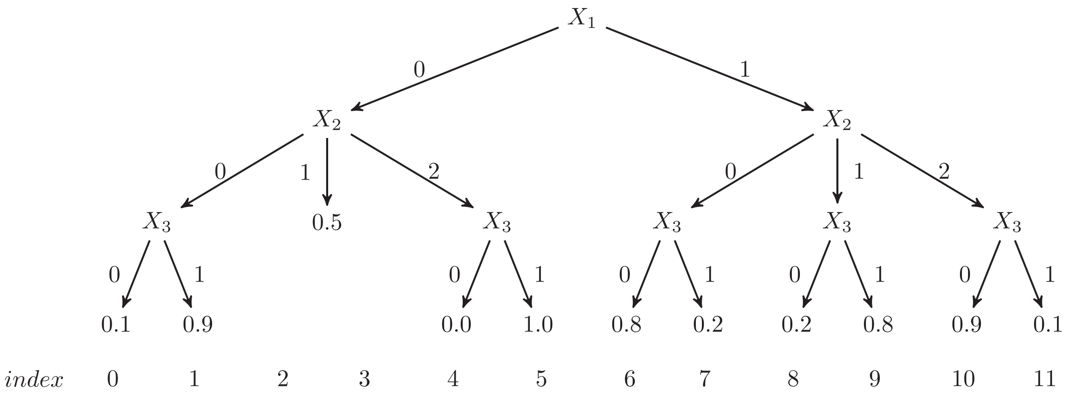


FIGURE 4 $\phi(X_1, X_2, X_3)$ as pruned probability tree

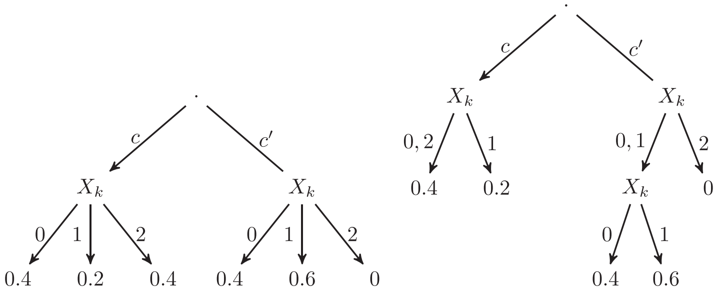


FIGURE 5 Binary tree representation

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 one uses different data types. For this analysis we will consider a potential ϕ defined over a set of N variables $\mathbf{X} := \{X_1, X_2, ...X_N\}$. Additionally, we define the following notation related to the different memory sizes:

- s_f is the memory required for storing a float value.
- s_i is the memory required for storing an index denoting a specific 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 fixed value for all of them (in fact, this memory space will be negligible compared to the memory used for storing a potential). Moreover, we can define a standard way of coding variables with numerical identifiers and follow 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). A specific notation will be used for this term hereafter: s_{arr}, s_{list}, s_{set}, and s_{dict}, respectively.

As mentioned before, the representation by means of 1DA offers an important advantage: The values for configurations are stored consecutively. That is, the value in position k corresponds to the kth configuration. In this way, there is no need to store information about indices. Therefore, the codification of the representation assumes 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 $\mathcal{A}_\phi$ be a 1DA representing $\phi(\mathbf{X})$, $|\mathbf{X}| = N$. Then, the amount of required memory is given by the following expression:*

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

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

Example 6. From the previous examples, consider potential $\phi(X_1, X_2, X_3)$ and its codification as a 1D-array given in Example 3. The estimated memory size is:

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

The tree representation (PT and PPT) is usually less efficient in terms of memory requirements, since the full structure of the tree must be stored. Thus, the amount of memory depends on the number of internal nodes, denoted by n_I, and the number of leaves n_L. It should be noted that $n_I + n_L = size(\mathcal{T})$. Internal nodes store links (or references) to subtrees (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 $\mathcal{T}_\phi$ be a tree representing $\phi(\mathbf{X})$, $|\mathbf{X}| = N$. Then, the amount of memory required is estimated as follows:*

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

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

In the case of a nonpruned tree, the number of leaf nodes will be equal to the number of configurations in the potential. Consequently, the first two terms in Equations (5) and (7) are equal. Thus, the main factors that determine the size of a tree are the data structure employed for storing links to subtrees and the information on variables repetition.

Example 7. Let $\mathcal{T}_\phi$ be the PT from Example 4 (Figure 3) containing seven internal nodes for binary variables, two internal nodes for ternary variables, and 12 leaves. Similarly, let $\mathcal{T}'_\phi$ be the PPT from Example 5 (Figure 4) with six internal nodes for binary variables, two internal nodes for ternary variables, and 11 leaf nodes. To compute their memory cost, the following expressions can be used:

$$memory(\mathcal{T}_\phi) = 3s_v + 12s_f + 7(s_v + s_s + 2s_r) + 2(s_v + s_s + 3s_r) \tag{8}$$

$$memory(\mathcal{T}'_\phi) = 3s_v + 11s_f + 6(s_v + s_s + 2s_r) + 2(s_v + s_s + 3s_r) \tag{9}$$

In the previous example, the pruning operation has reduced the number of internal nodes as well as the number of leaves; however, the cost is higher than the size of the table representation. Memory savings with PTs often require a large number of repeated values.

For a specific analysis, let us assume the following sizes of data types (the real sizes can vary depending on the architecture of the machine used; in fact, real sizes are not relevant as long as the same sizes are used for all comparisons):

- long: 4 bytes (for indices).
- 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 value of s_s will depend on the specific data structure employed:
  - array and list (s_{arr} and s_{list}, respectively): 16 bytes.
  - set (s_{set}): 32 bytes.
  - dictionary (s_{dict}): 64 bytes.

For these sizes, the estimated memory cost of representations $\mathcal{A}_\phi$, $\mathcal{T}_\phi$, and $\mathcal{T}'_\phi$ are 262, 1000, and 910, respectively.

4 | VALUE-BASED REPRESENTATIONS

4.1 | Motivation

At this point, it is clear the need for efficient quantitative information handling mechanisms for the representation, inference, and learning tasks of PGMs. We have already seen that PTs allow to capture some repetition patterns in very specific situations. For VBPs, the underlying idea is to let the values guide the representation process and save as much space from repetitions as possible. Therefore, VBPs were designed with the following goals in mind:

- Being able to take advantage of all repetition patterns, regardless of the order in which they appear.
- Allow an efficient access to values and provide the necessary operations to perform inference tasks. In this study, basic implementations for combination and marginalization operations are provided to get an initial estimate of the behavior of VBPs when using inference algorithms.
- Facilitate the approximation task and the parallel management. This study does no explore these features. Nevertheless, they must be considered to devise a good design taking into account these possibilities (to be explored as future research).

4.2 | Categorization of alternatives

The proposed alternatives can be classified in two groups depending on how the queries are made:

- Approaches *driven by values*, 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 Indices* (VDI).
- Alternatives *driven by indices*, where the keys are indices. From this group we present *Index-Driven with Indices* (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 indices and the information required to link indices and values. IDM uses a map (M) with indices 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 feature is outlined here. It must be clear that these representations require a search for managing the information. This can be exploited by defining a default value that will be returned 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, to reduce the memory requirements, even if computation requires more time. This study considers the first alternative (i.e., 0.0 as default value).

5 | VBPS DESCRIPTION

5.1 | Value-driven with grains

Identical values in potentials will often appear in configurations with consecutive indices. Consider for instance the potential given in Example 3, in which the value 0.5 appears in positions 2 and 3. A similar situation happens with value 0.2 in positions 7 and 8. It follows that the sets of configurations associated to the same value can be defined in a compact way by using 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 indices of valid configurations on $\Omega_{\mathbf{X}}$, with $i \leqslant j$. A grain $g(i, j)$ defines a sequence of consecutive indices $i, i + 1, ...j$. Grains will be used for representing sequences of repeated values in VBPs.

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

Definition 2 (VDG). Let ϕ be a potential defined over $\mathbf{X}$. Then a value-driven with grains for ϕ, VDG$_{\phi}$, is a dictionary D in which entries are defined as $<v, L_v>$, where $v \in \phi$ (key) is a nondefault value and L_v is a list of grains that store the associated indices. Therefore, for each grain $g(i, j) \in L_v$, all the indices correspond to v (i.e., for all $k = i, i + 1, ..., j$ then $\phi(\mathbf{x}_k) = v$).

Example 8. The potential $\phi(X_1, X_2, X_3)$ used in previous examples and presented in Figure 1 will be represented with VDG, as shown in Figure 6.

The outermost rectangle with rounded corners represents the dictionary. The circular nodes indicate entry keys. The related list of grains associated with each key is represented on the right. It can be seen in Figure 6 that each value is stored only once.

