Abstract

A graphical model is a multivariate (potentially very high dimensional) probabilistic model, which is formed by combining lower dimensional components. Inference (computation of conditional probabilities) is based on message passing algorithms that utilize conditional independence structures. In graphical models 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 inference, such tables must be multiplied which can lead to even larger tables. The sparta package meets this challenge by implementing methods that efficiently handles multiplication and marginalization of sparse tables. The package was written in the R programming language and is freely available from the Comprehensive R Archive Network (CRAN). The companion package jti, also on CRAN, was developed to showcase the potential of sparta in connection to the Junction Tree Algorithm. We show, that jti is able to handle highly complex graphical models which are otherwise infeasible due to lack of computer memory, using sparta as a backend for table operations.

Keywords: Bayesian network, graphical model, sparse table, R, C++.

1. Introduction

A multivariate probability distribution for discrete variables with finite state spaces can be represented by a multi-dimensional array. However, for high-dimensional distributions where each variable may have a large state space, lack of computer memory can become a problem. For example, an 80-dimensional random vector in which each variable has 10 levels will lead to a state space with $10^{80}$ cells. Such a distribution can not be stored 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. We describe here the R (R Core Team 2020) package sparta (Lindskou 2020b) for efficient multiplication and marginalization.
of sparse tables.

The family of graphical models (Pearl 2014; Lauritzen 1996; Højsgaard, Edwards, and Lauritzen 2012) is vast, and includes many different models. In this paper, we consider Bayesian networks and decomposable undirected graphical models. Undirected graphical models are also known as Markov random fields (MRFs). Decomposability is a property ensuring a closed form of the maximum likelihood parameters. Graphical models, enjoy the property that conditional independencies can be read off from a graph consisting of nodes, representing the random variables. In Bayesian networks, edges are directed from a node to another and represents a directed connection. In a MRF the edges are undirected and these should be regarded as associations between pairs of nodes in a broad sense. A decomposable MRF can be turned into a Bayesian network while retaining the correct likelihood function. However, one can no longer interpret conditional independencies in this transformed graph. This transformation is merely a step towards making inference in decomposable MRFs and as such, we shall mainly focus on introducing concepts regarding Bayesian networks.

We illustrate sparta using the jti package (Lindskou 2020a) which implements the Junction Tree Algorithm (JTA) for discrete variables using the Lauritzen-Spiegelhalter updating scheme (Lauritzen and Spiegelhalter 1988) with table operations relying on sparta. JTA is used for inference in Bayesian networks (BNs). We also show, that multiplication and marginalization of sparse tables is fast compared with standard built-in R functions and is comparable to gRbase when the tables are sparse.

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 2014) where the latter is not on the Comprehensive R Archive Network (CRAN). The only R package on CRAN that has an API for (dense) table operations is gRbase, which gRain depends upon. Some R packages that rely on gRain and gRbase are geneNetBP (Moharil 2016) and bnspatial (Masante 2020). The bnclassify package (Mihaljević, Bojan, Bielza, Concha, Larrañaga, and Pedro 2018) has a lower C++ class implementation of conditional probability tables. Our goal is twofold: Firstly, to provide an efficient back-end for sparse table operations for other R packages. Secondly, to provide a new implementation of JTA, namely jti, using sparta as a back-end.

In Section 2 we introduce basic notation and terminology and in Section 3 we motivate the usage 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 how to use sparta. Section 6 outlines how to use jti and gives specific examples, using two BNs which are well known from the literature. Moreover, we demonstrate the strength of sparta using the ess (Lindskou 2021) package to fit a decomposable MRF on real data. 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 from the ReppArmadillo 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 distribution for the random vector \( X = X_V = (X_v \mid v \in V) \) where \( V \) is a set of labels of the random variables \( X_v, v \in V \) and where each \( X_v \) has finite state space. A realized value \( x = (x_v)_{v \in V} \) is called a cell. If we denote by \( I_v \) the finite state space of variable \( X_v \), the domain of \( p \) is given by

\[
I = \times_{v \in V} I_v.
\]

