





## jti and sparta: Time and Space Efficient Packages for Model-Based Prediction in Large Bayesian Networks

Mads Lindskou   
Aalborg University

Torben Tvedebrink   
Aalborg University  
University of Copenhagen

Poul Svante Eriksen  
Aalborg University

Søren Højsgaard   
Aalborg University

Niels Morling   
Aalborg University  
University of Copenhagen

---

### Abstract

A Bayesian network is a multivariate (potentially very high dimensional) probabilistic model formed by combining lower-dimensional components. In Bayesian networks, the computation of conditional probabilities is fundamental for model-based predictions. This is usually done based on message passing algorithms that utilize conditional independence structures. In this paper, we deal with a specific message passing algorithm that exploits a second structure called a junction tree and hence is known as the junction tree algorithm (JTA). In Bayesian networks for discrete variables with finite state spaces, there is a fundamental problem in high dimensions: A discrete distribution is represented by a table of values, and in high dimensions, such tables can become prohibitively large. In JTA, such tables must be multiplied which can lead to even larger tables. The **jti** package meets this challenge by using the package **sparta** by implementing methods that efficiently handle multiplication and marginalization of sparse tables through JTA. The two packages are written in the R programming language and are freely available from the Comprehensive R Archive Network.

*Keywords:* Bayesian networks, junction trees, sparse tables, R, C++.

---

## 1. Introduction

This paper is concerned with Bayesian networks (BNs) for discrete variables with finite state spaces (Pearl 1988; Lauritzen 1996; Maathuis, Drton, Lauritzen, and Wainwright 2018). For

such models, interest is typically in efficient computation of conditional probabilities of some variables given information about the state of other variables.

The components of such models are multivariate probability mass functions which are typically represented by a multi-dimensional array. For high-dimensional distributions where each variable may have a large state space, such an array can be prohibitive in terms of memory. For example, an 80-dimensional random vector in which each variable has 10 levels gives a state space with  $10^{80}$  configurations. Such a distribution can not be stored directly in a computer; in fact,  $10^{80}$  is one of the estimates of the number of atoms in the universe. However, if the array consists of only a few non-zero values, we need only store these values along with information about their location. That is a sparse representation of a table. The R (R Core Team 2024) package **sparta** (Lindskou 2024b), available from the Comprehensive R Archive Network (CRAN) at <https://CRAN.R-project.org/package=sparta>, implements efficient multiplication and marginalization of such sparse tables. The presentation of **sparta** is one of the goals of this paper.

The other goal is to present the **jti** package which implements message passing via the junction tree algorithm (JTA) for computing conditional probabilities in Bayesian networks. Specifically, **jti** implements JTA for discrete variables using the Lauritzen-Spiegelhalter updating scheme (Lauritzen and Spiegelhalter 1988). The **jti** uses **sparta** for the computations and the two packages together thereby allow for handling very large Bayesian networks in R. Package **jti** (Lindskou 2024a) is available from CRAN at <https://CRAN.R-project.org/package=jti>. We mention that calculating conditional probabilities in Bayesian networks using JTA is also referred to as “belief propagation”.

There are several other interesting suggestions in the literature for exploiting sparsity in tables. Some interesting approaches include probability trees (PTs, Boutilier, Friedman, Goldszmidt, and Koller 2013) and value-based potentials (VBPs Gómez-Olmedo, Cabañas, Cano, Moral, and Retamero 2021), which we discuss further in Section 5.3.

In addition to **jti**, there are to our knowledge, three other packages for belief propagation in R; **gRain** (Højsgaard 2012), **BayesNetBP** (Yu, Moharil, and Blair 2020) and **RHugin** (Konis 2017), where the latter is not on CRAN. The only R package on CRAN with an API for (dense) table operations is **gRbase** (Dethlefsen and Højsgaard 2005), on which **gRain** depends. Some R packages that rely on **gRain** and **gRbase** are **geneNetBP** (Moharil 2016) and **bnspatial** (Masante 2020). The **bnclassify** package (Mihaljević, Bielza, and Larrañaga 2018) has an internal lower C++ class implementation of dense conditional probability tables.

Although the paper is not concerned with learning the graph structure, we mention the packages **bnlearn** (Scutari 2010), **gRain**, **sparsebn** (Aragam, Gu, and Zhou 2019), **deal** (Boettcher and Dethlefsen 2003) and **bnstruct** (Franzin, Sambo, and di Camillo 2017), which can all be used to learn the DAG of a Bayesian network.

In Section 2, we introduce basic notation and terminology and in Section 3, we motivate the use of sparse tables through JTA. Section 4 serves as a primer to our novel representation of tables and their algebra given in Section 5. In Section 5, we also demonstrate the use of **sparta**. Section 6 outlines how to use **jti** and provides a specific example of using BNs, with a network which is well known from the literature. In Section 7, we show that the trade-off between execution time and memory allocation using **sparta** is acceptable for small and medium-sized tables and comparable to **gRbase** in high dimensional sparse tables. Finally, we

mention that **sparta** leverages the **RcppArmadillo** package (Eddelbuettel and Sanderson 2014) by implementing compute-intensive procedures in C++ for better run-time performance.

## 2. Notation and terminology

Let  $p$  be a discrete probability mass function of a random vector  $X = (X_v \mid v \in V)$  where  $V$  is a set of labels. The state space of  $X_v$  is denoted  $I_v$  and the state space of  $X$  is then given by  $I = \times_{v \in V} I_v$ . A realized value  $x = (x_v)_{v \in V}$  is called a cell. Given a subset  $A$  of  $V$ , the  $A$ -marginal cell of  $x$  is the vector,  $x_A = (x_v)_{v \in A}$ , with state space  $I_A = \times_{v \in A} I_v$ . A Bayesian network can be defined as a directed acyclic graph (DAG), see Figure 1, for which each node represents a random variable. A directed graph is a pair  $(V, E)$ , where  $V$  is a set of vertices, and  $E$  is a set of directed edges. The graph is acyclic if it does not contain any directed cycles, i.e., for no node is it possible to get back to that node if traversing the graph in the directions of the nodes. A directed edge from a node, say  $a$  to  $b$ , indicates that the random variable  $X_b$  depends on the random variable  $X_a$ . The DAG can either be estimated from data, specified through expert knowledge or a combination of these. The joint probability of a Bayesian network can be written as

$$p(x) = \prod_{v \in V} p(x_v \mid x_{pa(v)}),$$

where  $x_{pa(v)}$  denotes the parents of  $x_v$ ; i.e., the set of nodes with an arrow pointing towards  $x_v$  in the DAG. Also,  $x_v$  is a child of the variables  $x_{pa(v)}$ . Notice, that  $p(x_v \mid x_{pa(v)})$  has domain  $I_v \times I_{pa(v)}$ . Hence, we can encode the conditional probabilities in a table, say  $\phi(x_v, x_{pa(v)})$ , of dimension  $|I_v| \cdot |I_{pa(v)}|$ . Each entry in such tables is a parameter that must either be learned from data or specified by prior knowledge. It is also common in the literature to refer to these tables as potentials, and we shall use these terms interchangeably. In general, a potential does not have an interpretation. Sometimes, we also use subscript notation to explicitly show the set of variables on which a potential depends. That is,  $\phi_A$  is a potential defined over the variables  $X_A$ . The product  $\phi_A \otimes \phi_B$  of two generic tables over  $A$  and  $B$  is defined cell-wise as

$$(\phi_A \otimes \phi_B)(x_{A \cup B}) := \phi_A(x_A) \phi_B(x_B).$$

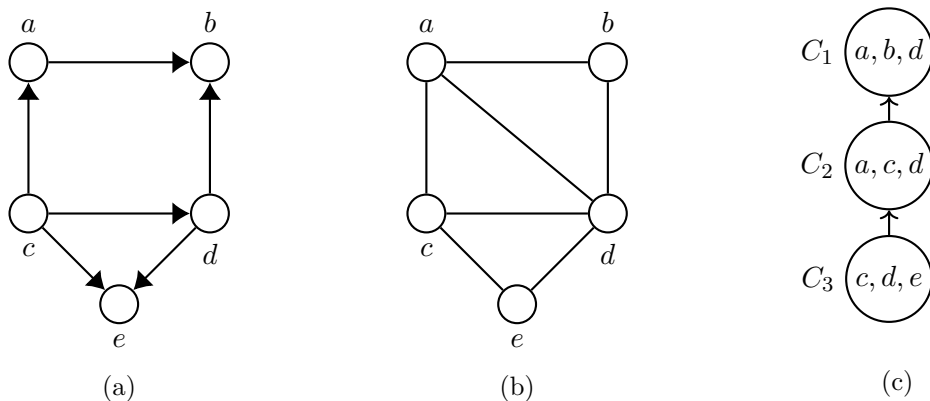


Figure 1: (a) A DAG. (b) A moralized and triangulated version of (a). (c) A rooted junction tree representation of (b) with root  $C_1$ .