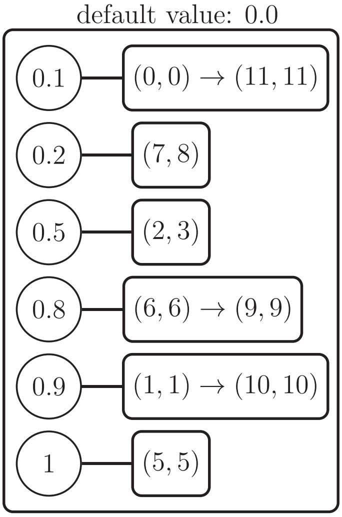

default value: 0.0
0.1
$(0,0) \rightarrow (11,11)$
0.2
$(7,8)$
0.5
$(2,3)$
0.8
$(6,6) \rightarrow (9,9)$
0.9
$(1,1) \rightarrow (10,10)$
1
$(5,5)$

FIGURE 6 $\phi(X_1, X_2, X_3)$ as value-driven with grains

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 indices 0 and 11. Since these indices are not consecutive, they must be stored in two different grains. Values 0.2 and 0.5 appear in consecutive indices and their corresponding grains capture the sequences of repetitions.

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

Algorithm 1 Access to index l in VDG_ϕ

```
1: function ACCESS ($\mathrm{VDG}_\phi$, $l$)
2:   $result = 0.0$                               ▷ sets the default value to result
3:   for each $v$ (key) in key set $D$ do         ▷ loop over dictionary entries
4:       $L_v \leftarrow D(v)$                      ▷ list of grains for $v$
5:       for each $g$ (grain) in $L_v$ do
6:         if $l \in g$                            ▷ $l$ 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 over the grains for this value. Since 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 (the default value) is returned. This would be the case for index 4.

Proposition 3 (Memory space for a VDG representing a potential). *Let* VDG_{ϕ} *be the representation of* $\phi(\mathbf{X})$*, with* $|\mathbf{X}| = N$*. Let us assume that* d *represents the number of different values in the potential (discarding the default value). The number of grains for each value is denoted by* $g_1 ... g_d$*. Then the amount of memory required is estimated as follows:*

$$memory(\text{VDG}_{\phi}) = N \cdot s_v + s_f + s_{dict} + d \cdot (s_f + s_{list}) + \sum_{j=1}^{d} 2 \cdot g_j \cdot s_i \tag{10}$$

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

Example 9. Let VDG_{ϕ} be the VDG from Example 8 with six different values to store in the dictionary and 0.0 as the default value. Therefore, the dictionary stores six entries. The sequences of repetitions require nine grains. Hence, the memory cost can be computed using the following expression:

$$memory(\text{VDG}_{\phi}) = 3s_v + s_f + s_{dict} + 6 \cdot (s_f + s_{list}) + 9 \cdot 2 \cdot s_i \tag{11}$$

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

5.2 | Value-driven with indices

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. The entry includes two grains of length 1: $(0, 0)$ and $(11, 11)$. This repetition can be avoided by, for example, associating values to the complete list of indices in which they appear. In this example, value 0.1 will be related to the list $(0 \rightarrow 11)$. This alternative will be advantageous if the sequence of values for a potential does not contain large series of repetitions. Having this idea in mind, the following representation can be defined.

Definition 3 (VDIs). Let ϕ be a potential defined over $\mathbf{X}$. A VDIs for ϕ, VDI_{ϕ}, is a dictionary D in which each entry $<v, L_v>$ contains a value (as key) and a list of indices L_v, such that $\phi(\mathbf{x}_i) = v$ for each $l \in L_v$.

Example 10. The potential $\phi(X_1, X_2, X_3)$ used before and described in Figure 1 will be represented as VDI as shown in Figure 7. The outermost rectangle represents the dictionary: keys of entries are drawn as circles. Keys give access to lists of indices (rectangles of rounded corners).

Algorithm 2 Access to index in VDI. Given VDI_{ϕ}, the algorithm for getting the value corresponding to a given index l is described in Algorithm 2.

Algorithm 2 Access to index l in VDI_ϕ

```
1: function ACCESS ($\text{VDI}_\phi, l$)
2:   $result = 0.0$                                   ▷ sets the default value to result
3:   for each $v$ (key) in key set $D$ do             ▷ loop over dictionary entries
4:       $L_v \leftarrow D(v)$                        ▷ list of indices for $v$
5:       for each $p$ in $L_V$ do                     ▷ loop over list of indices
6:         if $p == l$ then                           ▷ $l$ 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
```

Let us assume that 6 is the target index. The search progresses through dictionary entries until reaching key 0.8. The internal loop (Line 5) iterates over the list. Since it contains 6, the result would be 0.8. A failed search produces 0.0 as the result. This is the case for index 4.

Proposition 4 (Memory space for a VDI that represents a potential). *Let* VDI_ϕ *be the representation of* $\phi(\mathbf{X})$ *with* $|\mathbf{X}| = N$. *Let us assume that* d *represents the number of different values in the potential (discarding the default value). The number of indices for each value is denoted by* $i_1 \ldots i_d$. *The required amount of memory is estimated as follows:*

$$memory(\text{VDI}_\phi) = N \cdot s_v + s_f + s_{dict} + d \cdot (s_f + s_{list}) + \sum_{j=1}^{d} i_j \cdot s_i \qquad (12)$$

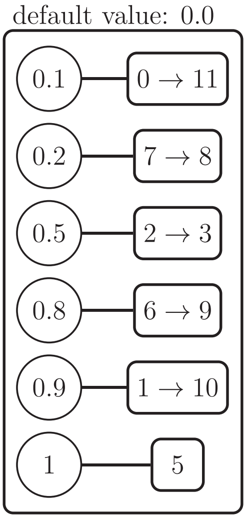


FIGURE 7 $\phi(X_1, X_2, X_3)$ as value-driven with indices

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

Example 11. Let VDI_ϕ be the VDI from Example 10 with six different values to store in the dictionary and 0.0 as default value. Therefore, the dictionary stores six pairs and six lists with 11 indices. The memory cost can be computed using the following expression:

$$memory(\text{VDI}_\phi) = 3s_v + s_f + s_{dict} + 6 \cdot (s_f + s_{list}) + 11 \cdot s_i \tag{13}$$

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

5.3 | 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, l, requires iterating over the list of entries and over the associated lists (of grains or indices). To avoid this double iteration and make the search more efficient, new structures are introduced in which the search is based on the indices themselves. This is the case of IDP and IDM.

Definition 4 (IDP). Let ϕ be a potential defined over $\mathbf{X}$. Then a structure IDP representing ϕ, IDP_ϕ, is a pair of arrays: V and L. Nonrepeated values in ϕ (excluding 0.0 as default value) are stored in $V := \{v_0, v_1, ..., v_{d-1}\}$. Let nd_ϕ represents the set of indices storing nondefault values. The array L is defined as follows:

$$L := \{(i, j) : \phi(\mathbf{x}_i) = v_j, i \in nd_\phi\} \tag{14}$$

That is, IDP is based on two components. First, an array storing the values (without repetitions, as before, and excluding 0.0 as the default value). Second, an array of pairs (index in potential—index in array of values). The second index of the pair keeps the relation between indices and values.

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

As explained before, IDP uses two coherent arrays. V stores nonrepeated values (except the default value). L contains pairs of indices. Let us consider value 0.5 in $\phi(X_1, X_2, X_3)$, presented in indices 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 indices 2 and 3 store the value in $V(2)$.

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

Algorithm 3 Access to $\boldsymbol{l}$ index in $\mathbf{IDP}_{\phi}$

```
function ACCESS ($\text{IDP}_{\phi}, l$)
    $result = 0.0$                                   ▷ sets the default value to result
    for each pair $t = (i_{\phi}, i_V)$ in $L$ do           ▷ loop over $L$ array pairs
        if $i_{\phi} == l$ then                       ▷ $l$ is found; value stored in $V(i_V)$
            $result = V(i_V)$ and stop iteration
        end if
    end for
    return $result$
end function
```

Let us assume that 6 is the target index. The search progresses through L until reaching the 6th position (pair $(6, 3)$). The value to be returned 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_{ϕ} *be the structure representing* $\phi(\mathbf{X})$, $|\mathbf{X}| = N$. *Let us assume that* d *represents the number of different values in the potential (discarding the default value). The number of indices corresponding to a nondefault value is* p. *The amount of memory is estimated as follows:*

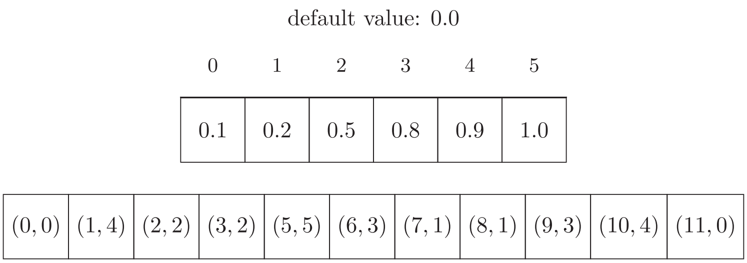
$$memory(\text{IDP}_{\phi}) = N \cdot s_v + s_f + 2 \cdot s_{arr} + d \cdot s_f + 2 \cdot s_i \cdot p \tag{15}$$

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

Example 13. Let IDP_{ϕ} be the IDP from Example (12) with six different values to store and 0.0 as the default value. Therefore, V stores six values and L 11 pairs. The memory cost can be computed using the following expression:

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

Using the specific memory sizes described in Section 3, the complete amount of memory is 326 bytes.



default value: 0.0

0	1	2	3	4	5
0.1	0.2	0.5	0.8	0.9	1.0

$(0,0)$	$(1,4)$	$(2,2)$	$(3,2)$	$(5,5)$	$(6,3)$	$(7,1)$	$(8,1)$	$(9,3)$	$(10,4)$	$(11,0)$

FIGURE 8 $\phi(X_1, X_2, X_3)$ as index-driven with pairs

5.4 | 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 indices. The definition of this new index-driven alternative can be found below.

Definition 5 (IDM). Let ϕ be a potential defined over $\mathbf{X}$. Then, a structure IDM representing ϕ, IDM_ϕ consists of a dictionary D and an array V. Nonrepeated values in ϕ are stored in $V := \{v_0, v_1, \ldots, v_{d-1}\}$. D entries $<i, j>$ are formed by indices (keys) ϕ and indices V. Let nd_ϕ represent the set of indices storing nondefault values. Given an entry $<i, j>$, then: $\phi(\mathbf{x}_i) = v_j$ and $i \in nd_\phi$.

Example 14. Figure 9 shows the representation as IDM of the potential $\phi(X_1, X_2, X_3)$ presented in Figure 1. The dictionary D is represented on the left side of Figure 9 and the array of values V is on the right side. Keys in D are drawn as circles and give access to V indices (boxes linked to keys).

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

Algorithm 4 Access to $\boldsymbol{l}$ index in $\mathbf{IDM}_\phi$