Given a subset \( A \) of \( V \), the \( A \)-marginal cell of \( x_V \) 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 together with a joint probability of the form

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

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 said to be 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 \( |I_v| \cdot |I_{pa(v)}| \). 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 for which a potential depends on. 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

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

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

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

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

\[
\phi_A^\downarrow(x_{A \setminus B}) = \phi_A(x_{A \setminus B}, x^*_B).
\]
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:

\[
p(x_a, x_b, x_c, x_d, x_e) = p(x_c)p(x_a | x_c)p(x_b | x_a, x_d)p(x_d | x_c)p(x_e | x_c, x_d)
\]  

(2)

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

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

![Figure 1](image)

The Junction Tree Algorithm (JTA) can be seen as an algorithm for automatically factorizing in order 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 et al. 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 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.

A graph is triangulated if it has no cycles of length greater than 3. If such cycles are present, we must add **fill edges** to produce a triangulated graph. A triangulated graph is also called **decomposable**, and hence the connection to decomposable MRFs shows here. Finding an
optimal triangulation (in terms of minimizing the number of fill edges) is a NP hard problem, but there exists good heuristic methods, see Flores and Gámez (2007). Finding a good triangulation can have huge impact on the performance of JTA which we detail in Section 6.2. A moralized and triangulated version of the graph in Figure 1(a) is shown Figure 1(b), where no fill edges 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 (2) to a clique potential for which the variables conform, e.g.

\begin{align*}
\phi_{C_1}(x_a, x_b, x_d) &\leftarrow p(x_a \mid x_b, 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{align*}

The clique potentials are now initialized (we also say that the network has been initialized) and note that the clique potentials in general do not have any interpretation at this stage.

We have obtained the clique potential representation

\begin{equation}
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_a, x_c, x_d).
\end{equation}

The network is said to be compiled at this stage, i.e. when moralization and triangulation has been performed and the clique potential representation is obtained. In complex networks with large clique potentials it might not be feasible to even initialize the clique potentials due to lack of memory. We give an example of how to overcome this in Section 6.2 by entering evidence into the model as described in Section 3.1.

Next, the message passing scheme can now be applied to the junction tree. We describe here, 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 an inward message by setting

\begin{equation}
\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)
\end{equation}

and update the leaf node as

\begin{equation}
\phi_{C_3}(x_c, x_d, x_e) \leftarrow \frac{\phi_{C_3}(x_c, x_d, x_e)}{\phi_{S_{32}}(x_c, x_d)},
\end{equation}

where $0/0 := 0$. We say that $C_2$ have 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
We say that \( C \) perform another inwards message by setting \( \phi_{S_2} (x_c) = \sum_{x_a, x_d} \phi_{C_2} (x_a, x_c, x_d) \) and update:

\[
\phi_{C_1} (x_a, x_b, x_d) \leftarrow \phi_{C_2} (x_a, x_c, x_d) \phi_{S_2} (x_c, x_d),
\]

\[
\phi_{C_3} (x_a, x_c, x_d) \leftarrow \phi_{C_2} (x_a, x_c, x_d) / \phi_{S_2} (x_c, x_d).
\]

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 what is called the *set chain representation*

\[
p(x_a, x_b, x_c, x_d) = \phi_{C_1} (x_a, x_b, x_d) \phi_{C_2} (x_a, x_c, x_d) \phi_{C_3} (x_c, x_d, x_e)
\]

\[
= p(x_a, x_b, x_d) p(x_a | x_c, x_d) p(x_c, x_d | x_e),
\]