In other words, the product is defined over the union of the variables of each of the two potentials. Division of two tables,  $\phi_A \oslash \phi_B$ , is defined analogously. The marginal table,  $\phi_A^{\downarrow B}$ , over the variables  $B \subseteq A$  is defined cell-wise as

$$\phi_A^{\downarrow B}(x_B) := \sum_{x_{A \setminus B} \in I_{A \setminus B}} \phi_A(x_B, x_{A \setminus B}).$$

Finally, for some  $B \subseteq A$ , fix  $x_B^*$ . The  $x_B^*$  slice of  $\phi_A(x)$  is then given by

$$\phi_A^{x_B^*}(x_{A \setminus B}) = \phi_A(x_{A \setminus B}, x_B^*).$$

### 3. Motivation through message passing in Bayesian networks

Consider the simple DAG given in Figure 1(a), from which the joint density can be read off:

$$p(x_a, x_b, x_c, x_d, x_e) = p(x_c)p(x_a | x_c)p(x_b | x_a, x_d)p(x_d | x_c)p(x_e | x_c, x_d). \quad (1)$$

If, for example, interest is in the joint distribution of  $(x_a, x_d)$  we have to sum over  $x_b, x_c$  and  $x_e$  and exploiting the factorization we could calculate this as

$$p(x_a, x_d) = \sum_{x_c} p(x_c)p(x_a | x_c)p(x_d | x_c) \sum_{x_e} p(x_e | x_c, x_d) \sum_{x_b} p(x_b | x_a, x_d).$$

The junction tree algorithm can be seen as an algorithm for automatically factorizing to circumvent the direct summation as described in what follows using a minimal example (a more general and technical exposition of the algorithm can be found in e.g., [Højsgaard, Edwards, and Lauritzen 2012](#)): First,

- moralize the DAG; i.e., connect nodes that share a common child node,
- remove directions in the DAG to obtain an undirected graph, and
- triangulate the resulting graph.

Moralization ensures that the corresponding parent and child nodes are put in the same maximal clique. A clique is a subset of the nodes for which the induced subgraph is complete (all vertices are neighbors), and it is maximal if it is not contained in any other clique. From here, by clique, we always mean a maximal clique, and we refer to these as the cliques of the graph.

An undirected graph is triangulated (or chordal) if it has no cycles of length greater than 3. If such cycles are present, the fill edges must be added to produce a triangulated graph. A triangulated graph is also called decomposable. Finding an optimal triangulation (in terms of minimizing the number of fill edges) is a NP hard problem, but good heuristic methods exists, see [Flores and Gámez \(2007\)](#). Finding a good triangulation can have a huge impact on the performance of JTA. A moralized and triangulated version of the graph in Figure 1(a) is shown Figure 1(b), where no fill edge was necessary to make the graph triangulated.

A triangulated graph can always be represented as a junction tree. A junction tree is a tree where the nodes are given by the cliques of the triangulated graph with the property that for

two cliques,  $C$  and  $C'$ , the intersection  $C \cap C'$  is contained in all clique nodes on the unique path between  $C$  and  $C'$ .

The cliques of the graph in Figure 1(b) are given as  $C_3 = \{c, d, e\}$ ,  $C_2 = \{a, c, d\}$  and  $C_1 = \{a, b, d\}$  where we arbitrarily designate  $C_1$  as the root to obtain the rooted junction tree in Figure 1(c). Now, assign each potential in Equation 1 to a clique potential for which the variables conform, e.g.,

$$\begin{aligned}\phi_{C_1}(x_a, x_b, x_d) &\leftarrow p(x_b \mid x_a, x_d), \\ \phi_{C_2}(x_a, x_c, x_d) &\leftarrow p(x_c)p(x_a \mid x_c), \\ \phi_{C_3}(x_c, x_d, x_e) &\leftarrow p(x_d \mid x_c)p(x_e \mid x_c, x_d).\end{aligned}$$

The clique potentials are now initialized (the network is also said to be initialized) and note that the clique potentials generally do not have any probability interpretation at this stage. We have obtained the clique potential representation

$$p(x_a, x_b, x_c, x_d, x_e) = \phi_{C_1}(x_a, x_b, x_d)\phi_{C_2}(x_a, x_c, x_d)\phi_{C_3}(x_c, x_d, x_e). \quad (2)$$

The network is said to be compiled at this stage, i.e., when moralization and triangulization have been performed, and the clique potential representation has been obtained. In complex networks with large clique potentials, it might not be feasible to even initialize the clique potentials due to lack of memory.

Next, the message passing scheme can now be applied to the junction tree. We describe the Lauritzen-Spiegelhalter (LS) scheme which works as follows. Locate a leaf node, here we choose  $C_3$ , and find the intersection,  $S_{32} = C_3 \cap C_2 = \{c, d\}$ , with its parent clique  $C_2$ . Then calculate the marginal potential  $\phi_{S_{32}}(x_c, x_d) = \sum_{x_e} \phi_{C_3}(x_c, x_d, x_e)$  and perform a so-called inward message by setting

$$\phi_{C_2}(x_a, x_c, x_d) \leftarrow \phi_{C_2}(x_a, x_c, x_d)\phi_{S_{32}}(x_c, x_d)$$

and update the leaf node as

$$\phi_{C_3}(x_c, x_d, x_e) \leftarrow \phi_{C_3}(x_c, x_d, x_e)/\phi_{S_{32}}(x_c, x_d),$$

where  $0/0 := 0$ . We say that  $C_2$  has collected its messages from all of its children. This procedure must be repeated until the root,  $C_1$ , has collected all its messages. Hence, we perform another inwards message by setting  $\phi_{S_{21}}(x_a, x_d) = \sum_{x_c} \phi_{C_2}(x_a, x_c, x_d)$  and update:

$$\begin{aligned}\phi_{C_1}(x_a, x_b, x_d) &\leftarrow \phi_{C_2}(x_a, x_c, x_d)\phi_{S_{21}}(x_a, x_d), \\ \phi_{C_2}(x_a, x_c, x_d) &\leftarrow \phi_{C_2}(x_a, x_c, x_d)/\phi_{S_{21}}(x_a, x_d).\end{aligned}$$

The inward phase terminates when the root clique potential has been normalized:

$$\phi_{C_1}(x_a, x_b, x_d) \leftarrow \phi_{C_1}(x_a, x_b, x_d) / \sum_{x_a, x_b, x_d} \phi_{C_1}(x_a, x_b, x_d).$$

To summarize, we have now obtained the set chain representation

$$\begin{aligned}p(x_a, x_b, x_c, x_d, x_e) &= \phi_{C_1}(x_a, x_b, x_d)\phi_{C_2}(x_a, x_c, x_d)\phi_{C_3}(x_c, x_d, x_e) \\ &= p(x_a, x_b, x_d)p(x_a \mid x_c, x_d)p(x_c, x_d \mid x_e),\end{aligned}$$

where the clique potentials are now conditional probability tables. Notice especially that  $\phi_{C_1}(x_a, x_b, x_d) = p(x_a, x_b, x_d)$ . In the so-called outward phase, we start by sending messages from the root by performing an outward message by letting  $\phi_{S_{12}}(x_a, x_d) = \sum_{x_b} \phi_{C_1}(x_a, x_b, x_d)$  and update:

$$\phi_{C_2}(x_a, x_c, x_d) \leftarrow \phi_{C_2}(x_a, x_c, x_d) \phi_{S_{12}}(x_a, x_d).$$

We say that  $C_1$  has distributed evidence to  $C_2$ . Notice, that  $\phi_{C_2}$  is now identical to the probability distribution defined over the variables  $x_a, x_c$ , and  $x_d$ . Finally, let  $\phi_{S_{23}}(x_c, x_d) = \sum_{x_a} \phi_{C_2}(x_a, x_c, x_d)$  and update  $\phi_{C_3}$ :

$$\phi_{C_3}(x_c, x_d, x_e) \leftarrow \phi_{C_3}(x_c, x_d, x_e) \phi_{S_{23}}(x_c, x_d).$$

As a consequence, we finally obtain the clique marginal representation

$$\begin{aligned} p(x_a, x_b, x_c, x_d, x_e) &= \frac{\phi_{C_1}(x_a, x_b, x_d) \phi_{C_2}(x_a, x_c, x_d) \phi_{C_3}(x_c, x_d, x_e)}{\phi_{S_{12}}(x_a, x_d) \phi_{S_{23}}(x_c, x_d)} \\ &= \frac{p(x_a, x_b, x_d) p(x_a, x_c, x_d) p(x_c, x_d, x_e)}{p(x_a, x_d) p(x_c, x_d)}, \end{aligned}$$

where all clique and separator potentials are identical to the marginal probability distribution over the variables involved. Hence, we can now find  $p(x_a, x_d)$  by locating a clique containing  $x_a$  and  $x_d$  and sum out all other variables. If we choose  $C_2$ , we get

$$p(x_a, x_d) = \sum_{x_c} \phi_{C_2}(x_a, x_c, x_d).$$

Each time we multiply, divide or marginalize potentials, a number of binary operations (addition, multiplication and division) are conducted under the machinery. For a network with 41 variables and a maximum size of the state space for each variable being 3, [Lepar and Shenoy \(1998\)](#) recorded a total number of 2 371 178 binary operations. We do not intend to follow the same analysis here. For sparse tables, however, the number of necessary binary operations is potentially much smaller.