```
function access ($\mathrm{IDM}_\phi, l$)
    $result = 0.0$                      ▷ sets the default value to result
    $entry(<l, j>) \leftarrow D(l)$     ▷ searches $l$ in dictionary
    if entry! = null then               ▷ dictionary contains $l$ as key
        $result \leftarrow V(j)$
    end if
    return $result$
end function
```

Let us assume that 6 is the target index. Since this is a valid key, entry <6, 3> is retrieved. The value to be returned 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_ϕ *be the structure representing* $\phi(\mathbf{X}), |\mathbf{X}| = N$. *Let us assume that* d *represents the number of different values in the potential (discarding the default value) and* p *is the number of indices storing nondefault values. The amount of memory is estimated as follows.*

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

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

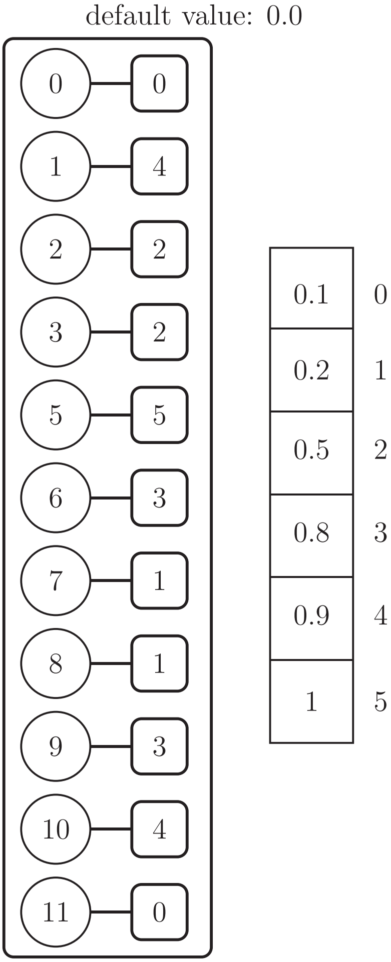


FIGURE 9 $\phi(X_1, X_2, X_3)$ as index-driven with map

Example 15. Let IDM $_\phi$ be the VBP from Example (14) with six different values to store and 0.0 as the default value. Hence, the array of values has six elements. The dictionary contains 11 entries. The memory cost can be computed using the following expression:

$$memory(\mathrm{IDM}_\phi) = 3s_v + s_f + s_{dict} + s_{arr} + 6 \cdot s_f + 2 \cdot 11 \cdot s_i \tag{18}$$

Considering the specific values as detailed in Section 3, the complete amount of memory is 374 bytes.

5.5 | Example of extreme case

Let us consider an example of using the representation structures under consideration for a potential with extreme features: only three different values (0.0, 0.5, and 1.0) and many repetitions of one of them (0.0 used as default value; around 70% of the indices are assigned to 0.0). The potential has 1024 possible values, with five variables in its domain, each with four states. The values are randomly generated. We have considered 10 random potentials with these features to get a reliable idea of the behavior for these representations. Results are shown in Table 1. 1DA and *PT* representations do not depend on the specific values and require memory sizes of 8574 and 53,816, respectively.

For 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 compared to 1DA, PT, and PPT are noticeable.

6 | EMPIRICAL EVALUATION

Two sets of Bayesian networks are used for evaluating VBPs capabilities against previous representations of potentials in PGMs: conditional probability Table 1 and trees (PT and PPT). The first set is taken from the *bnlearn* repository[41,42] and the second one from *UAI* competitions.[43,44] 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 time and computation time of posterior distributions using the variable elimination (VE) algorithm.[11,45,46]

The representations compared in experiments are:

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

6.1 | Features of Bayesian networks

Some basic information about the Bayesian networks employed is gathered in Tables 2 and 3: name of network; number of nodes, number of arcs; minimum, average and maximum number of variable states; and complete number of parameters for quantifying the uncertainty in the networks. Networks are ordered by number of parameters. Observe that networks from the *UAI* set require more parameters than the networks from *bnlearn*.

6.2 | Comparing memory sizes

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 the 1DA representation and m_{rep} the memory size for a different representation. The value s included in the table cells and representing the gain (or loss) for *rep* is computed as:

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

TABLE 1 Memory sizes for random potential representations

Iteration	1	2	3	4	5	6	7	8	9	10
PPT	39,090	38,332	37,658	38,396	40,946	38,910	42,610	38,120	42,706	38,544
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 2 *bnlearn* features of Bayesian networks

Network	Nnodes	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
hailfinder	56	66	2	3.98	11	3741
pigs	441	592	3	3	3	8427
water	32	66	3	3.625	4	13,484
munin1	186	273	2	5.33	21	19,226
link	724	1125	2	2.53	4	20,502
munin2	1003	1244	2	5.36	21	83,920
munin3	1041	1306	2	5.38	21	85,615
pathfinder	109	195	2	4.11	63	97,851
munin4	1038	1388	2	5.44	21	97,943
munin	1041	1397	2	5.43	21	98,423
barley	48	84	2	8.77	67	130,180
diabetes	413	602	3	11.34	21	461,069
mildew	35	46	3	17.6	100	547,158

TABLE 3 *UAI competition* features of Bayesian networks

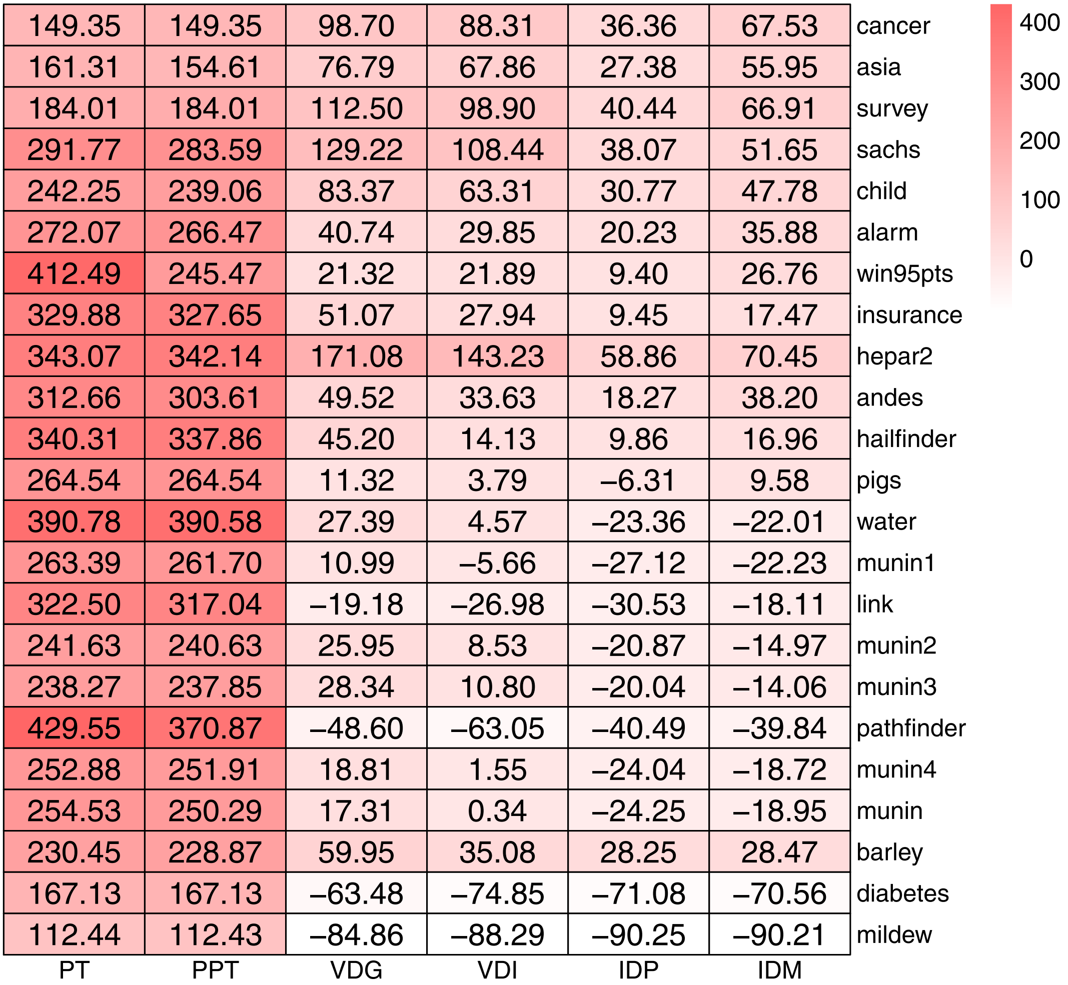
Network	Nodes	Arcs	Min. st.	Avg. st.	Max. st.	Parameters
BN_76	2155	3686	2	7.01	36	627,298
BN_87	422	867	2	2	2	933,776
BN_29	24	30	10	10	10	1,132,080
BN_125	50	375	2	2	2	2,117,680
BN_115	50	375	2	2	2	2,285,616
BN_119	50	375	2	2	2	2,410,544
BN_121	50	375	2	2	2	2,564,144
BN_123	50	375	2	2	2	3,249,200
BN_27	3025	7040	3	6	10	3,698,565
BN_117	50	375	2	2	2	4,003,888
BN_22	2425	4239	2	18.743	91	4,073,904
BN_111	50	375	2	2	2	4,238,512
BN_109	50	375	2	2	2	4,581,936
BN_113	50	375	2	2	2	4,669,488
BN_20	2483	5272	2	18.92	91	5,009,364
BN_107	50	375	2	2	2	5,154,864
BN_105	50	375	2	2	2	6,431,792

Thus, a negative value for s indicates that *rep* requires less memory space than 1DA. Conversely, positive values indicate a 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:

- 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, the memory size.
- For most of the networks, the most competitive representation is IDP. For networks with a number of parameters lower than 8427 (from **cancer** to **hailfinder**), IDP requires more memory than 1DA. But for the rest of the networks, IDP offers memory savings ranging from −6.31% to −90.25%; the **barley** network is an exception. The potentials in this network have short sequences of repeated values including few indices. For example, the potential for variable **jordn** has 4752 possible values, but only four are different. 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



PT	PPT	VDG	VDI	IDP	IDM	
149.35	149.35	98.70	88.31	36.36	67.53	cancer
161.31	154.61	76.79	67.86	27.38	55.95	asia
184.01	184.01	112.50	98.90	40.44	66.91	survey
291.77	283.59	129.22	108.44	38.07	51.65	sachs
242.25	239.06	83.37	63.31	30.77	47.78	child
272.07	266.47	40.74	29.85	20.23	35.88	alarm
412.49	245.47	21.32	21.89	9.40	26.76	win95pts
329.88	327.65	51.07	27.94	9.45	17.47	insurance
343.07	342.14	171.08	143.23	58.86	70.45	hepar2
312.66	303.61	49.52	33.63	18.27	38.20	andes
340.31	337.86	45.20	14.13	9.86	16.96	hailfinder
264.54	264.54	11.32	3.79	−6.31	9.58	pigs
390.78	390.58	27.39	4.57	−23.36	−22.01	water
263.39	261.70	10.99	−5.66	−27.12	−22.23	munin1
322.50	317.04	−19.18	−26.98	−30.53	−18.11	link
241.63	240.63	25.95	8.53	−20.87	−14.97	munin2
238.27	237.85	28.34	10.80	−20.04	−14.06	munin3
429.55	370.87	−48.60	−63.05	−40.49	−39.84	pathfinder
252.88	251.91	18.81	1.55	−24.04	−18.72	munin4
254.53	250.29	17.31	0.34	−24.25	−18.95	munin
230.45	228.87	59.95	35.08	28.25	28.47	barley
167.13	167.13	−63.48	−74.85	−71.08	−70.56	diabetes
112.44	112.43	−84.86	−88.29	−90.25	−90.21	mildew

FIGURE 10 Memory sizes for *bnlearn* networks [Color figure can be viewed at wileyonlinelibrary.com]

indices to store. In these cases, it would be appropriate to select the most repeated value as the default, but this would complicate the combination and marginalization operations.

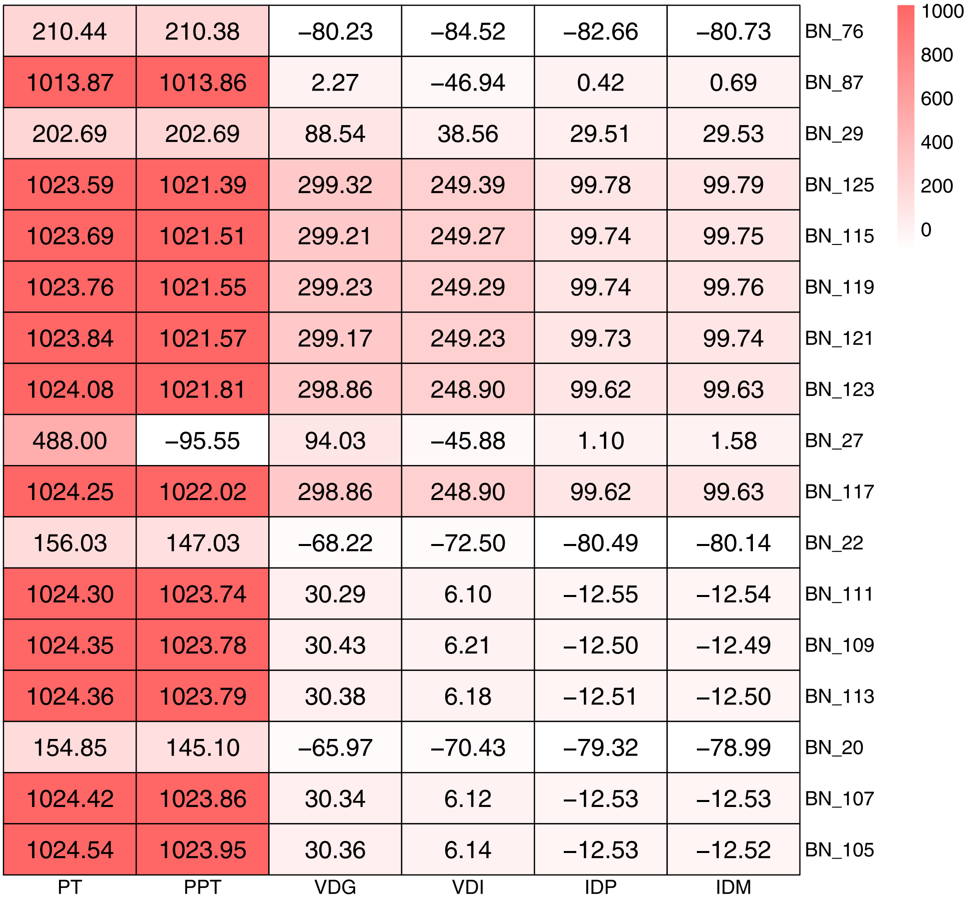
- More important savings correspond to **diabetes** and **mildew**. In these two networks there are several potentials with a high number of repeated values. For **diabetes** there are 25 potentials with 7056 parameters, but only 44 different values. The same circumstance arises in **mildew** (two examples are a potential with 39,040 possible values but only 1756 different values, and another one with 201,300 parameters and 4508 different values). In these potentials, repeated values cannot be collapsed when using PPTs.
- VDI is the best representation for **pathfinder** and **diabetes**. This is explained by the low number of different values compared 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.

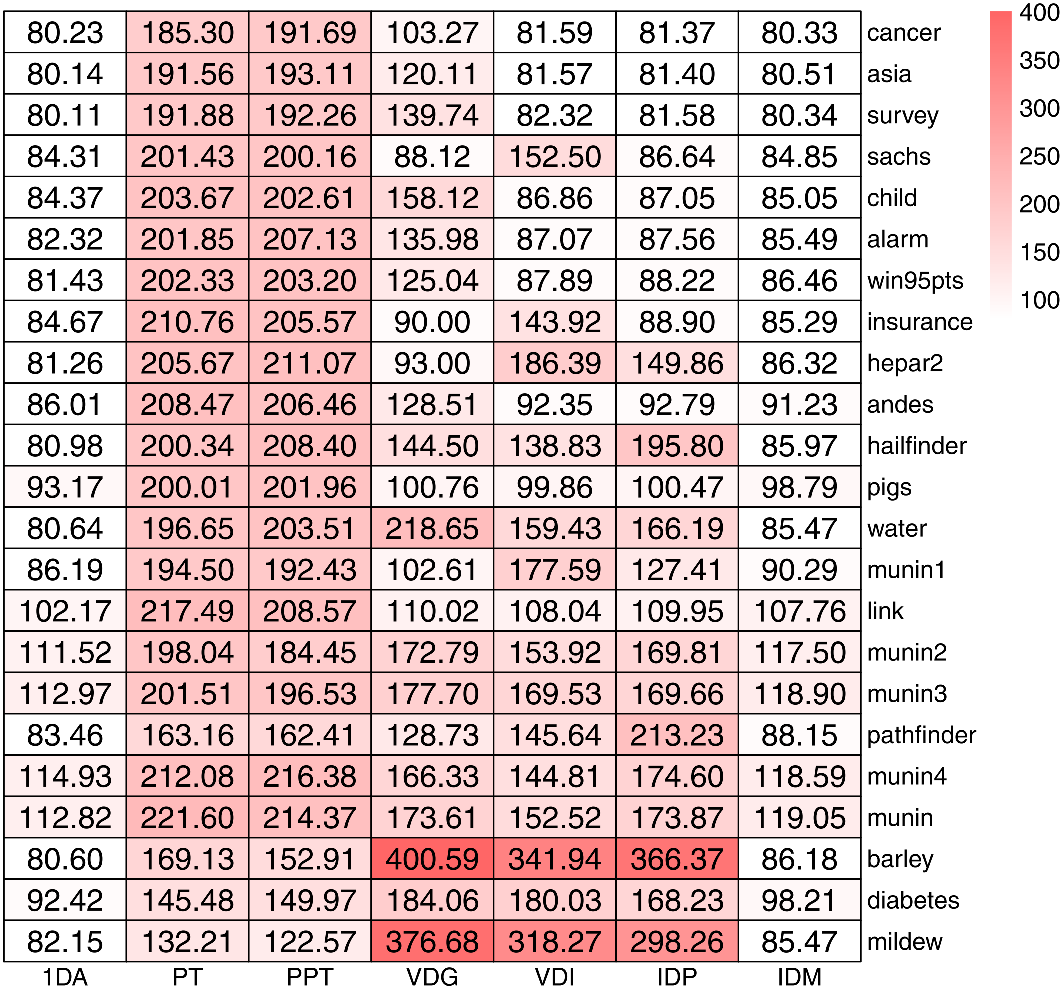
The following conclusions can be outlined from these results:

- Percentages for PTs and PPTs are always over 200%, except for **BN_27**. For 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 only 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**, and moderate savings for **BN_111**, **BN_109**, **BN_113**, **BN_107**, and **BN_106**. For the rest of the network, the percentages of increase with respect to 1DA are the lowest ones.
- All the proposed representations offer a competitive alternative to PTs and PPTs. In general, the most significant savings are found in networks with a very high number of parameters, where efficient representations are paramount for applying inference algorithms.



PT	PPT	VDG	VDI	IDP	IDM	
210.44	210.38	−80.23	−84.52	−82.66	−80.73	BN_76
1013.87	1013.86	2.27	−46.94	0.42	0.69	BN_87
202.69	202.69	88.54	38.56	29.51	29.53	BN_29
1023.59	1021.39	299.32	249.39	99.78	99.79	BN_125
1023.69	1021.51	299.21	249.27	99.74	99.75	BN_115
1023.76	1021.55	299.23	249.29	99.74	99.76	BN_119
1023.84	1021.57	299.17	249.23	99.73	99.74	BN_121
1024.08	1021.81	298.86	248.90	99.62	99.63	BN_123
488.00	−95.55	94.03	−45.88	1.10	1.58	BN_27
1024.25	1022.02	298.86	248.90	99.62	99.63	BN_117
156.03	147.03	−68.22	−72.50	−80.49	−80.14	BN_22
1024.30	1023.74	30.29	6.10	−12.55	−12.54	BN_111
1024.35	1023.78	30.43	6.21	−12.50	−12.49	BN_109
1024.36	1023.79	30.38	6.18	−12.51	−12.50	BN_113
154.85	145.10	−65.97	−70.43	−79.32	−78.99	BN_20
1024.42	1023.86	30.34	6.12	−12.53	−12.53	BN_107
1024.54	1023.95	30.36	6.14	−12.53	−12.52	BN_105

FIGURE 11 Memory sizes for *UAI* networks [Color figure can be viewed at wileyonlinelibrary.com]



1DA	PT	PPT	VDG	VDI	IDP	IDM	
80.23	185.30	191.69	103.27	81.59	81.37	80.33	cancer
80.14	191.56	193.11	120.11	81.57	81.40	80.51	asia
80.11	191.88	192.26	139.74	82.32	81.58	80.34	survey
84.31	201.43	200.16	88.12	152.50	86.64	84.85	sachs
84.37	203.67	202.61	158.12	86.86	87.05	85.05	child
82.32	201.85	207.13	135.98	87.07	87.56	85.49	alarm
81.43	202.33	203.20	125.04	87.89	88.22	86.46	win95pts
84.67	210.76	205.57	90.00	143.92	88.90	85.29	insurance
81.26	205.67	211.07	93.00	186.39	149.86	86.32	hepar2
86.01	208.47	206.46	128.51	92.35	92.79	91.23	andes