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 *outward phase* we start by sending messages from the root by performing an *outward message* by letting \( \phi_{S_1} (x_a, x_d) = \sum_{x_b} \phi_{C_1} (x_a, x_b, x_d) \) and update:

\[
\phi_{C_2} (x_a, x_c, x_d) \leftarrow \phi_{C_2} (x_a, x_c, x_d) \phi_{S_1} (x_c, x_d). \tag{5}
\]

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 \( \phi_{S_2} (x_c, x_d) = \sum_{x_a} \phi_{C_2} (x_a, x_c, x_d) \) and update \( \phi_{C_3} \) as

\[
\phi_{C_3} (x_c, x_d, x_e) \leftarrow \phi_{C_3} (x_c, x_d, x_e) \phi_{S_2} (x_c, x_d).
\]

As a consequence we finally obtain the *clique marginal representation*

\[
p(x_a, x_b, x_c, x_d, x_e) = \phi_{C_1} (x_a, x_b, x_d) \phi_{C_2} (x_a, x_c, x_d) \phi_{C_3} (x_c, x_d, x_e)
\]

\[
\phi_{S_1} (x_a, x_d) \phi_{S_2} (x_c, x_d)
\]

\[
= p(x_a, x_b, x_d) p(x_a, x_c, x_d) p(x_c, x_d, x_e) / p(x_a, x_d) p(x_c, x_d).
\]

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, Lepard and Shenoy
(1998) recorded a total number of 2,371,178 binary operations. We do not intend to follow the same analysis here, but for sparse tables 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 (3) 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 $x_E^*$ 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 a higher degree of sparsity. This is how evidence is handled in jti. In fact, this is the reason why jti is able to handle very complex networks by exploiting evidence using sparse tables from sparta. It is also possible to enter evidence into the factorization (2). This is the key to handle complex networks that are otherwise infeasible due to lack of memory as we show in Section 6.2. This trick is related to cutset conditioning and if one can enter evidence into variables that breaks cycles the effect can be huge (Pearl 2013). One can exploit the law of total probability to sum out the variables that was conditioned on (since we obtain the probability of the evidence during propagation) and recover whatever probability is of interest.

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
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
If we convert \( f \) and \( g \) to `data.frame` objects and exclude the cases with a value of zero we have:

```r
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)
X Y Z Freq
x1 y1 z1 5
x2 y1 z1 4
x2 y2 z1 7
x2 y1 z2 9
R> print(dg, row.names = FALSE)
Y Z W Freq
y1 z1 w1 7
y2 z1 w1 6
y2 z2 w1 6
y1 z2 w2 9
```

which literally leaves us with two sparse tables, \( df \) and \( dg \) respectively. In order to multiply \( df \) by \( dg \) we must, by definition, determine the cases that match on the variables \( Y,Z \) 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 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 bench-marking):

```r
R> sparse_prod <- function(df, dg){
+     S <- setdiff(intersect(names(df), names(dg)), "Freq")
+     mrg <- merge(df, dg, by = S)
+     val <- mrg$Freq.x * mrg$Freq.y
+     mrg$val <- val
+     mrg[, setdiff(names(mrg), c("Freq.x", "Freq.y"))]  
+ }
```

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 \( I = \times_{v \in V} I_v \) as described in 2. Define the level set \( \mathcal{L} := \times_{v \in V} \mathcal{L}_v \) where \( \mathcal{L}_v = \{1, 2, \ldots, |I_v|\} \) and let \( \# : I \to \mathcal{L} \) 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 \( i = (i_1, i_2, \ldots, i_{|V|}; i_v \in \mathcal{L}_v) \) 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 \( f \) 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 we can choose \# 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 \(I = \{(1, 1, 1), (2, 1, 1), (2, 2, 1), (2, 1, 2)\}\). Hence
\[
\Phi = \begin{bmatrix} 1 & 2 & 2 & 2 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix},
\]
with \(\phi = (5, 4, 7, 9)\), corresponding to \(\text{df}\) 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 \(J\). 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 transform \(\Phi\) into a look-up table\(^1\) 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
\[
M_S(\Phi)[k] = \{j \in \{1, 2, \ldots, |I|\} : \Phi[j] = k\}.
\]

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

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 \(k \in K\). 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, \(K\), 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 changing 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
\[
J = \{w_1, w_2\} \times \{y_1, y_2\} \times \{z_1, z_2\}.
\]

\(^1\)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 where the names are the keys and the values are the elements of the list.
Algorithm 1 Multiplication of Sparse Tables

1: procedure $(\tau = (\Phi, \phi), \gamma = (\Psi, \psi))$: sparse table
2: $S := V \cap U$
3: $K$: Mutual keys of $M_S(\Phi)$ and $M_S(\Psi)$
4: $N := \sum_{k \in K} |M_S(\Phi)[k]| \cdot |M_S(\Psi)[k]|$
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 $k \in K$ 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

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

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

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

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

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 value $\{1, 2\}$ which 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 \texttt{sparse_mult}. 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 is more demanding since we have to reconstruct zero-cells if one of the tables have 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 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

\begin{algorithm}
\caption{Marginalization of Sparse Tables}
\begin{algorithmic}[1]
\Procedure{\( \tau = (\Phi, \phi) \)}{sparse table, \( A \): Set of labels}
\State Construct \( H_A(\Phi) \)
\State \( N = |H_A(\Phi)| \)
\State Initialize the matrix \( \Phi^{\downarrow A} \) with \(|A|\) rows and \( N \) columns
\State Initialize the vector \( \phi^{\downarrow A} \) of dimension \( N \)
\State Let \( K \) be the keys of \( H_A(\Phi) \)
\State \( l := 1 \)
\For{\( k \in K \)}
\State \( (j, v) := H_A(\Phi)[k] \)
\State \( \Phi^{\downarrow A}[l] := \Phi_A[j] \) \Comment{deduced by picking elements from \( \Phi[j] \)}
\State \( \phi^{\downarrow A}_l := v \)
\State \( l = l + 1 \)
\EndFor
\Return \( (\Phi^{\downarrow A}, \phi^{\downarrow A}) \)
\EndProcedure
\end{algorithmic}
\end{algorithm}

sparse table \( \tau \) of \( T \) and let \( A = \{y, z\} \). Then, the resulting sparse marginal table have 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

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

and $\psi^A = (9, 7, 9)$. 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.

5.1. Sparse tables in JTA

Triangulation may lead to cliques for which no potential is assigned during compilation and the clique potential is then regarded as the unity table. In the dense version, this corresponds to an array filled with ones. However, the idea of sparsity will be ruined if we just adapt the dense version. For the general case we define a sparse unity table and show how to multiply it with a non-unity table in Appendix A. In the light of message passing, these unity tables are not needed. When a message is sent to a unity, we simply replace it with the message. And as such, we avoid a lot of computational overhead.

5.2. How to use sparta

In order 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
R> ftable(f, row.vars = "X")

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

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

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
R> sf <- as_sparta(f); sg <- as_sparta(g)

Printing sf

R> print.default(sf)
```
where we use the default print method to look under the hood. 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, 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 2. Using the regular 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
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 5.3. Converting `sf` into a conditional probability table (CPT) with conditioning variable `Z`:

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

We can further slice `sf_cpt` to obtain a probability distribution conditioned on $Z = z1$

```r
R> slice(sf_cpt, s = c(Z = "z1"))
```

Marginalizing out $Y$ in `sg` yields

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

which 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 can be necessary to use if the corresponding dense table is too large to have in memory. A table with relevant methods in `sparta` is given in Table 1.
### sparta: Sparse Tables

| Function name                  | Description                                                                 |
|-------------------------------|-----------------------------------------------------------------------------|
| as_<sparta>                   | Convert array-like object to a sparta object                                |
| as_<array/df/cpt>             | Convert sparta object to an array/data.frame/CPT                           |
| sparta_struct                 | Constructor for sparta objects                                             |
| mult, div, marg, slice        | Multiply/divide/marginalize/slice                                           |
| normalize                     | Normalize (the values of the result sum to one)                            |
| get_val                       | Extract the value for a specific named cell                                |
| get_cell_name                 | Extract the named cell                                                     |
| get_values                    | Extract the values                                                         |
| dim_names                     | Extract the domain                                                         |
| names                         | Extract the variable names                                                 |
| max/min                       | The maximum/minimum value                                                  |
| which_<max/min>_cell          | The column index referring to the max/min value                            |
| which_<max/min>_idx           | The configuration corresponding to the max/min value                        |
| sum                           | Sum the values                                                             |
| equiv                         | Test if two tables are identical up to permutations of the columns         |

Table 1: Description of essential functions from the sparta package.

#### 5.3. When to Use Sparta

As shown in Section 5.2, 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 and 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 are implicitly reflected by the memory needed to store the dense table. The more memory needed, the bigger state space of the variables. However, the more variables and the bigger state space of these 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 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. In Section 6.1 we fit a decomposable MRF to real data and show that the sparsity of the clique potentials is much favorable towards sparta.
Figure 2: The black identity line indicates the number of gigabytes needed to store the dense table with $x$ elements. The colored lines indicates the number of gigabytes required to store the equivalent sparta object with the respective number of variables and sparsity.

6. Usecases of jti and sparta

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 using the classic Bayesian network asia (Lauritzen and Spiegelhalter 1988) where the corresponding CPTs is part of jti. The network represents a simplified model to help diagnose the patients arriving at a respiratory clinic. A history of smoking has a direct influence on both whether or not a patient has bronchitis and whether or not a patient has lung cancer. Both lung cancer and bronchitis can result in dyspnoea. An x-ray result depends on the presence of either tuberculosis or lung cancer. Finally, a visit to Asia influence the probability of having tuberculosis. The DAG is depicted in Figure 3.

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

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 )

<cpt_list, list>
-------------------------

From the output we see the inferred CPTs corresponds to Figure 3 giving rise to a factorization in the same way as in (2). The network is now ready for compilation which involves moralization and triangulation. In jti there are four different 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. In Section 6.2 we show how to exploit information about the cliques with largest state space in order to compile and propagate in a network that is otherwise infeasible on a ’standard’ laptop. Now, let 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
-------------------------
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 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`.

![Figure 4: A junction tree for the asia network.](image-url)
6.1. Inference in Decomposable Markov Random Fields

In this section we illustrate the gain of using sparta in connection with the public-domain derma data set (part of the ess package) originally obtained from the UCI Machine Learning Repository (Dua and Graff 2017). The data set consists of 358 observations, 35 variables of which there is one class variable called ES. The class variable has six states; each representing a skin disease. The remaining 34 clinical variables (all with 4 states except age which has been discretized into six bins) are used to predict the skin disease. We first fit a decomposable MRF

\[
R> g \leftarrow \text{ess::fit\_graph(derma, } q = 1.5, \text{ sparse_qic = TRUE)}
\]

The fitted graph is plotted in Figure 5, where we just notice, that it is a rather complex graph with large cliques. Hence, for such a small data set and a graph with big cliques, the chance of sparse clique potentials is significant. We can now convert \( g \) into a Bayesian network simply by entering into the \texttt{cpt\_list} function and from there compile and propagate the network:

\[
R> cl \leftarrow \text{jtii::cpt\_list(derma, } \text{ess::as\_igraph(g)) } \\
R> j \leftarrow \text{jtii::jt(jtii::compile(cl, root\_node = } "\text{ES}\text{"), propagate = } "\text{no}\text{")}
\]

The argument \texttt{root\_node = } "ES" forces ES into the root clique. Hence, if we are interested in queries about ES we only need to conduct the inwards message passing, collect, since the root clique potential is identical to the probability distribution over the variables in the root clique. The argument \texttt{propagate = } "no" means that messages passing should be postponed. By doing so, initialization is only performed once and the junction tree can be exploited as many times as needed with different evidence.
The largest clique of \( j \) has 6 variables and the mean sparsity of the clique potential is given by

\[
R> \text{mean(sapply(jchargeC, sparta::sparsity))}
\]

[1] 0.9233838

showing that the tables are extremely sparse. The junction tree can now be used for e.g. classification. Consider the first observation in \texttt{derma}

\[
R> z <- \text{unlist(derma[1, -ncol(derma)])}
\]

and update the junction tree with evidence corresponding to \( z \) and perform inwards message passing

\[
R> jz <- \text{jti::propagate(jti::set_evidence(j, z), prop = "collect")}
\]

The probability distribution of the class variable given the evidence \( z \) is then

\[
R> q <- \text{jti::query_belief(jz, nodes = "ES", type = "marginal")[[1]]}
R> q[which.max(q)]
\]

seboreic dermatitis

1

Hence, we classify \( z \) as an observation from class seboreic dermatitis.

6.2. The Impact of Evidence

The Bayesian Network \texttt{Link} (Jensen and Kong 1999) is a large Bayesian network with 724 nodes and 1,125 arcs. The network has been used for linkage analysis where, previously, only approximate methods has been applied to make inference in the network.

We have verified that \texttt{sparta} generates a large amount of overhead in the CPTs for the \texttt{Link} network. In fact, \texttt{jti} uses approximately 16 times more memory to store the clique potentials than the dense version. However, in the following we show how \texttt{sparta} leverages from entering evidence and hence reducing the CPTs.

We picked a handful of the most comprehensive software packages, across different programming languages, for belief propagation and test if they were able to handle \texttt{Link} on a “standard” laptop machine. We used the R package \texttt{gRain}, the R package \texttt{BayesNetBP}, the Python package \texttt{pyArum} (Gonzales, Torti, and Wuillemin 2017), the Julia package \texttt{BayesNet} (Stanford 2020). Interestingly, all of these failed due to lack of computer memory on the first authors machine\(^2\) while \texttt{jti} succeeded. The \texttt{pyAgrum} package is a high-level interface to the

\(^2\)See Section 8 for details about this machine.
extremely efficient C++ library **aGrum** (Gonzales et al. 2017). In Gonzales et al. (2017) the authors of **pyAgrum** themselves failed to propagate in **Link** on a computer with 32Gb memory. An investigation revealed that exploiting the triangulation found by **jti, gRain** is now also able to make inference in **Link**. We have not tried this with the other packages mentioned but conjecture, that the other packages would also be able to handle the **Link** network. Interestingly, this triangulation was the result of one of the most well-known heuristics, namely **min_fill**. The reason that triangulation have such a big impact is, that it determines the resulting cliques and hence the size of the clique potentials. If the cliques are too big, and hence the number of elements in the state space is large, it may not be possible to initialize the clique potentials due to lack of memory.

We now compare the cliques with large state spaces resulting from the two heuristics **min_fill** and **min_nei** (a variant of **min_fill** which also tries to minimize the number of fill-in edges). We have extracted the **Link** network from [https://www.bnlearn.com/bnrepository/](https://www.bnlearn.com/bnrepository/) as a list of CPTs with information about child and parent-node relations. In **jti** we just need a list of CPTs which we extract as

```r
R> cpts <- bnfit_to_cpts(Link)
```

Next, we convert these CPTs to their **sparta** equivalent using **cpt_list** while deducing the underlying network, conducting some sanity checks and triangulate with the two heuristics

```r
R> cl <- cpt_list(cpts)
R> tri_min_fill <- triangulate(cl, tri = "min_fill")
R> tri_min_nei <- triangulate(cl, tri = "min_nei")
```

The size of the five largest clique state spaces can then be extracted as

```r
R> sp_min_fill <- sort(tri_min_fill$statespace, decreasing = TRUE)[1:5]
R> sp_min_nei <- sort(tri_min_nei$statespace, decreasing = TRUE)[1:5]
```

and the memory usage, for dense tables, in gigabytes of allocating memory for each of these are

```r
R> round(sum(sp_min_fill) / 1e9*8, 2)
[1] 0.2
```

```r
R> round(sum(sp_min_nei) / 1e9*8, 2)
[1] 26.98
```
This indicates that any software for message passing should be able to handle the link network on a standard laptop using the \texttt{min\_fill} heuristic from \texttt{jit}, but also that \texttt{min\_nei} will fail. We imagine that there is no triangulation of \texttt{Link}, such that any software packages, including \texttt{jit}, are able to compile and propagate in \texttt{Link} on a standard laptop. In what follows we show, that by exploiting evidence and sparsity, we can overcome the initialization step by using \texttt{jit}. We use the \texttt{min\_nei} heuristic since this was the worst, locate one of the cliques with largest state space and insert evidence on 10 nodes.

\begin{verbatim}
R> idx_max <- which.max(tri_min_nei$statespace)
R> max_vars <- tri_min_nei$cliques[[idx_max]]
R> max_dim <- jti::dim_names(cl)[max_vars[1:10]]
R> e <- sapply(max_dim, \texttt{\textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbackslash \textbacklash}
(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 \texttt{gRbase} is the standard and most mature package for graphical models in R, the performance comparisons are relative to that of \texttt{gRbase}. Hence, in Figure 6, \texttt{gRbase} performs better for tables with the 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 6 describes the size of the table resulting from multiplying two tables. It can be seen that \texttt{R} consistently produces tables of smaller sizes than \texttt{sparta} for very small tables with 100 \((10^2)\) cells. For tables with more than 100 cells, \texttt{sparta} consistently produces smaller tables except for a single case. Increasing the degree of sparsity leads to reduced object sizes for both \texttt{R} and \texttt{sparta}, and for tables with more than 75% sparsity, \texttt{sparta} outperforms both \texttt{R} and \texttt{gRbase} except in the first two panels with small tables.

**Multiplication (time):** The second row of Figure 6 describes the computing time for multiplying two tables. Clearly \texttt{sparta} outperforms \texttt{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.

**Marginalisation (time):** The third row of Figure 6 describes the computing time for marginalising a table. When the degree of sparsity increases, the computing time decreases. In the comparison between \texttt{gRbase} and \texttt{sparta}, we see that the marginalization implementation of \texttt{sparta} is competitive to that of \texttt{gRbase}. For tables with 10,000 \((10^4)\) or fewer cells it is faster irrelevant of the sparsity except in a single case. For 75% sparsity \texttt{sparta}’s marginalization is consistently faster.

This small, and non-exhaustive, 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 \texttt{gRbase} and \texttt{R}.
Figure 6: Relative comparison of 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 shows the relative timing for multiplication and marginalizaion, respectively.

8. Summary

We have presented a novel method for 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 presented the companion package jti to illustrate some of the advantages of sparta in connection to the Junction Tree Algorithm. We hope to explore the benefit from the C API for working with external pointers in order to reduce the memory usage for sparta objects in the future.

The memory footprint of the clique potentials can become prohibitively large when the size 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 variables in such a clique. Finally, we hope to explore pedigree networks that can potentially benefit immensely from sparta in the future.
Computational details

In addition to the packages already mentioned, the following R packages was used to make the benchmark results

- dplyr version 1.0.6
- glue version 1.4.2
- tictoc version 1.0
- ggplot2 version 3.3.3

We used R version 4.1.0. All computations were carried out on a 64-bit Linux computer with Ubuntu 20.04.2 and Intel(R) