Speeding up the execution of a large number of statistical tests of independence

F. Schlüter, F. Bromberg, S. Perez

{federico.schluter, fbromberg, dsperez}@frm.utn.edu.ar
DHARMa Lab, Desarrollo Herramientas Aprendizaje y Razonamiento de Máquinas.
Dept. Sistemas de Información, Facultad Regional Mendoza,
Universidad Tecnológica Nacional, Mendoza, Argentina

Abstract. A massive amount of conditional independence tests on data must be performed in the problem of learning the structure of probabilistic graphical models when using the independence-based approach. An intermediate step in the computation of independence tests is the construction of contingency tables from the data. In this work we present an intelligent cache of contingency tables that allows the tables stored to be reused not only for the same test, in the not uncommon case that the test must be performed again, but for an exponential number of other tests, all those involving a subset of the variables of the test stored. In practice, however, not so many tests actually reuse the tables stored. We show results when testing the cache with IBMAP-HC, a recently proposed algorithm for learning the structure of Markov networks, a.k.a. undirected graphical models. The experiments show that in all cases, above 95% of the running time spent by IBMAP-HC in reading data is saved by the cache. The savings in running time for IBMAP-HC were up to 80% for datasets above 40,000 datapoints.

Keywords: Statistical tests of independence, contingency tables, probabilistic graphical models, structure learning.

1 Introduction

This work proposes novel practices for speeding up the execution of a large number of statistical independence tests (hereon independence tests, statistical tests, or simply tests). Statistical tests are procedures developed by statisticians for determining dependences between random variables through an exploratory analysis of data. An intermediate step in the computation of independence tests is the construction of *contingency tables* from data, which requires a scan of the whole input dataset. Our contribution is a cache over these tables for saving the computational cost of constructing them from data.

In probability theory, two variables X and Y are said to be *independent* (*dependent*) if the conditional distribution of X given Y matches that of X alone, i.e., if $\Pr(X \mid Y) = \Pr(X)$. This means that the distribution of X is unaffected by the knowledge that Y takes a certain value. If the above doesn't

hold, the two variables are said to be dependent. Two variables X and Y are said to be *conditionally independent* given a set of variables $\mathbf{Z}$ (or alternatively we say triplet $(X, Y \mid \mathbf{Z})$ is independent) if the distribution of X given Y and given $\mathbf{Z}$ matches that of X given $\mathbf{Z}$, i.e., if $\Pr(X \mid Y, \mathbf{Z}) = \Pr(X \mid \mathbf{Z})$. For dependence, the distributions do not match, i.e., $\Pr(X \mid Y, \mathbf{Z}) \neq \Pr(X \mid \mathbf{Z})$. Intuitively, the independence means that any knowledge of the value taken by the variables in $\mathbf{Z}$ renders any knowledge of the value taken by Y completely uninformative about the value taken by X. There are many different statistical tests for assessing independences among random variables. The most commonly used are the Pearson's χ^2 and the G^2 tests [1], the mutual information test [8], and more recently the *Bayesian test* [10, 11]. All these tests requires the computation of contingency tables for analyzing the relation between variables, and thus can benefit from the speed up introduced in this work.

Contingency tables record the frequency distribution of the variables in a matrix format, and require reading the whole dataset for computing each frequency (see more in Section 1.2). The novelty of our approach consists in a procedure that makes it possible to reuse the contingency table of some test t over any triplet involving variables $\{X, Y\} \cup \mathbf{Z}$ stored in the cache, not only in the trivial case when t has to be performed again, but (through an efficient computation) in any other test t' whose set of variables $\{X', Y'\} \cup \mathbf{Z}'$ is a subset of the variables in t, i.e., $\{X', Y'\} \cup \mathbf{Z}' \subseteq \{X, Y\} \cup \mathbf{Z}$. Since the number of subsets of a set grows exponentially with the size of the set, the tables of one test in the cache has the potential of being reusable for an exponential number of other tests. In practical scenarios, however, it may be the case that not all these exponentially many tests must be performed. In our experiments (c.f. §3) we present results for one possible practical scenario: algorithms for learning *probabilistic graphical models* from data (more in Section 1.1). The experimental results show savings of more than 95% in the time spent constructing contingency tables, and above 80% savings overall. Although inspired by, and designed for the problem of learning graphical models, the speed up approach presented is general and can be applied in principle in any other situation requiring massive computations of statistical independence tests. The authors, unfortunately, are unaware of such other situations at the time of this writing.

The rest of the paper is organized as follows. In the next two subsections 1.1 and 1.2, we discuss some preliminaries, including: Probabilistic graphical models, as well as details on the workings of contingency tables. Section 2 explains in detail our approach, including the two main procedures for constructing tables of some test from the tables of other tests: sub-conditioning and pivoting. Then, in Section 3 we present the results of our experiments. We conclude with a summary of our work and possible directions of future works in Section 4.

1.1 Probabilistic graphical models

In this section we present *Probabilistic graphical models*, the practical scenario chosen for evaluating the actual speed ups reached by using the cache of tests. Probabilistic graphical models provide a general purpose modeling scheme for

exploiting conditional independences among random variables of high dimensional probability distributions. Explicit encoding of these independencies allow a compact representation of the distribution, resulting in sometimes exponential reductions in the space complexity of storing the distribution in memory, in the time complexity of inference computations performed over the distribution (i.e., the computation of conditional, marginal or joint probabilities of interest), and the sample complexity of the data required for learning the distribution. A graphical model for a domain over set $\mathbf{V}$ of random variables is presented in terms of a graph and a set of numerical parameters. The graph, also commonly known as *independence structure* (or simply *structure*), contains $|\mathbf{V}| = n$ nodes, each representing a random variable, and the edges between the nodes encodes statistical dependencies/independencies among the variables. The numerical parameters are tables of real values that, together with the graph, result in a well-formed probabilistic model. *Markov networks* (**MN**s) are a type of graphical models that use an undirected graph for its representation of dependencies (see [12] for more details). Other type of graphical models are the well known *Bayesian networks* (**BN**s), that use a directed acyclic graph for representing the dependencies. MNs can represent certain probability distributions that BNs cannot, and vice versa. Learning graphical models thus consist in first learning its independence structure and, given the structure, learn its numerical parameters. A general strategy for learning the structure of MNs and BNs are independence-based algorithms [14, 2, 5, 11]. These algorithms discover the independence structure by taking a quite natural approach: they execute a sequence of independence tests and use the outcome of those tests, that is, independence statements about the variables of the domain, to infer the structure. An important advantage of these algorithms is that they can converge efficiently to the structure. When the outcomes of the independence tests are fully reliable, i.e., tests always decides correctly on the dependence or independence of variables, these algorithms not only converges efficiently, reaching time complexity in the order of $O(n^2)$ tests, but it can be formally demonstrated that they converge to the correct structure.

In contrast, when tests are not reliable, the structures obtained may be incorrect. In practice, for statistical tests to be reliable they require a number of data points exponential in the number of variables involved in the test. To address this shortcoming, several novel independence-based algorithms were proposed recently that increase the quality of the output structures, at the expense of larger running times. These algorithms improve the quality of their outcomes by correcting errors in the independencies asserted by statistical independence tests. To correct the outcome of some independence test, these algorithms execute extra tests over sets of variables related to the variables in the test of interest (thus the increase in runtime). With the outcomes of these tests they correct errors in the test of interest by resolving constrains – Pearl's axioms of independence [12] – that must be satisfied between the independence/dependence asserted by the test, and independence/dependence asserted by the extra tests. If these independence assertions do not satisfy the constrain, one of them must be wrong

(and can thus be easily corrected by flipping its independence value). Different approaches are taken to decide which of them is wrong. One such approach is presented in [4], which proposes the use of a knowledge base of independence facts that are related through Pearl's axioms. Errors in the independence assertions of statistical tests may result in inconsistencies in the knowledge base. The algorithm then proposes the use of the argumentation framework [3, 9] for inferring independencies/dependencies in this inconsistent knowledge base. Another, more recent example of an algorithm that uses such approach is presented in [6, 13, 7]. This algorithm takes a Bayesian approach, maintaining a posterior probability over possible structures by relaxing the outcome of tests to probabilistic outcomes, namely, posterior probabilities of the variables being independent or dependent given the data. Learning the structure thus reduces to finding the *maximum a posteriori*, that is, the most probable independences structure given the data. The important result is the introduction of an efficient procedure for computing the posterior probability, which is based on the posterior probabilities of independencies. This is achieved through Margaritis' test of independence [10, 11], also known as the *Bayesian test*.

The above algorithms are effective in producing better quality outputs (i.e., closer to the true underlying structure), but at the expense of performing a massively larger number of tests, with important increases in running time. All these tests, however, are performed over different subsets of the same set of variables, i.e., the variables in the domain. Many of these tests have a chance of being subsets of each other, thus reusable from the cache. Experimental results presenting the savings obtained are reported in detail in the experimental section (c.f. §3). Before that, let us first describe in detail contingency tables, and how is that they can be reused for tests involving subsets of variables.

1.2 Contingency Tables

We discuss now in detail how the tables of some test are constructed from an input dataset, and then, in the next section, how contingency tables stored in the cache can be reused to compute efficiently the tables of other tests.

In this work we assume a domain $\mathbf{V}$ of discrete random variables, denoted with capital letters, e.g. $\mathbf{V} = \{U, V, W, X, Y, Z\}$. A *dataset* D is a sample of complete configurations of $\mathbf{V}$ given in tabular form, with one column per random variable, and each of the N rows containing the outcome of a random experiment. Each cell at the i-th column contains the value sampled in the experiment for the i-th random variable. An example dataset with $N = 7$ rows, for $n = 4$ binary random variables $\{X, Y, U, V\}$ is shown in Fig. 1 (a).