80.98	200.34	208.40	144.50	138.83	195.80	85.97	hailfinder
93.17	200.01	201.96	100.76	99.86	100.47	98.79	pigs
80.64	196.65	203.51	218.65	159.43	166.19	85.47	water
86.19	194.50	192.43	102.61	177.59	127.41	90.29	munin1
102.17	217.49	208.57	110.02	108.04	109.95	107.76	link
111.52	198.04	184.45	172.79	153.92	169.81	117.50	munin2
112.97	201.51	196.53	177.70	169.53	169.66	118.90	munin3
83.46	163.16	162.41	128.73	145.64	213.23	88.15	pathfinder
114.93	212.08	216.38	166.33	144.81	174.60	118.59	munin4
112.82	221.60	214.37	173.61	152.52	173.87	119.05	munin
80.60	169.13	152.91	400.59	341.94	366.37	86.18	barley
92.42	145.48	149.97	184.06	180.03	168.23	98.21	diabetes
82.15	132.21	122.57	376.68	318.27	298.26	85.47	mildew

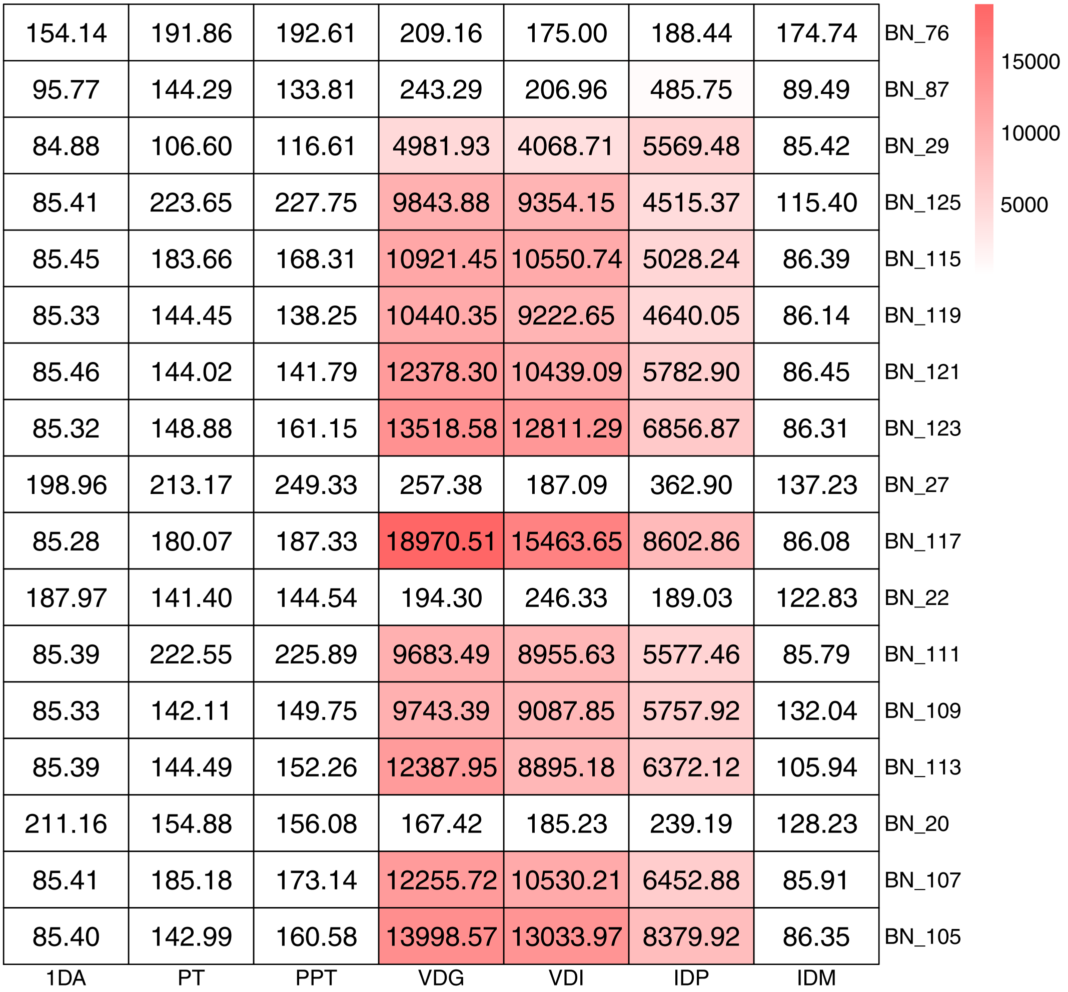
FIGURE 12 Access times for *bnlearn* networks [Color figure can be viewed at wileyonlinelibrary.com]

6.3 | Access time

Although the treatment of complex models implies the need of alternative models with higher computation times (this is assumed as a tradeoff for saving memory space), it is important to take into account the speed of access to potential values and the efficiency of the operations required for the inference tasks. Complex model representations with long computation times would be totally impractical.

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. The experiment is based on a random selection of 10,000 pairs (potential, index). This set will be used for all the representations of each network. The tables show the results with times in milliseconds.

To do a reliable estimation of access times, we have used the **Scalameter** library.[47] This tool allows to configure the time measurement experiments ensuring that the machine reaches a steady state (after a warm-up phase); after that, it repeats several times the procedure of interest and finally reports the average time.



1DA	PT	PPT	VDG	VDI	IDP	IDM	
154.14	191.86	192.61	209.16	175.00	188.44	174.74	BN_76
95.77	144.29	133.81	243.29	206.96	485.75	89.49	BN_87
84.88	106.60	116.61	4981.93	4068.71	5569.48	85.42	BN_29
85.41	223.65	227.75	9843.88	9354.15	4515.37	115.40	BN_125
85.45	183.66	168.31	10921.45	10550.74	5028.24	86.39	BN_115
85.33	144.45	138.25	10440.35	9222.65	4640.05	86.14	BN_119
85.46	144.02	141.79	12378.30	10439.09	5782.90	86.45	BN_121
85.32	148.88	161.15	13518.58	12811.29	6856.87	86.31	BN_123
198.96	213.17	249.33	257.38	187.09	362.90	137.23	BN_27
85.28	180.07	187.33	18970.51	15463.65	8602.86	86.08	BN_117
187.97	141.40	144.54	194.30	246.33	189.03	122.83	BN_22
85.39	222.55	225.89	9683.49	8955.63	5577.46	85.79	BN_111
85.33	142.11	149.75	9743.39	9087.85	5757.92	132.04	BN_109
85.39	144.49	152.26	12387.95	8895.18	6372.12	105.94	BN_113
211.16	154.88	156.08	167.42	185.23	239.19	128.23	BN_20
85.41	185.18	173.14	12255.72	10530.21	6452.88	85.91	BN_107
85.40	142.99	160.58	13998.57	13033.97	8379.92	86.35	BN_105

FIGURE 13 Access times for *UAI* networks [Color figure can be viewed at wileyonlinelibrary.com]

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 1DA-like times. The times for the other alternatives are in most cases shorter than those required for PT and PPT, except for **barley** and **mildew**. For 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 higher 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 study is to investigate the possibilities of using VBPs in inference algorithms with Bayesian networks, the marginalization and combination operations must be defined on these structures.

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

$$\phi^{\downarrow \mathbf{Z}}(\mathbf{z}) = \sum_{\mathbf{x}^{\downarrow \mathbf{Z}} = \mathbf{z}} \phi(\mathbf{x}), \quad \forall \mathbf{z} \in \Omega_{\mathbf{Z}} \tag{20}$$

where $\mathbf{x}^{\downarrow \mathbf{Z}}$ denotes the projection of configuration $\mathbf{x}$ on $\mathbf{Z}$. This operation can be done by iteratively marginalizing out each variable $Y \in \mathbf{X} \backslash \mathbf{Z}$.

In the case the of combination operation, given two potentials $\phi_1(\mathbf{X})$ and $\phi_2(\mathbf{Y})$, the combination of ϕ_1 and ϕ_2 is the potential denoted by $\phi_1 \otimes \phi_2$, which is defined on $\mathbf{Z} = \mathbf{X} \cup \mathbf{Y}$ using pointwise multiplication:

$$\phi_1 \otimes \phi_2(\mathbf{z}) = \phi_1(\mathbf{z}^{\downarrow \mathbf{X}}) \cdot \phi_2(\mathbf{z}^{\downarrow \mathbf{Y}}), \quad \forall \mathbf{z} \in \mathbf{X} \cup \mathbf{Y} \tag{21}$$

We have developed simple and direct algorithms for the 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), so the access operation is extremly important. Since there is a one-to-one correspondence between indices and configurations, we refer to them interchangeably (e.g., index l on $\mathbf{Z}$ corresponds to configuration $\mathbf{z}_l$).

Algorithm 5 (Marginalization in VBPs). Given $VBP_\phi(\mathbf{X})$, a potential defined over $\mathbf{X}$ and $Y \in \mathbf{X}$, the method for marginalizing out Y from VBP_ϕ is described in Algorithm 5. It must be noted that this algorithm can be used for all VBP alternatives.

Algorithm 5 Marginalization of $\boldsymbol{Y}$ from $\boldsymbol{VBP_\phi(\mathbf{X})}$