### 3.1. Evidence and slicing

Suppose it is known, before message passing, that  $X_E = x_E^*$  for some labels  $E \subset V$ . We refer to  $x_E^*$  as evidence. Evidence can be entered into the clique potential representation in Equation 2 as follows. For each  $v \in E$ , choose an arbitrary clique,  $C$ , where  $v \in C$ , and set entries in  $\phi_C$  that are inconsistent with  $x_v^*$  equal to zero. The resulting clique potential is then said to be sliced. After message passing, all queries are then conditional on  $X_E = x_E^*$ . Thus, entering evidence leads to more zero-cells, and in a sparse setup, the resulting clique potentials will be even more sparse. After message passing, the clique potential  $\phi_C(x_C)$  is now equal to the conditional probability  $p(x_{C \setminus E} | x_E^*)$ .

It suffices to modify a single clique potential such that it is inconsistent with  $v \in E$ , for all  $v$  as described above. However, for sparse tables, it is advantageous to enter evidence in all clique potentials containing  $v$  since this leads to higher sparsity.

This is how evidence is handled in **jti**. In fact, this is one of the reasons why **jti** is able to handle very complex networks by exploiting the evidence using sparse tables from **sparta**.

Whenever a clique, say  $C$ , receives evidence on some variables, the size of the sparse potential corresponding to  $C$  will be reduced. This will be illustrated in Section 5.1.

It is also possible to enter evidence into the factorization in Equation 1. This is the key to handle complex networks that are otherwise infeasible due to lack of memory.

#### 4. An intuitive way of representing sparse tables

Before describing our method for multiplication and marginalization of sparse tables, it is illuminating to describe sparse tables in a standard R language setup. Consider two arrays  $f$  and  $g$ :

```
R> dn <- function(x) setNames(lapply(x, paste0, 1:2), toupper(x))
R> d <- c(2, 2, 2)
R> f <- array(c(5, 4, 0, 7, 0, 9, 0, 0), d, dn(c("x", "y", "z")))
R> g <- array(c(7, 6, 0, 6, 0, 0, 9, 0), d, dn(c("y", "z", "w")))
```

with flat layouts

```
R> ftable(f, row.vars = "X")           R> ftable(g, row.vars = "W")

  Y y1  y2
  Z z1 z2 z1 z2
X
x1  5  0  0  0
x2  4  9  7  0

  Y y1  y2
  Z z1 z2 z1 z2
W
w1  7  0  6  6
w2  0  9  0  0
```

Converting  $f$  and  $g$  to `data.frame` objects and exclude the cases with a value of zero:

```
R> df <- as.data.frame.table(f, stringsAsFactors=FALSE)
R> df <- df[df$Freq != 0,]
R> dg <- as.data.frame.table(g, stringsAsFactors=FALSE)
R> dg <- dg[dg$Freq != 0,]

R> print(df, row.names = FALSE)           R> print(dg, row.names = FALSE)

  X  Y  Z Freq           Y  Z  W Freq
x1 y1 z1     5         y1 z1 w1     7
x2 y1 z1     4         y2 z1 w1     6
x2 y2 z1     7         y2 z2 w1     6
x2 y1 z2     9         y1 z2 w2     9
```

This leaves us with two sparse tables,  $df$  and  $dg$ , respectively. To multiply  $df$  by  $dg$ , we must, by definition, determine the cases that match on the variables  $Y$  and  $Z$  that they have in common. For example, row 4 in  $df$  must be multiplied with row 4 in  $dg$  such that  $(y1, z2, x2, w2)$  is an element in the product with the value 81. And since the tables are sparse, no multiplication by zero will be performed. The multiplication can be performed with the following small piece of R code (which will be used in Section 7 in connection with the benchmarking exercise):

```
R> sparse_prod <- function(df, dg) {
+   S   <- setdiff(intersect(names(df), names(dg)), "Freq")
+   mrg <- merge(df, dg, by = S, suffixes = c("_df", "_dg"))
+   mrg <- within(mrg, val <- Freq_df * Freq_dg)
+   mrg[, setdiff(names(mrg), c("Freq_df", "Freq_dg"))]
+ }
```

The `merge` function performs, by default, what is also called an inner join or natural join in SQL terminology, which is exactly how we defined table multiplication in Section 2. Multiplying `df` and `dg` yields

```
R> sparse_prod(df, dg)

  Y Z X W val
1 y1 z1 x1 w1 35
2 y1 z1 x2 w1 28
3 y1 z2 x2 w2 81
4 y2 z1 x2 w1 42
```

Marginalization is even more straightforward. Marginalizing out  $X$  from `df` can for example be done using the built-in R function `aggregate` (which is also used in Section 7 for benchmarking):

```
R> aggregate(Freq ~ Y + Z, data = df, FUN = sum)

  Y Z Freq
1 y1 z1   9
2 y2 z1   7
3 y1 z2   9
```

Thus, we have the necessary tools to implement JTA using sparse tables. So why should we bother redefining sparse tables and algebras on these; because of execution time and memory storage. In Section 7, we show the effect of the effort of going beyond the `merge` and `aggregate` functions.

## 5. Sparse tables

Let  $T$  be a dense table with domain  $I = \times_{v \in V} I_v$ . Define the level set  $\mathcal{L} := \times_{v \in V} \mathcal{L}_v$  where  $\mathcal{L}_v = \{1, 2, \dots, |I_v|\}$  and let  $\# : I \rightarrow \mathcal{L}$  be a bijection. We define the sparse table  $\tau = (\Phi, \phi)$ , of  $T$  as the pair where  $\Phi$  is a matrix with columns given by the set of vectors in the sparse domain  $\mathcal{I} := \{\#(x) \mid T(x) \neq 0, x \in I\}$ , consisting of non-zero cells and where  $\phi$  is the corresponding vector of values. Thus, a column in  $\Phi$  represents a cell in  $\mathcal{I}$  and is written a tuple  $i = (i_1, i_2, \dots, i_{|V|}; i_v \in \mathcal{L}_v)$  which explicitly determines the ordering of the labels and hence the order of the rows in  $\Phi$ . The order of the columns in  $\Phi$  is not important as long as it agrees with  $\phi$ . We denote by  $\Phi[j]$  the  $j$ 'th column of  $\Phi$  and by  $\phi_j$  the corresponding  $j$ 'th value in  $\phi$ . The sub-matrix  $\Phi_S$  defined over the set of labels,  $S \subseteq V$ , is the resulting matrix when rows corresponding to labels in  $V \setminus S$  have been removed. Let  $T$  be the table  $\mathbf{f}$  from Section 4:



Y	y1	y2			
Z	z1	z2	z1	z2	
X					
x1	5	0	0	0	
x2	4	9	7	0	

The domain is given by

$$I = \{x_1, x_2\} \times \{y_1, y_2\} \times \{z_1, z_2\}$$

and  $\#$  can be chosen as the map  $(x_{\ell_1}, y_{\ell_2}, z_{\ell_3}) \mapsto (\ell_1, \ell_2, \ell_3)$  for  $\ell_1, \ell_2, \ell_3 \in \{1, 2\}$ . The non-zero cells can be identified from the table and we have  $\mathcal{I} = \{(1, 1, 1), (2, 1, 1), (2, 2, 1), (2, 1, 2)\}$ . Hence,

$$\Phi = \begin{bmatrix} 1 & 2 & 2 & 2 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix},$$

with values  $\phi = (5, 4, 7, 9)$ , corresponding to **df** in Section 4. Let  $G$  be another dense table with domain  $J = \times_{u \in U} J_u$  and sparse representation  $\gamma = (\Psi, \psi)$  with sparse domain  $\mathcal{J}$ . We then aim at defining the sparse multiplication  $\tau \otimes \gamma$  of  $T \otimes G$ . Let  $S = V \cap U$  be the separator labels shared between the two sparse tables  $\tau$  and  $\gamma$ . Next, define the map  $M_S(\Phi)$ , which transforms  $\Phi_S$  into a look-up table<sup>1</sup> as follows: the keys are the unique columns of  $\Phi_S$  and the value of  $M_S(\Phi)$  at key  $k$  is the set of column indices where column  $k$  can be found in  $\Phi_S$  and hence also in  $\Phi$  and is given by

$$M_S(\Phi)[k] = \{j \in \{1, 2, \dots, |\mathcal{I}|\} : \Phi[j] = k\}.$$

Let  $\mathcal{K}$  denote the mutual keys of  $M_S(\Phi)$  and  $M_S(\Psi)$ . The number of columns in the matrix of the resulting product  $\tau \otimes \gamma$  is then given as

$$N := \sum_{k \in \mathcal{K}} |M_S(\Phi)[k]| \cdot |M_S(\Psi)[k]|.$$

This observation is crucial, since the memory storage of the sparse product can then be computed in advance. If  $(\Pi, \pi)$  is the sparse product of  $\tau$  and  $\gamma$ , we can therefore initialize  $\Pi$  as a matrix with  $|V| + |U \setminus V|$  rows and  $N$  columns and  $\pi$  as an  $N$ -dimensional vector. Finally,  $\pi$  is given by the values  $\phi_j \cdot \psi_{j'}$  for  $j \in M_S(\Phi)[k]$  and  $j' \in M_S(\Psi)[k]$  for all  $k \in \mathcal{K}$ . The procedure is formalized in Algorithm 1. The number of binary operations is smaller than the equivalent dense table multiplication since every multiplication with zero is avoided. Moreover, since we only loop over the mutual keys,  $\mathcal{K}$ , the execution time will depend on the table having the least unique keys over the separator labels. Trivially, division of two sparse tables can be obtained by substituting line 11 of Algorithm 1 with  $\pi_l = \phi_j / \phi'_j$ .

Now, let  $G$  be the table **g** from Section 4 where the domain is given by

$$J = \{w_1, w_2\} \times \{y_1, y_2\} \times \{z_1, z_2\}.$$

Choose the map  $(w_{\ell_1}, y_{\ell_2}, z_{\ell_3}) \mapsto (\ell_1, \ell_2, \ell_3)$  for  $\ell_1, \ell_2, \ell_3 \in \{1, 2\}$ . In summary, we have the tables

$$\Phi = \begin{bmatrix} 1 & 2 & 2 & 2 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix}, \quad \Psi = \begin{bmatrix} 1 & 2 & 2 & 1 \\ 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 2 \end{bmatrix},$$

<sup>1</sup>A lookup table is a list arranged as key-value pairs. In R one can think of a look-up table as a named list in which the names are the keys and the values are the elements of the list.

**Algorithm 1** Multiplication of sparse tables.

---

```

1: procedure ( $\tau = (\Phi, \phi)$ : sparse table,  $\gamma = (\Psi, \psi)$ : sparse table)
2:    $S := V \cap U$ 
3:    $\mathcal{K}$ : Mutual keys of  $M_S(\Phi)$  and  $M_S(\Psi)$ 
4:    $N := \sum_{k \in \mathcal{K}} |M_S(\Phi)[k]| \cdot |M_S(\Psi)[k]|$ 
5:   Initialize the matrix  $\Pi$  with  $|V| + |U \setminus V|$  rows and  $N$  columns
6:   Initialize the vector  $\pi$  of dimension  $N$ 
7:    $l := 1$ 
8:   for  $k \in \mathcal{K}$  do
9:     for  $j \in M_S(\Phi)[k]$  and  $j' \in M_S(\Psi)[k]$  do
10:       $\Pi[l] := (\Phi[j], \Psi_{U \setminus V}[j'])$ 
11:       $\pi_l := \phi_j \cdot \psi_{j'}$ 
12:       $l = l + 1$ 
13:     end for
14:   end for
15:   return  $(\Pi, \pi)$ 
16: end procedure

```

---

and  $\phi = (5, 4, 7, 9)$  and  $\psi = (7, 6, 6, 9)$ . The separator labels are given by  $S = \{y, z\}$ , and the lookup tables of  $\Phi$  and  $\Psi$  are given by

$$\begin{aligned}
M_S(\Phi) &= \{(1, 1) := \{1, 2\}, & (2, 1) := \{3\}, & (1, 2) := \{4\}\} \\
M_S(\Psi) &= \{(1, 1) := \{1\}, & (2, 1) := \{2\}, & (1, 2) := \{4\}, & (2, 2) := \{3\}\}.
\end{aligned}$$

Above,  $y$  corresponds to row two, and  $z$  corresponds to row three in  $\Phi$ . So, for example,  $M_S(\Phi)[(1, 1)] = \{1, 2\}$  means that the key  $(1, 1)$  has the value  $\{1, 2\}$ , which in turn means that  $(1, 1)$  is found in columns 1 and 2 in  $\Phi$ . Therefore, all values  $\phi_j$  for  $j \in M_S(\Phi)[(1, 1)]$  must be multiplied with all values  $\psi_j$  for  $j \in M_S(\Psi)[(1, 1)]$ , etc. Hence,

$$\Pi = \begin{bmatrix} 1 & 2 & 2 & 2 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 1 & 1 & 2 \end{bmatrix},$$

and

$$\pi = (\phi_1 \cdot \psi_1, \phi_2 \cdot \psi_1, \phi_3 \cdot \psi_2, \phi_4 \cdot \psi_4) = (35, 28, 42, 81),$$

as expected from the result in Section 4 using `sparse_prod`. Notice, that we save any computation with  $\psi_3$  since  $(2, 2)$  is not a key in  $M_S(\Phi)$ .

We mention that, addition and subtraction of sparse tables are more demanding since we have to reconstruct zero-cells if one of the tables has a non-zero cell -value while the other table has a zero-cell in the corresponding separator cell. Fortunately, these operations are not needed in JTA.

The marginal sparse table  $\tau^{\downarrow A} = (\Phi^{\downarrow A}, \phi^{\downarrow A})$  of  $\tau$ , corresponding to  $T^{\downarrow A}$ , can be calculated using the map  $M_A(\Phi)$  and, for each key  $k \in M_A(\Phi)$ , sum the corresponding values in  $\phi$ . However, for massive tables, the memory footprint of  $M_A(\Phi)$  is unnecessarily large. Instead, we construct the lookup-table  $H_A(\Phi)$  where the keys are the unique columns of  $\Phi_A$ , as was

**Algorithm 2** Marginalization of sparse tables.

---

```

1: procedure ( $\tau = (\Phi, \phi)$ : sparse table,  $A$ : Set of labels)
2:   Construct  $H_A(\Phi)$ 
3:    $N = |H_A(\Phi)|$ 
4:   Initialize the matrix  $\Phi^{\downarrow A}$  with  $|A|$  rows and  $N$  columns
5:   Initialize the vector  $\phi^{\downarrow A}$  of dimension  $N$ 
6:   Let  $\mathcal{K}$  be the keys of  $H_A(\Phi)$ 
7:    $l := 1$ 
8:   for  $k \in \mathcal{K}$  do
9:      $(j, v) := H_A(\Phi)[k]$ 
10:     $\Phi^{\downarrow A}[l] := \Phi_A[j]$  ▷ deduced by picking elements from  $\Phi[j]$ 
11:     $\phi_l^{\downarrow A} := v$ 
12:     $l = l + 1$ 
13:   end for
14:   return  $(\Phi^{\downarrow A}, \phi^{\downarrow A})$ 
15: end procedure

```

---

the case in  $M_A(\Phi)$ . However, the values are themselves pairs where the first element is an index to any of the column indices where the corresponding key can be found in  $\Phi_A$ . The second element is the final cell value in the marginalized table corresponding to the key. The pair corresponding to the key  $k$  is therefore on the form

$$H_A(\Phi)[k] = (j, v), \quad v = \sum_{\ell: \Phi_A[\ell]=k} \phi_\ell \text{ and } \Phi_A[j] = k.$$

The value  $v$  can easily be computed iteratively. The point here is, that we never have to store  $\Phi_A$  since we can deduce all information from  $\Phi$  on the fly given the row indices corresponding to  $A$  in  $\Phi$ . The number of columns in the final matrix  $\Phi^{\downarrow A}$ , and hence the number of elements in  $\phi^{\downarrow A}$ , is given by  $|H_A(\Phi)|$ . The procedure is formalized in Algorithm 2. Consider again the sparse table  $\tau$  of  $T$  and let  $A = \{y, z\}$ . Then, the resulting sparse marginal table has two rows corresponding to  $y$  and  $z$ . The construction of  $H_A(\Phi)$  is as follows. The first column in  $\Phi$  is extracted, and the entry corresponding to  $x$  is deleted. Call the resulting vector (key)  $k_1$ . Now, set  $H_A(\Phi)[k_1] = (j = 1, v = 5)$  since  $\phi_1 = 5$ . Extract now, the second column of  $\Phi$  and let  $k_2$  be the resulting key when removing the entry corresponding to  $x$ . Since  $k_1 = k_2$  and  $\phi_2 = 4$  we update  $H_A(\Phi)[k_1] = (j = 2, v = 9)$ . Proceeding this way gives

$$H_A(\Phi) = \{(1, 1) := (j = 2, v = 9), (2, 1) := (j = 3, v = 7), (1, 2) := (j = 4, v = 9)\}.$$

Thus

$$\Phi^{\downarrow A} = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix},$$

and  $\psi^{\downarrow A} = (9, 7, 9)$ . For  $B \subset V$ , the  $i_B^*$ -slice of a sparse table  $\tau = (\Phi, \phi)$  is obtained by removing columns,  $k$ , in  $\Phi$  for which  $k$  does not agree with  $i_B^*$ . We leave out the formal procedure for slicing.

Finally, we mention that Algorithm 1 is generic and applies in every situation. However, if the domain of one of the tables is a subset of the domain in the other table, multiplication can

be performed much faster since we do not have to create new cells. This is always the case in JTA since the domain of the message is a subset of the domain in the potential receiving this message. We leave out the formal algorithm and just mention, that this algorithm also exploits the lookup table  $M$  whereafter it locates the cells to keep in the larger table without constructing new cells.

### 5.1. How to use sparta

The **sparta** package can be installed from within an R session by typing `install.packages("sparta")`. To demonstrate the use of **sparta**, we revisit the example from Section 4 of the two (dense) tables  $f$  and  $g$  with mutual variables,  $Y$  and  $Z$ :

```
R> ftable(f, row.vars = "X")           R> ftable(g, row.vars = "W")

  Y y1  y2
  Z z1 z2 z1 z2
X
x1   5  0  0  0
x2   4  9  7  0

  Y y1  y2
  Z z1 z2 z1 z2
W
w1   7  0  6  6
w2   0  9  0  0
```

We can convert these to their equivalent **sparta** versions as

```
R> library("sparta")
R> sf <- as_sparta(f)
R> sg <- as_sparta(g)
```

Printing the object by the default printing method yields

```
R> print.default(sf)

  [,1] [,2] [,3] [,4]
X     1     2     2     2
Y     1     1     2     1
Z     1     1     1     2
attr("vals")
[1] 5 4 7 9
attr("dim_names")
attr("dim_names")$X
[1] "x1" "x2"

attr("dim_names")$Y
[1] "y1" "y2"

attr("dim_names")$Z
[1] "z1" "z2"

attr("class")
[1] "sparta" "matrix"
```

The columns are the cells in the sparse matrix, and the `vals` attribute are the corresponding values which can be extracted with the `vals` function. Furthermore, the domain resides in the `dim_names` attribute, which can also be extracted using the `dim_names` function. From the output, we see that `(x2, y2, z1)` has a value of 7. Using the `sparta` print method prettifies things:

```
R> print(sf)
```

```
   X Y Z val
1 1 1 1   5
2 2 1 1   4
3 2 2 1   7
4 2 1 2   9
```

where row  $i$  corresponds to column  $i$  in the sparse matrix. We settled for this print method because printing column wise leads to unwanted formatting when the values are decimal numbers. Consider the cell `(2,1,1)`. The corresponding named cell is then

```
R> get_cell_name(sf, sf[, 2L])
```

```
   X   Y   Z
"x2" "y1" "z1"
```

where `sf[, 2L]` is the second column (row in the output) of `sf`, which is `(2, 1, 1)`. The product of `sf` and `sg` is

```
R> mfg <- mult(sf, sg)
```

```
R> mfg
```

```
   X Y Z W val
1 2 1 2 2  81
2 2 2 1 1  42
3 1 1 1 1  35
4 2 1 1 1  28
```

The equivalent dense table has  $2^4 = 16$  entries. However, `mfg` stores 20 values after all, 16 of which are information about the cells. That is, there is some overhead storing the information about the cells, see Section 5.2. Converting `sf` into a conditional probability table (CPT) with conditioning variable Z:

```
R> sf_cpt <- as_cpt(sf, y = "Z")
```

```
R> sf_cpt
```

```
   X Y Z   val
1 1 1 1 0.312
2 2 1 1 0.250
3 2 2 1 0.438
4 2 1 2 1.000
```

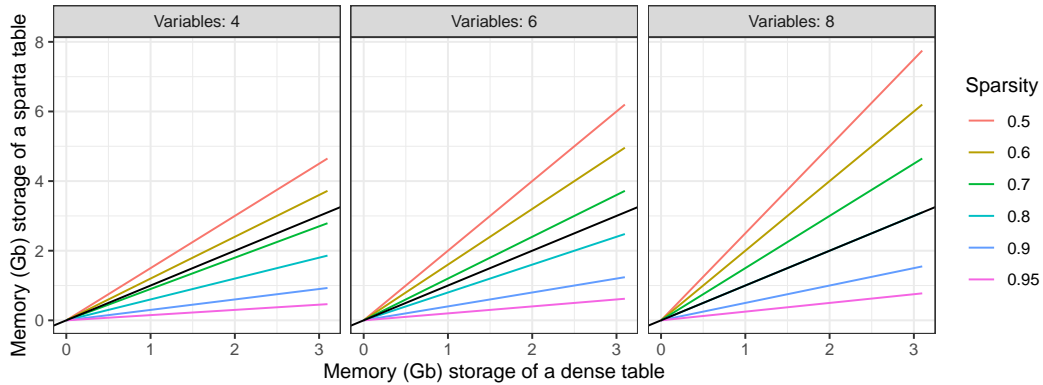


Figure 2: The black identity line indicates the number of gigabytes needed to store the dense table with  $x$  elements. The colored lines indicate the number of gigabytes required to store the equivalent **sparta** object with the respective number of variables and sparsity.

Slicing `sf` on  $X = x1$

```
R> slice(sf, s = c(X = "x1"), drop = TRUE)
```

```
  Y Z val
1 1 1   5
```

reduces `sf` to a single non-zero element, whereas the equivalent dense case would result in a (Y,Z) table with one non-zero element and three zero-elements. This `slice` function is used in **jti** when the evidence  $X = x1$  is entered into the clique potential corresponding to `sf`. Marginalizing out Y in `sg` yields

```
R> marg(sg, y = "Y")
```

```
  Z W val
1 2 2   9
2 2 1   6
3 1 1  13
```

This is in correspondence with the example in Section 5. Finally, we mention that a sparse table can be created using the constructor `sparta_struct`, which is necessary if the corresponding dense table is too large to have in memory.

## 5.2. When to use **sparta**

As shown in Section 5.1, there is an overhead of storing the information in a **sparta** object. A dense array with  $x$  elements takes up  $8x$  bytes plus some negligible memory of storing the variable names etc. On the contrary, a **sparta** object with  $y < x$  elements takes up  $y(4k + 8)$  bytes, where  $k$  is the number of variables (these can be stored as integers and hence only requires 4 bytes each). In Figure 2, we have plotted this relation for  $k = 4, 6$  and  $8$ , and different levels of sparsity. That is, a sparsity of  $1/2$  implies that  $y = x/2$ . The black identity

line indicates the number of gigabytes needed to store the dense table with  $x$  elements. The size of the state spaces of the variables is implicitly reflected by the memory is needed to store the dense table. The more memory needed, the larger state space of the variables. However, more variables and a larger state space of the variables will intuitively result in a more sparse table, making **sparta** efficient even for several variables.

The take away message from Figure 2 is that when the state space of the variables and the sparsity increases the benefit of storing the tables using **sparta** will outweigh the overhead of storing the additional information.

In connection to JTA, **sparta** is favorable when cliques with many variables imply a high degree of sparsity. In particular, this is often the case for tables in a Bayesian network representing a genetic pedigree. In this case, cliques tend to be small, but the state space of the variables can be arbitrarily large due to the large amount of DNA information for each member of the pedigree.

### 5.3. Probability trees and value based potentials

This section is devoted to discussing differences between **sparta** and probability trees (PTs, [Boutilier et al. 2013](#)) and value based potentials (VBPs, [Gómez-Olmedo et al. 2021](#)). Firstly, PTs are potentials that are represented as trees where context specific information (CSI) (conditional independencies that only hold in specific contexts) can be exploited by collapsing nodes in the tree. This, however calls for methods to learn these CSIs, which can be computationally demanding. Moreover, PTs can be pruned, further resulting in an approximation of the potential by collapsing leaf nodes (in the same branch though) for which all values are close to the mean value of that branch. Finally, PTs can not disregard zero-cells in any way, which, by construction, is the power of **sparta**. We are not aware of any open source code that implements PTs.

Very recently, and almost parallel to this paper, VBPs were introduced to overcome the limitations of PTs ([Gómez-Olmedo et al. 2021](#)). VBPs is an acronym for four new potentials that can be either value driven or index driven. We shall only focus on the new potentials called index driven with map (IDM) since these have the most resemblance with **sparta**, and also because they seem to perform best overall according to the benchmarks provided in [Gómez-Olmedo et al. \(2021\)](#).

To introduce IDMs, we first consider a dense table, `dt`, over the variables  $Z$ ,  $Y$ , and  $X$  where all cells are represented by a vector of integers and assign to each cell a unique index:

```
R> dt <- cbind(
+   expand.grid(Z = 1:2, Y = 1:2, X = 1:2),
+   Freq = c(.4, .1, .7, .1, .6, 0, 0, .9),
+   idx = 1:2^3
+ )
R> print(dt, row.names = FALSE)
```

```
Z Y X Freq idx
1 1 1 0.4 1
2 1 1 0.1 2
1 2 1 0.7 3
```

```

2 2 1 0.1 4
1 1 2 0.6 5
2 1 2 0.0 6
1 2 2 0.0 7
2 2 2 0.9 8

```

Since indices are unique, the table can be represented by the vector

```
R> structure(dt$Freq, names = dt$idx)
```

```

 1  2  3  4  5  6  7  8
0.4 0.1 0.7 0.1 0.6 0.0 0.0 0.9

```

where the names are the indices. The correspondence between indices and cells is as follows (Gómez-Olmedo *et al.* 2021): First assign to variable  $\ell$  the weight  $w_\ell = w_{\ell+1} \cdot |I_{\ell+1}|$  for  $\ell < |V|$  and  $w_{|V|} = 1$  where  $|V|$  is the number of variables. The ordering is given from first to last, i.e.,  $Z$  is the first,  $Y$  is the second, and  $X$  is the third variable in the case of `dt`.

The index of a given cell,  $i$ , is then given by

$$\text{idx}(i) = \sum_{\ell=1}^{|V|} (i_\ell - 1) \cdot w_\ell. \quad (3)$$

Thus, the index of cell  $(z_1, y_2, x_2)$  is  $(1 - 1) \cdot 4 + (2 - 1) \cdot 2 + (2 - 1) \cdot 1 = 4$  as expected. Given an index,  $k$ , the  $k$ 'th component of the corresponding cell is given by

$$\lfloor k/w_\ell \rfloor \text{ modulo } |I_\ell|, \quad (4)$$

where  $I_\ell$  is the state space of the  $\ell$ 'th variable. Thus, it is a fairly cheap operation to convert between the index and the cell, which is the backbone when manipulating IDMs. The IDM,  $\text{IDM}_\phi$ , of the potential  $\phi$  is a representation of  $\phi$  consisting of a lookup table  $D$  and an array  $A$ .  $A$  holds all unique values of  $\phi$ , excluding zero. The keys in  $D$  are the indices in  $\phi$  excluding indices corresponding to zero-cells and the values are indices from  $A$  corresponding to the cell value. We can now form the IDM of `dt`:

```

R> dt_no_zeroes <- subset(dt, Freq != 0)
R> unique_values <- unique(dt_no_zeroes$Freq)
R> A <- structure(unique_values, names = seq_along(unique_values))
R> D <- structure(apply(dt_no_zeroes$Freq, function(k) {
+   match(k, A)
+ }), names = dt_no_zeroes$idx)
R> (IDM <- list(A = A, D = D))

```

`$A`

```

 1  2  3  4  5
0.4 0.1 0.7 0.6 0.9

```

`$D`

```

1 2 3 4 5 8
1 2 3 2 4 5

```



First, notice that **A** and **D** are not true lookup tables with constant lookup, but ordinary R vectors. At first, it may seem redundant to form the **A** instead of just forming the structure

```
R> (B <- structure(dt_no_zeroes$Freq, names = dt_no_zeroes$idx))

  1  2  3  4  5  8
0.4 0.1 0.7 0.1 0.6 0.9
```

This is because for IDMs, repeated values, like 0.1, take up a single float in the memory as opposed to **B** that must store a float for each repetition of the same value. For tables with many repeated values, this can potentially save a lot of memory. The question is now whether or not IDMs can be multiplied and marginalized within reasonable time. There are no benchmarks of multiplication and marginalization in Gómez-Olmedo *et al.* (2021). Still, we can with reasonable confidence state that multiplication of IDMs becomes increasingly slower than with **sparta** as the state space of the product increases. We state this for two reasons. Firstly, we did actually consider a variant of IDMs for which multiplication turned out to be extremely slow for large tables. Secondly, when IDMs are multiplied one must loop over the entire dense state space of the resulting table. In more detail, suppose we want to find the product  $IDM_{\phi_Z} = IDM_{\phi_X} \otimes IDM_{\phi_Y}$ . Let  $z_\ell$  be the  $\ell$ 'th cell in  $IDM_{\phi_Z}$ . Then for all  $\ell = 1, 2, \dots, |I_Z|$ , one must use Equation 4 to construct the cell  $z_\ell$ , project this cell onto **A**, use Equation 3 to convert to and index in  $IDM_{\phi_A}$  and lookup the value and test if this is zero. If not, also project  $z_\ell$  onto **B**, multiply the values and append the scalar product and  $\ell$  to  $IDM_{\phi_C}$ . Imagine  $|I_C|$  being huge, then multiplication of IDMs is cumbersome.

Based on this, we do not intend to benchmark IDMs against **sparta**, but we summarize the discussion by saying that neither IDMs nor **sparta** is, generally, better than the other. It depends on the problem at hand. IDMs are most often more memory efficient than **sparta** but for large tables with high degrees of sparsity, **sparta** is faster. For very large networks, it may even make sense to encode a Bayesian network to hold both IDMs and **sparta** tables such that the IDM potentials ensure that it is even possible to compile the network. This calls for methods to combine IDMs and **sparta** tables which we leave for future research.

## 6. A usecase of **jti**

The **jti** package can be installed from within an R session by `install.packages("jti")`. In **jti**, there are two ways of specifying a Bayesian network, either by a list of CPTs or a dataset together with a DAG. In the latter case, the CPTs are found using maximum likelihood estimates. Here, we describe how to use **jti** with the classic Bayesian network **asia** (Lauritzen and Spiegelhalter 1988) where the corresponding CPTs are part of **jti**. The network represents a simplified model to help diagnose patients arriving at a respiratory clinic. A history of smoking has a direct influence on whether or not a patient has bronchitis and whether or not a patient has lung cancer. Both lung cancer and bronchitis can result in dyspnea. An x-ray result depends on the presence of either tuberculosis or lung cancer. Finally, a visit to Asia influences the probability of having tuberculosis. The DAG is depicted in Figure 3.

We use the version of **asia** called **asia2**, both shipped with **jti**, which is a list of CPTs shipped with **jti**. The first step is to call `cpt_list` for some initial checks and conversion to **sparta** tables:

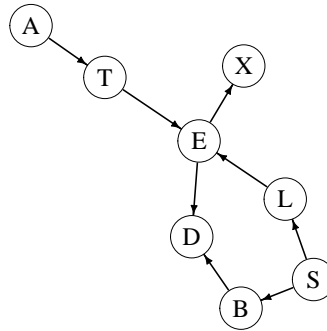


Figure 3: The DAG for the `asia` network with variables `asia` (A), `tuberculosis` (T), `either` (E), `x-ray` (X), `lung` (L), `dysp` (D), `smoke` (S) and `bronc` (B).

```
R> cl <- cpt_list(asia2)
R> cl
```

List of CPTs

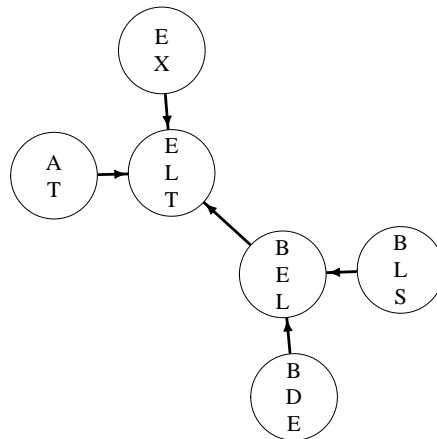
```
-----
P( asia )
P( tub | asia )
P( smoke )
P( lung | smoke )
P( bronc | smoke )
P( either | lung, tub )
P( xray | either )
P( dysp | bronc, either )
```

```
<bn_, cpt_list, list>
-----
```

From the output, we see the inferred CPTs correspond to Figure 3, giving rise to a factorization in the same way as in Equation 1. The network is now ready for compilation which involves moralization and triangulation. In `jti` there are four choices for triangulation which are all based on the elimination game algorithm, see Flores and Gámez (2007). One of the most well-known heuristics is `min_fill` which tries to minimize the number of fill edges. Evidence can be entered either at compile stage or just before message passing begins. It is always advisable to enter evidence at compile stage since we know from Section 3.1 that this reduces the number of non-zero elements in the CPTs and hence the memory footprint and execution time. A good strategy might be to locate one or more of the largest cliques and enter evidence on the nodes contained in these. We can investigate the cliques and their state spaces prior to compilation by triangulating the graph as follows:

```
R> tri <- triangulate(cl, tri = "min_fill")
```

The `tri` object is a list containing the triangulated graph, `new_graph` as a matrix, a list of `fill_edges`, the `cliques`, and the size of the dense `statespace` of each clique. Now, let

Figure 4: A junction tree for the `asia` network.

for example `tub = yes` be the evidence indicating that a given person has tuberculosis. The compiled network is then constructed as

```
R> cp <- compile(cl, evidence = c(tub = "yes"), tri = "min_fill")
R> cp
```

```
Compiled network (cpts initialized)
```

```
-----
Nodes: 8
Cliques: 6
- max: 3
- min: 2
- avg: 2.67
Evidence:
- tub: yes
<bn_, charge, list>
-----
```

The cliques can be extracted from the compiled object with `get_cliques(cp)`. The compiled object can now be entered into the message passing procedure as:

```
R> j <- jt(cp)
```

The junction tree can be visualized by plotting the object as `plot(j)`, see Figure 4. Finally, we can calculate the probability of a given person having a positive x-ray result, `xray = yes`, given that the person has tuberculosis as

```
R> query_belief(j, nodes = "xray")
```

```
$xray
xray
  yes  no
0.98 0.02
```

Thus, given that a person has tuberculosis, the probability of observing a positive x-ray result is 0.98. The probability of observing a positive x-ray result given that `tub = "no"` can be calculated accordingly and equals 0.1012. Joint queries can be calculated by specifying `type = "joint"` in `query_belief`.

## 7. Time and memory trade-off in sparta

We investigate the trade-off between memory allocation and execution time for multiplication and marginalization on **sparta** objects. If **sparta** objects do not perform reasonably well for small and medium sized tables, their practical usage is limited in real usecases.

Thus, we compare three functions for multiplication and three functions for marginalization: (1) The `mult` and `margin` functions from **sparta**, (2) the `tabMult` and `tabMarg` functions from **gRbase** and (3) the `sparse_prod` and `aggregate` functions from Section 4. In the latter case, we refer to **base R** as the package.

We randomly generate pairs of tables such that the number of cells in the product of the two tables does not exceed  $10^6$ . We varied the sparsity of the product of the tables; 0% sparsity (dense tables), (1, 75]% sparsity, and (75, 99]% sparsity. For each pair of tables, we multiply them together and record the memory usage (in megabytes) of the product and the execution time (in seconds). As **gRbase** is the standard and most mature package for Bayesian networks in R, the performance comparisons are relative to that of **gRbase**. Hence, in Figure 5, **gRbase** performs better for tables with relative scores above one (indicated by horizontal dashed lines), whereas values below one show cases where the alternative approaches are better. The comparisons are plotted for different ranges of table sizes (panels) and sparsity of the resulting table (first axis).

**Multiplication (size):** The first row of Figure 5 describes the size of the table resulting from multiplying two tables. The **base R** package consistently produces tables of smaller sizes than **sparta** for very small tables with 100 ( $10^2$ ) cells. For tables with more than 100 cells, **sparta** consistently produces smaller tables except for a single case. Increasing the degree of sparsity leads to reduced object sizes for both **base R** and **sparta**, and for tables with more than 75% sparsity, **sparta** outperforms both **base R** and **gRbase** except in the first two panels with small tables.

**Multiplication (time):** The second row of Figure 5 describes the computing time for multiplying two tables. Clearly, **sparta** outperforms **base R** by orders of magnitude. For larger tables (the two rightmost columns), there is also a clear effect of the degree of sparsity on the computing time.

**Marginalization (time):** The third row of Figure 5 describes the computing time for marginalizing out all variables in a table. When the degree of sparsity increases, the computing time decreases. In the comparison between **gRbase** and **sparta**, we see that the marginalization implementation of **sparta** is competitive with **gRbase**, especially for tables with 10 000 ( $10^4$ ) or fewer cells. Efficient marginalization is not only relevant for propagation, but also for querying probabilities in a junction tree that has been fully propagated.

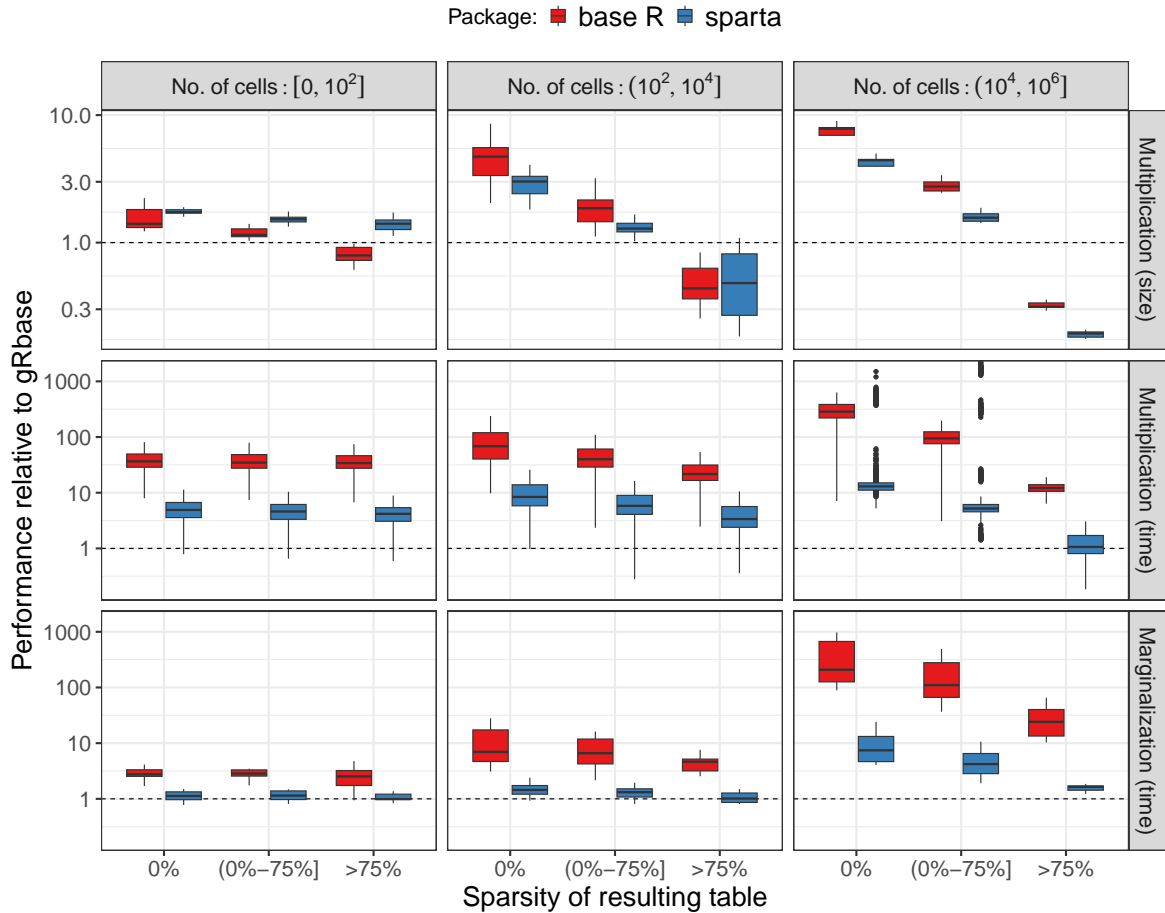


Figure 5: Comparison of **base R** (`sparse_prod` and `aggregate`) and **sparta** (`mult` and `marg`) to **gRbase** (`tabMult` and `tabMarg`) in terms of memory usage and timing. The top row shows the relative memory usage for multiplication, whereas the two lower rows show the relative timing for multiplication and marginalization, respectively.

## 8. Summary

We have presented a novel method for the multiplication and marginalization of sparse tables. The method is implemented in the R package **sparta**. However, the method is generic, and we have provided detailed pseudo algorithms facilitating the extension to other languages. In addition, we have presented the companion package **jti** to illustrate some of the advantages of **sparta** in connection with the junction tree algorithm. We hope to explore the benefit of the C API for working with external pointers to reduce memory usage for **sparta** objects in the future. We also described IDM potentials which are very memory efficient but less time efficient. We intend to explore how IDMs and **sparta** can be combined in the future.

The memory footprint of the clique potentials can become prohibitively large when the sizes of the cliques are large. This may not be true in general for sparse tables. As a matter of fact, it may be optimal to have large cliques if they are very sparse and/or if it is common to observe evidence variables in such cliques.

The benchmark study indicates that our proposed method for table multiplication and marginalization performs well for small, medium, and large tables. However, our real interest is in the performance on massive tables, which is impossible to benchmark in this paper due to the increased running time and memory usage of **gRbase** and **base R**. For pedigree networks e.g., the CPTs can be huge. Thus, it is possible to construct sparse tables without representing the dense arrays. Finally, we mention that the benchmark did not consider tables where the domain of one of the tables is contained in the domain in the second table. As discussed in Section 5, these are the typical cases in JTA and the performance of function `mult` from **sparta**, in these cases, is considerably faster.

## Computational details

Detailed examples, source code, and information about installation of packages **jti** and **sparta** can be found at <https://github.com/mlindsk/jti> and <https://github.com/mlindsk/sparta>. **jti** has a GNU general public license, whereas **sparta** has a MIT license.

All computations were carried out on a 64-bit Linux computer with Ubuntu 20.04.2 and Intel Core i7-6600U CPU 2.60GHz LTS. The machine has approximately 6Gb of free memory for use in calculations.

## Acknowledgments

We are thankful for the constructive comments from the anonymous reviewers. The paper has been greatly improved by these.

## References

- Aragam B, Gu J, Zhou Q (2019). “Learning Large-Scale Bayesian Networks with the **sparsebn** Package.” *Journal of Statistical Software*, **91**(11), 1–38. doi:10.18637/jss.v091.i11.
- Boettcher SG, Dethlefsen C (2003). “**deal**: A Package for Learning Bayesian Networks.” *Journal of Statistical Software*, **8**(20), 1–40. doi:10.18637/jss.v008.i20.
- Boutilier C, Friedman N, Goldszmidt M, Koller D (2013). “Context-Specific Independence in Bayesian Networks.” *arXiv 1302.3562*, arXiv.org E-Print Archive. doi:10.48550/arxiv.1302.3562.
- Dethlefsen C, Højsgaard S (2005). “A Common Platform for Graphical Models in R: The **gRbase** Package.” *Journal of Statistical Software*, **14**(17), 1–12. doi:10.18637/jss.v014.i17.
- Eddelbuettel D, Sanderson C (2014). “**RcppArmadillo**: Accelerating R with High-Performance C++ Linear Algebra.” *Computational Statistics & Data Analysis*, **71**, 1054–1063. doi:10.1016/j.csda.2013.02.005.
- Flores MJ, Gámez JA (2007). “A Review on Distinct Methods and Approaches to Perform Triangulation for Bayesian Networks.” In P Lucas, JA Gámez, A Salmerón (eds.), *Advances*

- in *Probabilistic Graphical Models*, pp. 127–152. Springer-Verlag, Berlin, Heidelberg. doi: [10.1007/978-3-540-68996-6\\_6](https://doi.org/10.1007/978-3-540-68996-6_6).
- Franzin A, Sambo F, di Camillo B (2017). “**bnstruct**: An R Package for Bayesian Network Structure Learning in the Presence of Missing Data.” *Bioinformatics*, **33**(8), 1250–1252. doi: [10.1093/bioinformatics/btw807](https://doi.org/10.1093/bioinformatics/btw807).
- Gómez-Olmedo M, Cabañas R, Cano A, Moral S, Retamero OP (2021). “Value-Based Potentials: Exploiting Quantitative Information Regularity Patterns in Probabilistic Graphical Models.” *International Journal of Intelligent Systems*, **36**(11), 6913–6943. doi: [10.1002/int.22573](https://doi.org/10.1002/int.22573).
- Højsgaard S (2012). “Graphical Independence Networks with the **gRain** Package for R.” *Journal of Statistical Software*, **46**(10), 1–26. doi: [10.18637/jss.v046.i10](https://doi.org/10.18637/jss.v046.i10).
- Højsgaard S, Edwards D, Lauritzen S (2012). *Graphical Models with R*. Use R!, 1st edition. Springer-Verlag. doi: [10.1007/978-1-4614-2299-0](https://doi.org/10.1007/978-1-4614-2299-0).
- Konis K (2017). **RHugin**. R package version 8.4, URL <https://rhugin.R-Forge.R-project.org/>.
- Lauritzen SL (1996). *Graphical Models*, volume 17 of *Oxford Statistical Science Series*. Clarendon Press.
- Lauritzen SL, Spiegelhalter DJ (1988). “Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems.” *Journal of the Royal Statistical Society B*, **50**(2), 157–194. doi: [10.1111/j.2517-6161.1988.tb01721.x](https://doi.org/10.1111/j.2517-6161.1988.tb01721.x).
- Lepar V, Shenoy PP (1998). “A Comparison of Lauritzen-Spiegelhalter, Hugin, and Shenoy-Shafer Architectures for Computing Marginals of Probability Distributions.” In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pp. 328–337. Morgan Kaufmann Publishers. doi: [10.5555/2074094.2074133](https://doi.org/10.5555/2074094.2074133).
- Lindskou M (2024a). **jti**: *Junction Tree Inference*. doi: [10.32614/CRAN.package.jti](https://doi.org/10.32614/CRAN.package.jti). R package version 1.0.0.
- Lindskou M (2024b). **sparta**: *Sparse Tables*. doi: [10.32614/CRAN.package.sparta](https://doi.org/10.32614/CRAN.package.sparta). R package version 1.0.1.
- Maathuis M, Drton M, Lauritzen S, Wainwright M (2018). *Handbook of Graphical Models*. 1st edition. Chapman & Hall/CRC, Boca Raton. doi: [10.1201/9780429463976](https://doi.org/10.1201/9780429463976).
- Masante D (2020). **bnsatial**: *Spatial Implementation of Bayesian Networks and Mapping*. R package version 1.1.1, URL <https://CRAN.R-project.org/package=bnsatial>.
- Mihaljević B, Bielza C, Larrañaga P (2018). “**bnclassify**: Learning Bayesian Network Classifiers.” *The R Journal*, **10**(2), 455–468. doi: [10.32614/rj-2018-073](https://doi.org/10.32614/rj-2018-073).
- Moharil J (2016). **geneNetBP**: *Belief Propagation in Genotype-Phenotype Networks*. R package version 2.0.1, URL <https://CRAN.R-project.org/package=geneNetBP>.
- Pearl J (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, San Francisco. doi: [10.5555/534975](https://doi.org/10.5555/534975).

- R Core Team (2024). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.
- Scutari M (2010). “Learning Bayesian Networks with the **bnlearn** R Package.” *Journal of Statistical Software*, **35**(3), 1–22. doi:10.18637/jss.v035.i03.
- Yu H, Moharil J, Blair R (2020). “**BayesNetBP**: An R Package for Probabilistic Reasoning in Bayesian Networks.” *Journal of Statistical Software*, **94**(3), 1–31. doi:10.18637/jss.v094.i03.

**Affiliation:**

Mads Lindskou, Torben Tvedebrink, Poul Svante Eriksen, Søren Højsgaard  
Department of Mathematical Sciences  
Aalborg University  
Skjernvej 4A, DK-9220 Aalborg Øst, Denmark  
E-mail: [mads@math.aau.dk](mailto:mads@math.aau.dk), [tvede@math.aau.dk](mailto:tvede@math.aau.dk), [svante@math.aau.dk](mailto:svante@math.aau.dk),  
[sorenh@math.aau.dk](mailto:sorenh@math.aau.dk)  
URL: <https://github.com/mlindsk>

Niels Morling  
Department of Mathematical Sciences  
Aalborg University  
Skjernvej 4A, DK-9220 Aalborg Øst  
*and*  
Section of Forensic Genetics  
Department of Forensic Medicine  
Faculty of Health and Medical Sciences  
University of Copenhagen  
11 Frederik V's Vej, DK-2100 Copenhagen, Denmark  
E-mail: [niels.morling@sund.ku.dk](mailto:niels.morling@sund.ku.dk)