A contingency table of variables (X, Y) for some dataset D is a double-entry table used for analyzing the unconditional independence between X and Y in the probability distribution of which D is a sample. Its number of rows and columns is $|X|$ and $|Y|$, respectively, where $|X| = |dom(X)|$, $|Y| = |dom(Y)|$ and $dom(X)$, $dom(Y)$ are the domains of X and Y. A cell (x, y), $x \in dom(X), y \in dom(Y)$, represents the number of datapoints in D in which $X = x$ and $Y = y$. Figure 1(b) exemplifies the table for X and Y, computed from the dataset of Fig. 1 (a). So

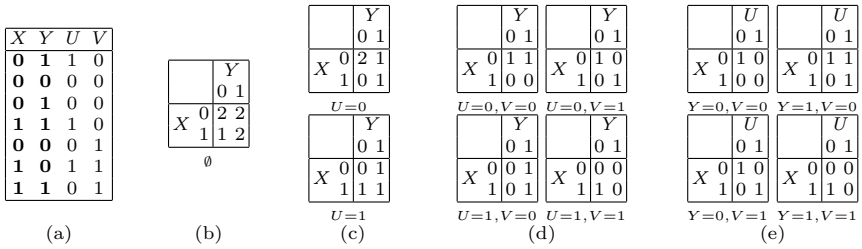

Fig. 1. (a) Example dataset D for binary random variables $\{X, Y, U, V\}$, and example (b) unconditional contingency table for triplet $(X, Y \mid \emptyset)$, (c) for triplet $(X, Y \mid \{U\})$, (d) triplet $(X, Y \mid \{U, V\})$, and (e) triplet $(X, U \mid \{Y, V\})$, all computed from D.

for instance, the top-right cell contains a value 2 because there are 2 rows in the dataset (first and third rows) for which $X = 0$ and $Y = 1$.

Conditional independence of a triplet $t = (X, Y \mid \mathbf{Z})$, that is, between X and Y given a conditioning set $\mathbf{Z}$, is estimated using several contingency tables over X and Y, one per configuration $\mathbf{z}$ of values of $\mathbf{Z}$ (a.k.a. *slice*). A cell (x, y) in a slice $\mathbf{z}$ of t stores the count of datapoints $\mathbf{d}$ in D for which $X = x$, $Y = y$ and $\mathbf{Z} = \mathbf{z}$, denoted $c_{x,y|\mathbf{z}}$. Formally:

$$c_{x,y|\mathbf{z}} = |\{\mathbf{d} \in D \mid X = x \wedge Y = y \wedge \mathbf{Z} = \mathbf{z}\}| \tag{1}$$

The number of slices $\sigma_{\mathbf{Z}}$ of $\mathbf{Z}$ is therefore $\sigma_{\mathbf{Z}} = \prod_{Z \in \mathbf{Z}} |Z|$ slices. Although this grows exponentially, in practice slices with empty tables in them (i.e., all its counts equal to 0) are ignored. The upper bound in $\sigma_{\mathbf{Z}}$ occurs when the tables of each slices has one cell with a count of 1 and all other counts equal to 0. In that case $\sigma_{\mathbf{Z}} = N$ slices. In summary, $\sigma_{\mathbf{Z}} = \max\{N, \prod_{Z \in \mathbf{Z}} |Z|\}$.

Figure 1 (d) shows an example of the tables of triplet $(X, Y \mid U, V)$. Since U and V are binary variables, there are $2 \times 2 = 4$ slices, corresponding to all possible configurations of $\{U, V\}$.

The computation of the tables of a triplet $t = (X, Y \mid \mathbf{Z})$ from dataset D, requires processing each of the N rows in D. This processing involves reading the value that each variable in t takes in that row, a cost of $2 + |\mathbf{Z}|$. The total cost of performing a test is therefore $N(2 + |\mathbf{Z}|)$, a value we call *weighted cost* of the test. Once the conditioning tables are computed, a statistical test analyzes them to decide on the independence or dependence. Due to space restrictions we do not discuss in detail how the different tests perform these calculations. We do however would like to stress that the computational cost of these calculations is a constant time $O(1)$ calculation per count, i.e., assuming the number of values of each variable equals d, the time complexity is $O(d^2 \sigma_{\mathbf{Z}})$. In summary,

$$O(\textit{weighted cost } + d^2 \sigma_{\mathbf{Z}}) = O(N(2 + |\mathbf{Z}|) + d^2 \sigma_{\mathbf{Z}}). \tag{2}$$

2 Our Approach

We explain now our main contribution: the cache for contingency tables. First, we explain the procedure for computing efficiently a contingency table from

the contingency tables of other triplets, avoiding the expensive computation of constructing the tables by scanning the whole dataset. Then, we discuss how this procedure can be make effective in a cache for tables.

2.1 Reusing Contingency Tables

We explain this procedure in two steps: *sub-conditioning* and *pivoting*. Sub-conditioning computes the tables of a triplet $t = (X, Y \mid \mathbf{Z})$ from a *super-conditioning* triplet $t' = (X, Y \mid \mathbf{Z}')$ that satisfies $\mathbf{Z} \subseteq \mathbf{Z}'$. Pivoting computes the tables of t from a *pivot* triplet $t' = (X', Y' \mid \mathbf{Z}')$ that satisfies $\{X, Y\} \cup \mathbf{Z} = \{X', Y'\} \cup \mathbf{Z}'$. It is easy to see that by applying first a pivot and then a sub-conditioning, the tables of a triplet $t' = (X', Y' \mid \mathbf{Z}')$ can be *reused* to obtain the tables of any other triplet $t = (X, Y \mid \mathbf{Z})$ whose set of variables is a subset of the set of variables of t' (with equality requiring only a pivot operation). We say that t is a *sub-triplet* of t' (and t' a *super-triplet* of t). We now proceed to explain in detail the computations involved in sub-conditioning, and later, in Lemma 2, we present the computation of pivoting.

Lemma 1 (Sub-conditioning). *Let $t = (X, Y \mid \mathbf{Z})$ and $t' = (X, Y \mid \mathbf{Z}')$ with $\mathbf{Z} \subseteq \mathbf{Z}'$. Then the counts stored in the tables of t can be computed by the counts in the tables of t' as follows:*

$$c_{x,y|\mathbf{z}} = \sum_{\mathbf{z}'' \in slices(\mathbf{Z}' - \mathbf{Z})} c_{x,y|\mathbf{z} \cup \mathbf{z}''}. \tag{3}$$

The above expression means the counts $c_{x,y|\mathbf{z}}$ at cells in the table of t are equal to the sum of counts at cells (x, y) of all slices $\mathbf{z}'$ of $\mathbf{Z}'$ whose assignments for those variables in $\mathbf{Z}'$ that are also in $\mathbf{Z}$ are those indicated by $\mathbf{z}$.

Before proving the lemma, let us first illustrate the procedure through the example shown in Fig. 1, where $\mathbf{Z} = \{U\}$, whose tables are shown in part (c), and $\mathbf{Z}' = \{U, V\}$, whose tables are shown in part (d). Here, $\mathbf{Z}' - \mathbf{Z} = \{V\}$, and the sum in Eq. (3) goes over $V = 0$ and $V = 1$, and is thus summing the counts $c_{x,y|\mathbf{z} \cup \{V=0\}}$ and $c_{x,y|\mathbf{z} \cup \{V=1\}}$. For instance, the count $c_{X=0,Y=0|\{U=0\}}$ is shown in the upper-left cell of slice $\{U = 0\}$, and equals 2. According to the equation, it is equal to the sum of $c_{X=0,Y=0|\{U=0\} \cup \{V=0\}}$ stored in the upper-left cell of slice $\{U = 0, V = 0\}$, and equals 1; and $c_{X=0,Y=0|\{U=0\} \cup \{V=1\}}$, stored in upper-left cell of slice $\{U = 0, V = 1\}$), and equals 1. These counts are not arbitrary, but corresponds to the dataset of Fig. 1(a). Let us now prove the Lemma.

Proof. From Eq. (1), we have that

$$c_{x,y|\mathbf{z}} = |\{\mathbf{d} \in D \mid X = x \wedge Y = y \wedge \mathbf{Z} = \mathbf{z}\}|$$

and since $\vee_{z'' \in slices(\mathbf{Z}'')} \mathbf{Z}'' = \mathbf{z}''$ is always true (something must be assigned to $\mathbf{Z}''$), we can add it in the conditioning of the above equation without affecting it. We do it for the special case of $\mathbf{Z}'' = \mathbf{Z}' - \mathbf{Z}$ obtaining,

$$c_{x,y|\mathbf{z}} = \left|\left\{\mathbf{d} \in D \mid X = x \wedge Y = y \wedge \mathbf{Z} = \mathbf{z} \wedge \left(\vee_{z'' \in slices(\mathbf{Z}' - \mathbf{Z})} \mathbf{Z}' - \mathbf{Z} = \mathbf{z}''\right)\right\}\right|$$

Distributing $\wedge$ over $\vee$, and properties of sets, the above results in

$$c_{x,y|\mathbf{z}} = \left|\cup_{z'' \in slices(\mathbf{Z}'-\mathbf{Z})} \{\mathbf{d} \in D \mid X = x \wedge Y = y \wedge \mathbf{Z} = \mathbf{z} \wedge \mathbf{Z}' - \mathbf{Z} = \mathbf{z}''\}\right|$$

Noticing that because $\mathbf{Z} \subseteq \mathbf{Z}'$, $\mathbf{Z} \cup (\mathbf{Z}' - \mathbf{Z}) = \mathbf{Z}'$. Also that the cardinality of the union equals the sum of the cardinalities. Therefore,

$$c_{x,y|\mathbf{z}} = \sum_{z'' \in slices(\mathbf{Z}'-\mathbf{Z})} |\{\mathbf{d} \in D \mid X = x \wedge Y = y \wedge \mathbf{Z}' = \mathbf{z} \cup \mathbf{z}''\}|$$

And finally, by the definition of counts in Eq. (1), the terms in the summation are $c_{x,y|\mathbf{z}\cup\mathbf{z}''}$, which proves the lemma. □