```
function MARGINALIZE ($VBP_\phi$, $Y$)
    $\mathbf{Z} = \mathbf{X} \backslash Y$                                      ▷ make result domain
    creates $VBP_{\phi_r}(\mathbf{Z})$                                 ▷ empty result potential
    for each $l = \{0 \ldots k\}$, $k = |\Omega_\mathbf{Z}| - 1$ do              ▷ loop over $VBP_\phi(\mathbf{Z})$ indices
        for each $y \in \Omega_Y$ do
            $\mathbf{x}_{ly} \leftarrow \mathbf{z}_l^{\uparrow Y=y}$                  ▷ get configuration $\mathbf{x}_{ly}$ compatible with $\mathbf{z}_l$
            $\mathbf{v}_{ly} \leftarrow VBP_\phi(\mathbf{x}_{ly})$                     ▷ value in $VBP_\phi(\mathbf{X})$ for $\mathbf{x}_{ly}$
        end for
        $VBP_{\phi_r}(\mathbf{z}_l) \leftarrow \sum_{y \in \Omega_y} \mathbf{v}_{ly}$
    end for
    return $VBP_{\phi_r}(\mathbf{Z})$
end function
```

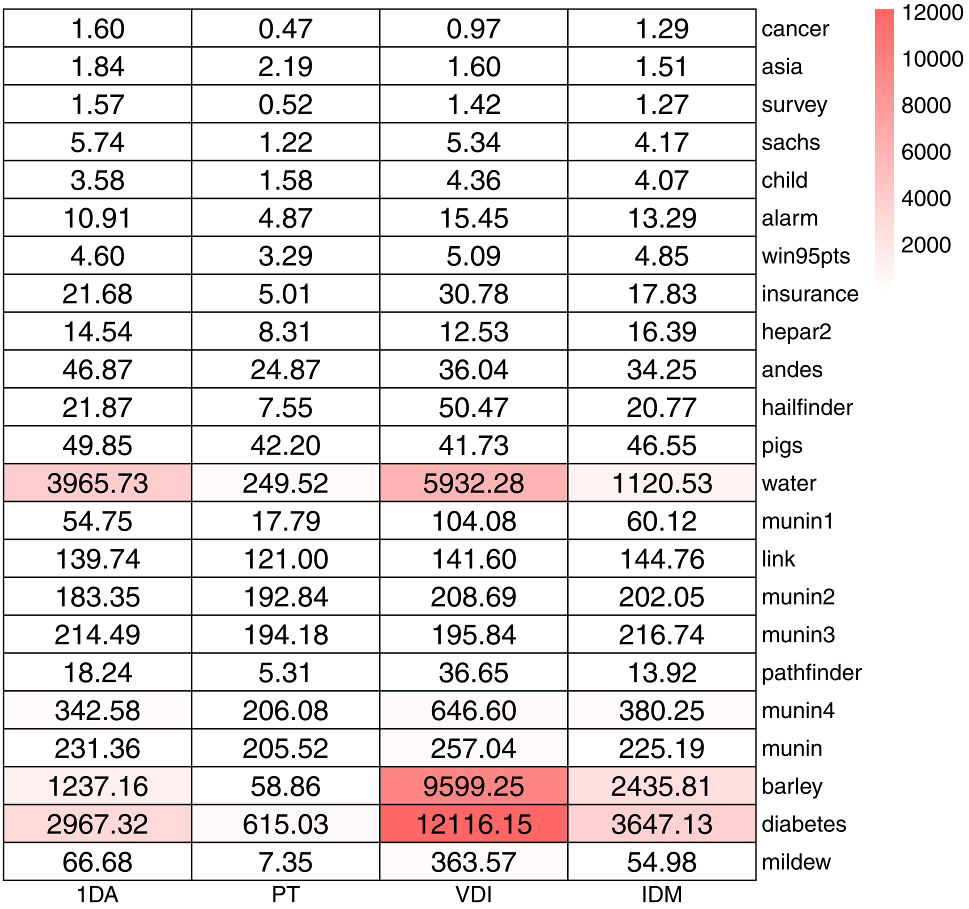
Algorithm 5 removes a variable Y from $VBP_{\phi}(\mathbf{X})$. In Lines 2 and 3, the final potential domain $\mathbf{Z} = \mathbf{X} \backslash Y$ is used to create the resulting potential, which will be empty initially. Line 4 iterates over indices $VBP_{\phi_r}(\mathbf{Z})$. Let us assume that l corresponds to a specific configuration of variables in $\mathbf{Z}$ (denoted by $\mathbf{z}_l$). Compatible indices $\mathbf{x}_l$ refer to configurations produced by completing $\mathbf{z}_l$ with the possible values of Y. This operation is denoted as $\mathbf{z}_l^{\uparrow Y}$. An internal loop (Lines 5 to 8) iterates over the Y values. The sum of $\mathbf{v}_{ly}$ values is assigned to the resulting potential (Line 9).

> **Algorithm 6** (Combination in VBPs). Given $VBP_{\phi_1}(\mathbf{X})$ and $VBP_{\phi_2}(\mathbf{Y})$, 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 described previously: VDG, VDI, IDP, and IDM.

Algorithm 6 Combination of $\boldsymbol{VBP_{\phi_1}}(\mathbf{X})$ and $\boldsymbol{VBP_{\phi_2}}(\mathbf{Y})$

```
1: function COMBINE ($VBP_{\phi_1}(\mathbf{X})$, $VBP_{\phi_2}(\mathbf{Y})$)
2:     $\mathbf{Z} = \mathbf{X} \cup \mathbf{Y}$                                   ▷ make result domain
3:     creates $VBP_{\phi_r}(\mathbf{Z})$                             ▷ empty result potential
4:     for each $l = \{0 \dots k\}$, $k = |\Omega_{\mathbf{Z}}| - 1$ do     ▷ loop over indices $VBP_{\phi}(\mathbf{Z})$
5:         $v_l \leftarrow 0.0$
6:         $\mathbf{x}_l \leftarrow \mathbf{z}_l^{\downarrow \mathbf{X}}$                              ▷ project index $\mathbf{z}_l$ on $\mathbf{X}$
7:         $v_1 \leftarrow VBP_{\phi_1}(\mathbf{x}_l)$                    ▷ value in $VBP_{\phi_1}(\mathbf{X})$ for $\mathbf{x}_l$
8:         if $v_1 \neq 0.0$ (default value) then
9:             $\mathbf{y}_l \leftarrow \mathbf{z}_l^{\downarrow \mathbf{Y}}$                          ▷ project index $\mathbf{z}_l$ on $\mathbf{Y}$
10:            $v_2 \leftarrow VBP_{\phi_2}(\mathbf{y}_l)$                ▷ value in $VBP_{\phi_2}(\mathbf{Y})$ for $\mathbf{y}_l$
11:            if $v_2 \neq 0.0$ then
12:                $v_l = v_1 \cdot v_2$
13:            end if
14:        end if
15:        $VBP_{\phi_r}(\mathbf{z}_l) \leftarrow v_l$
16:    end for
17:    return $VBP_{\phi_r}(\mathbf{Z})$
18: end function
```

Algorithm 6 combines two potentials, $VBP_{\phi_1}(\mathbf{X})$ and $VBP_{\phi_2}(\mathbf{Y})$. Line 2 produces the domain $\mathbf{Z}$ as $\mathbf{Z} = \mathbf{X} \cup \mathbf{Y}$. This is the domain of the potential $VBP_{\phi_r}(\mathbf{Z})$ that must be returned. Line 4 iterates over indices $VBP_{\phi_r}(\mathbf{Z})$. Let l be the index under consideration (it corresponds to a given configuration $\mathbf{z}_l$). The value that is going to be assigned to l is initialized to 0.0 (Line 5). Configuration $\mathbf{z}_l$ must be projected into $VBP_{\phi_1}(\mathbf{X})$ (Line 6) and $VBP_{\phi_2}(\mathbf{Y})$ (Line 9). These operations are denoted by $\mathbf{z}_l^{\downarrow \mathbf{X}}$ and $\mathbf{z}_l^{\downarrow \mathbf{Y}}$, and entail removing from $\mathbf{z}_l$ those variables values that do not belong to $\mathbf{X}$ and $\mathbf{Y}$, respectively. The index of configuration $\mathbf{x}_l$ is used to get v_1 (Line 7). If v_1 is 0.0, then $v_l = 0.0$ for sure, so no more operations are required. Otherwise, $VBP_{\phi_2}(\mathbf{y}_l)$ must be accessed as well (see Line 10). Finally, v_l is assigned to $VBP_{\phi_r}(\mathbf{Z})$ in Line 15.



1DA	PT	VDI	IDM	
1.60	0.47	0.97	1.29	cancer
1.84	2.19	1.60	1.51	asia
1.57	0.52	1.42	1.27	survey
5.74	1.22	5.34	4.17	sachs
3.58	1.58	4.36	4.07	child
10.91	4.87	15.45	13.29	alarm
4.60	3.29	5.09	4.85	win95pts
21.68	5.01	30.78	17.83	insurance
14.54	8.31	12.53	16.39	hepar2
46.87	24.87	36.04	34.25	andes
21.87	7.55	50.47	20.77	hailfinder
49.85	42.20	41.73	46.55	pigs
3965.73	249.52	5932.28	1120.53	water
54.75	17.79	104.08	60.12	munin1
139.74	121.00	141.60	144.76	link
183.35	192.84	208.69	202.05	munin2
214.49	194.18	195.84	216.74	munin3
18.24	5.31	36.65	13.92	pathfinder
342.58	206.08	646.60	380.25	munin4
231.36	205.52	257.04	225.19	munin
1237.16	58.86	9599.25	2435.81	barley
2967.32	615.03	12116.15	3647.13	diabetes
66.68	7.35	363.57	54.98	mildew

FIGURE 14 Variable elimination times for *bnlearn* networks [Color figure can be viewed at wileyonlinelibrary.com]

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

In any case, it is important to highlight that the goal of these experiments is to get an overall idea of the behavior of VBP structures. A foreseen line of work will be to carry out specific implementations of the marginalization and combination operations, taking 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 show the best tradeoff between memory use and access times within their category. They are compared with 1DA and PT (when using trees, there is not much difference between PT and PPT in general). We have also used the **Scalameter** library for measuring computation times. The results for this section are presented in Figure 14.

Regarding these results, it is observed that the inference with PT shows the higher efficiency. This is due to the specific implementations of marginalization and combination operations in PTs (see Reference [22]). The implementation of these methods is recursive. It should be noted that for some of the *UAI* networks, the execution of the algorithm produces a stack overflow error when generating PTs with many variables. 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 suggests that more refined implementations of the marginalize and combine operations, fitted to their structure, will improve current computation times. More efficient implementations of these operations should try to avoid iterating over all indices of the resulting potential, using instead only those which are stored. For some networks, this can offer significant time reductions.

7 | CONCLUSION

Regarding the use of memory space, it is observed that all the alternatives proposed offer competitive results compared to 1DA, PT and PPT. For 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. In our opinion, these results are promising. We also think that more efficient implementations will produce better results. This task will be the addressed in future research.

Another important feature of VBPs (which has not been used in this study) should be noted: VBPs can be approximated. The approximation operation assumes a 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 reducing the number of values that must be stored. With this operation, any algorithm involving the use of approximate potentials will become approximate as well, and will ultimately offer nonexact solutions. This is helpful, since for very complex problems it is always better to have at least one approximate solution (see References [23–26] as examples of approximation with PTs).

The software used in this paper was implemented in **Scala**. The code is available in https://github.com/mgomez-olmedo/VBPots and the materials also include the information required to reproduce the experiments. The functional programming paradigm combined with object orientation that **Scala** offers can be used to parallelize well-defined operations on multicore CPUs when possible. Some of these benefits were studied in Reference [48].

ACKNOWLEDGMENTS

This study 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). Funding for open access charge from Universidad de Granada/CBUA.

CONFLICT OF INTERESTS

The authors declare that there are no conflict of interests.

AUTHOR CONTRIBUTIONS

All the results and contents presented in this study have been developed jointly by all the authors.

ORCID

Manuel Gómez-Olmedo https://orcid.org/0000-0002-3817-8723
Rafael Cabañas https://orcid.org/0000-0002-5034-582X
Andrés Cano https://orcid.org/0000-0003-1733-9441
Serafín Moral https://orcid.org/0000-0002-5555-0857
Ofelia P. Retamero https://orcid.org/0000-0002-6521-470X

REFERENCES

1. Pearl J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA: Morgan Kaufmann; 1988.
2. Lauritzen SL. *Graphical Models*. New York, NY: Oxford University Press; 1996.
3. Koller D, Friedman N. *Probabilistic Graphical Models: Principles and Techniques*. Cambridge, MA: MIT Press; 2009.
4. Pearl J. *Bayesian Networks: A Model of Self-Activated Memory for Evidential Reasoning*. Los Angeles, CA: Computer Science Department, University of California; 1985.
5. Pearl J, Russell S. *Bayesian Networks*. Los Angeles, CA: Computer Science Department, University of California; 1998.
6. Olmsted SM. *On representing and solving decision problems*. PhD Thesis, Department of Engineering-Economic Systems, Stanford University; 1983.
7. Howard RA, Matheson JE. Influence diagram retrospective. *Decision Anal*. 2005;2(3):144-147.
8. David AP. Applications of a general propagation algorithm for probabilistic expert systems. *Stat Comput*. 1992;2:25-36. https://doi.org/10.1007/BF01890546
9. Kwisthout J. Most probable explanations in Bayesian networks: Complexity and tractability. *Int J Approx Reason*. 2011;52(9):1452-1469. https://doi.org/10.1016/j.ijar.2011.08.003
10. Dagum P, Luby M. An optimal approximation algorithm for Bayesian inference. *Artif Intell*. 1997;93(1):1-27.
11. Dechter R. Bucket elimination: A unifying framework for probabilistic inference. In: *Learning in graphicalmodels*, Springer; 1998:75-104.
12. Jensen CS, U, Kjærulff, Kong A. Blocking Gibbs sampling in very large probabilistic expert systems. *Int J Human-Comput Studies*. 1995;42(6):647-666.
13. Jensen FV, Nielsen TD. *Bayesian Networks and Decision Graphs*, Springer Verlag; 2007.
14. Li Z, D'Ambrosio B. Efficient inference in Bayes networks as a combinatorial optimization problem. *Int J Approx Reason*. 1994;11(1):55-81.
15. Madsen AL, Jensen FV. Lazy evaluation of symmetric Bayesian decision problems. In: *Proceedings of the 15th Conference on Uncertainty in AI*, Morgan Kaufmann Publishers Inc.; 1999:382-390.
16. Madsen AL, Jensen FV. Lazy propagation: a junction tree inference algorithm based on lazy evaluation. *Artif Intell*. 2004;113(1-2):203-245.
17. Pearl J. Evidential reasoning using stochastic simulation of causal models. *Artif Intell*. 1987;32(2):245-257.
18. Shachter RD. Evaluating influence diagrams. *Operat Res*. 1986;34(6):871-882.
19. Shachter RD, D'Ambrosio B, Del Favero B. Symbolic probabilistic inference in belief networks. In: *AAAI Proceedings*; 1990:126-131.
20. Boutilier C, Friedman N, Goldszmidt M, Koller D. Context-specific independence in Bayesian networks. In: *Proceedings of the 12th International Conference on Uncertainty in Artificial Intelligence*, Morgan Kaufmann Publishers Inc.; 1996:115-123.
21. Cano A, Moral S, Salmerón A. Penniless propagation in join trees. *Int J Intell Syst*. 2000;15(11):1027-1059.
22. Salmerón A, Cano A, Moral S. Importance sampling in Bayesian networks using probability trees. *Comput Stat Data Anal*. 2000;34(4):387-413.

23. Gómez-Olmedo M, Cano A. Applying numerical trees to evaluate asymmetric decision problems. In: Nielsen T, Zhang N, eds. *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*. Lecture Notes in Computer Science. Vol. 2711. Berlin, Heidelberg: Springer; 2003. https://doi.org/10.1007/978-3-540-45062-7_16
24. Cano A, Gómez-Olmedo M, Moral S. Approximate inference in Bayesian networks using binary probability trees. *Int J Approx Reason*. 2011;52(1):49-62.
25. Cabañas R, Gómez M, Cano A. Approximate inference in influence diagrams using binary trees. In: *Proceedings of the Sixth European Workshop on Probabilistic Graphical Models (PGM-12)*; 2012.
26. Cabañas R, Gómez-Olmedo M, Cano A. Using binary trees for the evaluation of influence diagrams. *Int J Uncertain Fuzziness Knowl-Based Syst*. 2016;24(1):59-89.
27. Arias M, Díez F. Operating with potentials of discrete variables. *Int J Approx Reason*. 2007;46(1):166-187.
28. Heckerman D, Mamdani A, Wellman MP. Real-world applications of Bayesian networks. In: *Association for Computing Machinery*; 1995. https://doi.org/10.1145/203330.203334
29. Gómez-Olmedo M. Real-World applications of influence diagrams. In: Gámez JA, Moral S, Salmerón A, eds. *Advances in Bayesian Networks. Studies in Fuzziness and Soft Computing*. Berlin, Heidelberg: Springer; 2004. https://doi.org/10.1007/978-3-540-39879-0_9
30. Díez FJ, Luque M, Arias M, Pérez-Martín J. Cost-effectiveness analysis with unordered decisions. *Artif Intell Med*. 2021;117. https://doi.org/10.1016/j.artmed.2021.102064
31. Wong TL, Xie H, Lam W, Wang FL. A learning framework for information block search based on probabilistic graphical models and Fisher Kernel. *Int J Mach Learn Cybernet*. 2018;6:1473-1487. https://doi.org/10.1007/s13042-017-0657-9
32. Yang L, Guo Y. Combining pre- and post-model information in the uncertainty quantification of non-deterministic models using an extended Bayesian melding approach. *Inform Sci*. 2019;502:146-163. https://doi.org/10.1016/j.ins.2019.06.029
33. Lee W, Zabaras N. Parallel probabilistic graphical model approach for nonparametric Bayesian inference. *J Computat Phys*. 2018;372(1):546-563. https://doi.org/10.1016/j.jcp.2018.06.057
34. Alaa AM, van der Schaar M. Bayesian nonparametric causal inference: Information rates and learning algorithms. *IEEE J Selected Topics Signal Process*. 2018;12(5):1031-1046. https://doi.org/10.1109/JSTSP.2018.2848230
35. Chavira M, Darwiche A. Compiling Bayesian networks using variable elimination. In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence*; 2007:2443-2449.
36. Sanner S, McAllester D. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence*; 2005:1384-1390.
37. Choi A, Kisa D, Darwiche A. Compiling probabilistic graphical models using sentential decision diagrams. In: *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, Springer; 2013:121-132.
38. Oztok U, Darwiche A. A top-down compiler for sentential decision diagrams. In: *Proceedings of the 24th International Conference on Artificial Intelligence*; 2015:3141-3148.
39. Cano A, Gómez-Olmedo M, Moral S, Pérez-Ariza C, Salmerón A. Recursive probability trees for Bayesian networks. In: *Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2009: Current Topics in Artificial Intelligence*; 2012. https://doi.org/10.1007/978-3-642-14264-2_25
40. Cano A, Gómez-Olmedo M, Moral S, Pérez-Ariza C, Salmerón A. Learning recursive probability trees from probabilistic potentials. *Int J Approx Reason*. 2012;53(9):1367-1387. https://doi.org/10.1016/j.ijar.2012.06.026
41. Scutari M. Bayesian network constraint-based structure learning algorithms: Parallel and optimized implementations in the bnlearn R package. *J Stat Software*. 2017;77(2). https://doi.org/10.18637/jss.v077.i02
42. Scutari M. Learning Bayesian networks with the bnlearn R package. *J Stat Software*. 2010;35(3). https://doi.org/10.18637/jss.v035.i03
43. UAI 2016 Inference Competition. 2016. http://www.hlt.utdallas.edu/~vgogate/uai16-evaluation/
44. UAI 2014 Inference Competition. 2016. http://www.hlt.utdallas.edu/~vgogate/uai14-competition/index.html

45. Shenoy P, Shafer GR. Axioms for probability and belief-function propagation. In: Shachter RD, Levitt TS, Kanal LN, Lemmer JF, eds. Uncertainty in Artificial Intelligence, Vol. 9 of Machine Intelligence and Pattern Recognition, 1990. https://doi.org/10.1016/B978-0-444-88650-7.50019-6
46. Zhang NL, Poole D. Exploiting causal independence in Bayesian network inference. *J Artif Intell Res*. 1996; 5:301-328.
47. Scalameter: Automate your performance testing today; 2021. http://www.github.com/deepfakes/
48. Masegosa AR, Martinez AM, Borchani H. Probabilistic graphical models on multi-core cpus using Java 8. *IEEE Computat Intell Magazine*. 2016:11(2):41-54.

How to cite this article: Gómez-Olmedo M, Cabañas R, Cano A, Moral S, Retamero OP. Value-based potentials: exploiting quantitative information regularity patterns in probabilistic graphical models. *Int J Intell Syst*. 2021;1-31.
https://doi.org/10.1002/int.22573