The time complexity of sub-conditioning $(X, Y \mid \mathbf{Z}')$ to $(X, Y \mid \mathbf{Z})$ using Eq. (3) is $|slices(\mathbf{Z}' - \mathbf{Z})| = 2^{|\mathbf{Z}'|-|\mathbf{Z}|}$ times the complexity of retrieving the counts in the summation. With $2D$ arrays for the table at each slice, and a hash-table for the slices, both with $O(1)$ for cost of retrieval, the cost for retrieving the counts reduces to computing the key for the hash-table, which is $|\mathbf{Z}'|$. Since there are $2^{|\mathbf{Z}|}$ slices in $(X, Y \mid \mathbf{Z})$, and $|X||Y|$ counts per table, the total cost of sub-conditioning all counts is $\left(d^2 2^{|\mathbf{Z}|}\right)\left(|\mathbf{Z}'|2^{\mathbf{Z}'-\mathbf{Z}}\right) = d^2|\mathbf{Z}'|2^{|\mathbf{Z}'|}$, where we assume all domains has the same number of values d. In practice, many of the slices in $\mathbf{Z}'$ has 0 counts in all its cells, and thus are ignored. A more efficient algorithm would not actually traverse all configurations of $\mathbf{Z}'$ systematically, but would instead traverse all non-empty slices, whose quantity was denoted by $\sigma_{\mathbf{Z}'}$, that is, $\mathbf{Z}'$ into $\mathbf{Z}$ is $O(d^2|\mathbf{Z}'|\sigma_{\mathbf{Z}'})$. Let's discuss now the lemma that defines pivoting:

Lemma 2 (Pivot). *Given two triplets $t = (X, Y \mid \mathbf{Z})$ and $t' = (X', Y' \mid \mathbf{Z}')$ such that $\{X, Y\} \cup \mathbf{Z} = \{X', Y'\} \cup \mathbf{Z}'$, then, each cell (x', y') of slice $\mathbf{z}'$ in the tables of t' holds the same count as cell (x, y) of slice $\mathbf{z} = \mathbf{z}' \cup \{x', y'\} - \{x, y\}$ in the tables of t, i.e.,*

$$c_{x,y|\mathbf{z}'\cup\{\mathbf{x}',\mathbf{y}'\}-\{\mathbf{x},\mathbf{y}\}} = c_{x',y'|\mathbf{z}'}. \tag{4}$$

Before proving the lemma, let's note that the lemma states that the count in the tables of t' (cell (x', y') of slice $\mathbf{z}'$) not only can be obtained from the tables of t (cell (x, y) of slice $\mathbf{z}' \cup \{x, y\} - \{x', y'\}$), but each of them can be obtained with a computational cost of only four set operations. Using an efficient data structure for sets, such as a bitset with constant cost $O(1)$ for adding and removing members, the pivot between the tables of t and those of t' can be performed in $O(d^2|\mathbf{Z}'|\sigma_{|\mathbf{Z}'|})$. The total running of computing a statistical test when tables are computed from other tables, i.e., of sub-conditioning, pivoting, and remaining calculations of the test, is $O(d^2|\mathbf{Z}'|\sigma_{\mathbf{Z}'})$, an important reduction from $O(N(2 + |\mathbf{Z}|) + d^2\sigma_{\mathbf{Z}})$, the time complexity of a statistical test performed on data, when $\sigma_{\mathbf{Z}'} \ll N$. Moreover, since the upper bound of $\sigma_{\mathbf{Z}'}$ is N, the above is always smaller than the cost of computing the test from data.

Let's proceed with the proof now:

Proof. We start by noticing that the set of variables in the l.h.s. count of Eq. (4), viz. $\{X, Y\} \cup \mathbf{Z}' \cup \{X', Y'\} - \{X, Y\}$, is equal to $\{X', Y'\} \cup \mathbf{Z}'$. Therefore, by Eq. (1), the counts on each side must be equal, i.e.,

$$\begin{aligned} c_{x,y|\mathbf{z}' \cup \{x',y'\} - \{x,y\}} &= |\{\mathbf{d} \in D \mid x \wedge y \wedge \mathbf{z}' \cup \{x', y'\} \cup \{x, y\}\}| \\ &= |\{\mathbf{d} \in D \mid x' \wedge y' \wedge \mathbf{z}'\}| \\ &= c_{x',y'|\mathbf{z}'}. \end{aligned}$$

□

The pivot operation is exemplified in Fig. 1 between triplet $(X, Y \mid U, V)$ (d), and triplet $(X, U \mid Y, V)$ (e). Since the set of variables in each of them is the same, $\{X, Y, U, V\}$, they are a pivot of each other. We can see for $(X, U \mid Y, V)$, for instance, that the count for cell $X = 0, U = 0$ (top-left) of slice $\{Y = 0, V = 0\}$ (first) is 1. According to Eq. (4), it should match the count for $X = 0, Y = 0$ (top-left) of slice $\{Y = 0, V = 0\} \cup \{X = 0, U = 0\} - \{X = 0, Y = 0\} = \{U = 0, V = 0\}$ (first). As shown in the figure, it is in fact 1. Another example is $c_{X=1,U=0|\{Y=0,V=1\}} = 0 = c_{X=1,Y=0|\{U=0,V=1\}}$.

2.2 Cache of contingency tables

The basic idea of the cache is quite simple. Whenever a test for a triplet $t = (X, Y \mid \mathbf{Z})$ must be done, a super-triplet $t' = (X', Y' \mid \mathbf{Z}')$, $\{X, Y\} \cup \mathbf{Z} \subseteq \{X', Y'\} \cup \mathbf{Z}'$ is first looked-up in the cache. If it exists, its tables are reused by first pivoting the tables of t' into the tables of a triplet $(X, Y \mid \mathbf{Z}'')$, and then, since $\mathbf{Z} \subseteq \mathbf{Z}''$, the tables of t are computed by sub-conditioning.

As discussed above, the runtime complexity of the sub-conditioning and pivoting of two triplets grows with the cardinality of the largest conditioning set $\mathbf{Z}'$. Thus, when looking for a super-triplet in the cache, we want the smallest one. There is, however, no data-structure for sets with an efficient operation for finding the smallest superset of some given set. We propose a simple data structure that works well in practice. The cache is stored in a matrix of $n \times n$ cells (recall $n =\mid \mathbf{V} \mid$, the number of variables in the domain), with columns representing cardinalities and rows representing variables. If $m =\mid \{X\} \cup \{Y\} \cup \mathbf{Z} \mid$ is the amount of variables in a triplet, its tables are stored in m cells, each with column m, and the rows corresponding to its variables. Then, assuming an arbitrary but fixed ordering of the variables, and denoting by i_X the index of X in this ordering, the tables of $(X, Y \mid Z_1, ..., Z_{m-2})$ would be stored in all cells (i_X, m), (i_Y, m), (i_{Z_1}, m), ... , $(i_{Z_{m-2}}, m)$. To illustrate, consider a triplet $(X, Y \mid \{U, V\})$, and an ordering $[X, Y, Z, U, V, W]$. Then, $i_X = 1$, $i_Y = 2$, $i_U = 4$, $i_V = 5$, and then the tables would be stored in all cells $(1, 4)$, $(2, 4)$, $(4, 4)$, and $(5, 4)$. Note that more than one triplet could be stored in the same cell. For instance, $(X, U \mid W, Z)$ would be stored in cells $(1, 4)$, $(4, 4)$, $(6, 4)$ and $(3, 4)$. For that reason, in each cell, triplets and their tables are stored in a linked list. Because of the linked-list, the data-structure just described could turn very inefficient if many triplets were stored in the same position. In practice, however, we observed small linked-lists. These values are not reported in this work.

To finalize, let's illustrate how one can use the cache for retrieving the smallest superset of $\{X, Y, V\}$. Iteratively, explore the columns (i.e., cardinalities) starting from 3, the cardinality of the input set $\{X, Y, V\}$. By the definition of our data structure, a superset of $\{X, Y, V\}$, if it exists in the cache, would be stored in the lists of X, of Y, and of V. We then search the smallest one at each column.

3 Experiments

In this section we report the results of applying the proposed optimization strategies. For that, we compare the performance of different algorithms when they run Bayesian tests directly on data, denoted below BT (for Bayesian test), and when they run the Bayesian test but reusing tables from the cache, denoted $BT^\star$. For a fair comparison we also compare results of algorithms using a simple cache of tests, denoted $BT^\dagger$, that stores and retrieve tests using a hash table. As the following results show, our main contribution not only produces improvements over BT, but also over the more efficient $BT^\dagger$.

The comparison in performance between the algorithms is reported through two quantities: *runtime*, and *weighted number of tests* (**WNT**). The former is simply the total time taken by the algorithm to conclude. The latter reports the sum of weighted costs (see first term in Eq. (2)) of all tests executed (i.e., could not be reused from the cache). WNT accounts for the savings of reading from data, while runtime is also reporting the cost of deciding independences from the tables (i.e., both terms of Eq. (2)).

We considered two types of experiments. First we report runtime improvements of running a test over triplet t directly on data, versus running the same test but instead of data, it reuses tables stored in the cache, that is, when the cache holds a super-triplet of t. Second, we report runtime and WNT improvements of different runs of an independence-based algorithm called IBMAP-HC (explained in more detail below). In both experiments, statistical independence tests were performed over datasets sampled from known, randomly generated MNs using a Gibbs sampler. MNs with $n = 36$ random variables were generated by connecting each node randomly and uniformly with $\tau = 2, 4, 8$ other nodes in the network. One dataset was sampled for each value pair n and τ.

Figure 2 shows the results of our first experiment with several scatter plots. The purpose of these experiments is to compare the savings in runtime of $BT^\star$ (y-axis) vs BT (x-axis), i.e., the closer a point to the x-axis, larger is the improvement of $BT^\star$ vs BT (the different shaded regions help in this comparison). In this experiments we force that $BT^\star$ actually produces savings by adding, for each triplet $(X, Y \mid \mathbf{Z})$ tested, a super-triplet $(X, Y \mid \mathbf{Z}')$ with κ variables, i.e., $\kappa = 2 + |\mathbf{Z}'|$, and $\mathbf{Z} \subseteq \mathbf{Z}'$. Since runtime of reusing tables depends on $|\mathbf{Z}'|$, and the cost of running a test depends on N, we considered $\kappa = \{8, 32\}$ (first and second **rows**, respectively) and $N = \{100, 1000, 10000\}$ (first, second, and third **columns**, respectively). For each pair (κ, N) the experiment consisted in first producing randomly a triplet $(X, Y \mid \mathbf{Z}')$, with $|\mathbf{Z}'| = \kappa$, computing its tables,

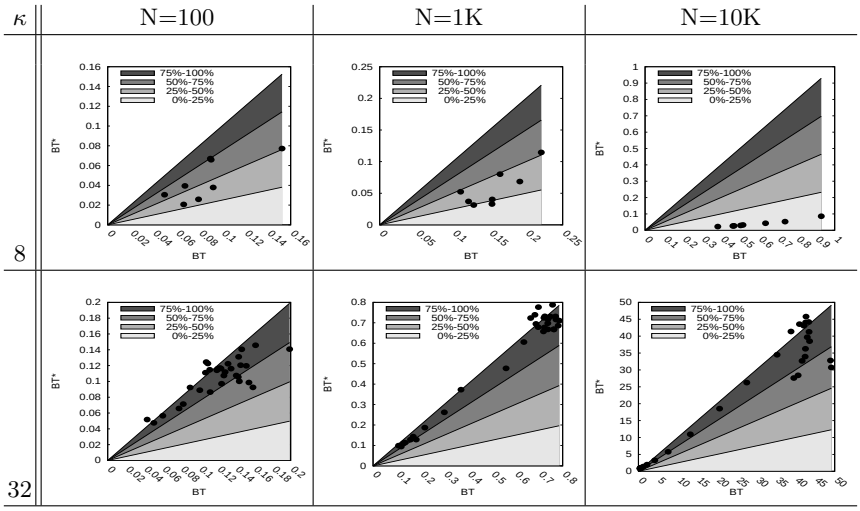

Fig. 2. Runtime comparisons of $BT^\star$ vs BT for tests run over datasets with $N = \{100, 1K, 10K\}$ datapoints, and caches with supertriplets with $\kappa = \{8, 32\}$

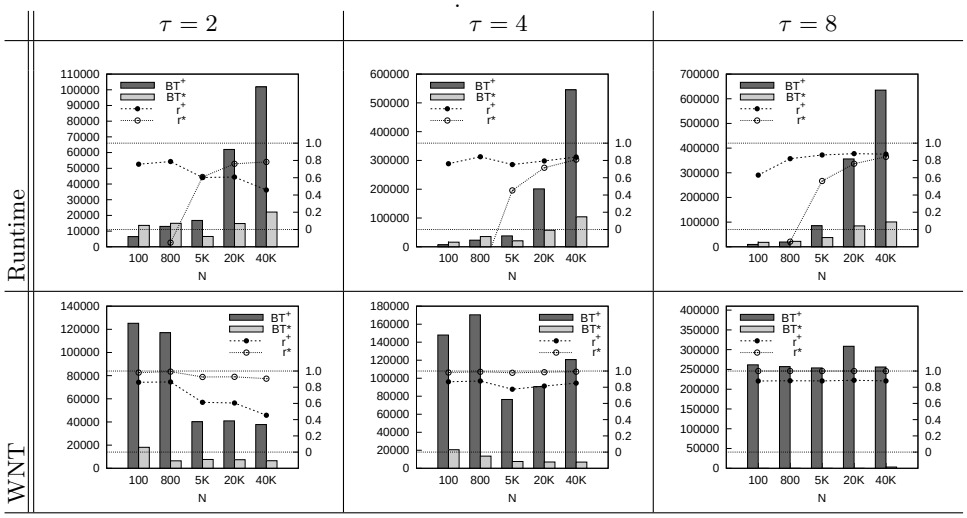

Fig. 3. Runtime and WNT of several hill-climbing searches for datasets with $n = 20$, $\tau = 2, 4, 8$ (columns), $N = 100, 800, 5K, 20K, 40K$, using the tests $BT^\dagger, BT^\star$.

and storing them in the cache. Then, for each $\kappa' < \kappa$, 100 triplets $(X, Y \mid \mathbf{Z})$ were generated randomly, with $|\mathbf{Z}| = \kappa'|$. BT and $BT^\star$ were conducted on each of those, reporting the mean value of the running time of each as a point in the plot. The results show improvements of $BT^\star$ w.r.t. BT in $\kappa = 8$ and not in $\kappa = 32$, increasing with the value of N (i.e., they get closer to the x-axis). The pour results of $\kappa = 32$ are expected, as they require large number of sums to generate the tables from tables with, in the worst case, $\max\{N, 2^{32}\}$ more slices.

The above experiments show improvements of $BT^\star$ vs BT when, for each test performed, there is a super-triplet stored in the cache. In practice, however, such super-triplet may not exist. As discussed in the introduction, we test this in our second set of experiments, i.e., on the problem of learning probabilistic models from data using the independence-based approach. In particular, we compared the running time of the independence-based algorithm called IBMAP-HC, presented in [13, 7]. The algorithm performs a hill-climbing search for maximizing a score of the structure, computed from the posteriors of several tests of independence using the Bayesian test (more details in the references). It starts computing the score for an initial structure and the score of all its neighbors, repeating this on the best neighbor until it arrives to a local maximum. To compute the score, it performs $2 \times (n-1)$ statistical test on each neighbor, with $\binom{n}{2} = n(n-1)/2$ neighbors. $BT^\dagger$ itself may result in important improvements, as many tests must be performed more than once. However, as the following results show, $BT^\star$ produces even further improvements over $BT^\dagger$. IBMAP-HC has some flexibility in the ordering that some tests are performed. In our experiments we took advantage of these flexibility and, whenever possible, a test with larger number of variables was conducted before tests with less variables.

Figure 3 shows results for these experiments. Each of the 3 plots in the top row show two super-imposed graphs, a bar graph and linepoints graph. The bar graph plots the runtimes of $BT^\dagger, BT^\star$ (in that order) over increasing number of datapoints, i.e., $N = \{100, 800, 5000, 20000, 40000\}$. The bars for BT were too large, and thus, were omitted for improving the readability of the most important comparison: $BT^\dagger$ vs $BT^\star$. The linepoints graph plots the runtime ratios $r^\star = 1 - \frac{BT^\star}{BT^\dagger}$ for assessing the saving obtained by $BT^\star$ over $BT^\dagger$, as well as $r^\dagger = 1 - \frac{BT^\dagger}{BT}$ for assessing the saving obtained by $BT^\dagger$ over BT, over the same values of N, resulting in improvements when the ratio is greater than 0. In other words, the ratio represents the savings. For instance, $r^\dagger = 0.8$ reads as 80% savings in runtime. The bottom row shows equivalent plots but comparing the WNT, and the WNT ratios over the same algorithms, in bar graphs and linepoint graphs, respectively. In both runtimes and WNT, the columns show the results for learning structures over datasets sampled from networks with increasing connectivities $\tau = \{2, 4, 8\}$. In all cases the datasets where sampled from networks with $n = 20$ variables.

The results show important improvements of our main contribution, the cache over tables $BT^\star$, over both other cases (second bar vs. first bar, $r^\dagger$, and $r^\star$). For large datasets the improvements in runtime of $BT^\star$ over $BT^\dagger$ are comparable to the improvements $BT^\dagger$ gained over BT (i.e., $r1 \approx r2$), reaching over 80% savings in runtime for large enough datasets (with exception of $\tau = 2$), demonstrating the practical effectiveness of our approach. A closer look shows that for small datasets, $N = 100, 800$, the $BT^\star$ test results in worst running times. This can be easily explained by the fact the gain in runtime from BT to $BT^\star$ is in the saving in reading the dataset. Therefore, for small enough datasets this gain may be lost, as demonstrated by these results. This is corroborated by the results of WNT, that show only the runtime involved in reading data. We can see that

savings of $BT^{\star}$ vs $BT^{\dagger}$ is over 95% for all Ns, reaching practically 100% for $\tau = 8$. The savings in running time for IBMAP-HC were up to 80% for datasets above 40,000 datapoints.

4 Conclusions and future work

The main contribution of our paper include the efficient sub-conditioning and pivoting operations, resulting in important improvements in the running time of the execution of a large number of statistical tests. Our evaluation on data from sampled networks shows that our methods perform well as the amount of data increases. Moreover, there remain several possible extensions to the current work for getting yet better results, stemming from the consideration of reusing computation in the rest of the steps of the computation of independence tests.

References

[1] A. Agresti. *Categorical Data Analysis*. Wiley, 2nd edition, 2002.

[2] C. F. Aliferis, I. Tsamardinos, and A. Statnikov. HITON, a novel Markov blanket algorithm for optimal variable selection. In *Proceedings of the American Medical Informatics Association (AMIA) Fall Symposium*, 2003.

[3] L. Amgoud and C. Cayrol. A reasoning model based on the production of acceptable arguments. *Annals of Maths. and Artif. Intel.*, 34:197–215, 2002.

[4] F. Bromberg and D. Margaritis. Improving the reliability of causal discovery from small data sets using argumentation. *JMLR*, 10:301–340, Feb 2009.

[5] F. Bromberg, D. Margaritis, and H. V. Efficient markov network structure discovery using independence tests. *JAIR*, 35:449–485, July 2009.

[6] F. Bromberg and F. Schluter. Variante de grow shrink para mejorar la calidad de markov blankets. In *CLEI, Pelotas, Brasil.*, September 2009.

[7] F. Bromberg and F. Schluter. Efficient independence-based map for robust markov networks structure discovery. *JMLR*, Submitted on May 2010.

[8] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley and Sons, 1999.

[9] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77:321–357, 1995.

[10] D. Margaritis. Distribution-free learning of Bayesian network structure in continuous domains. In *Proceedings of AAAI*, 2005.

[11] D. Margaritis and F. Bromberg. Efficient markov network discovery using particle filter. *Computational Intelligence*, 25(4):367–394, September 2009.

[12] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., 1988.

[13] F. Schluter and F. Bromberg. Enfoque perturbativo para mejorar la calidad de modelos probabilísticos gráficos. In *EnIDI, S. Rafael, Argentina.*, 2009.

[14] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. Adaptive Computation and Machine Learning Series. MIT Press, 2000.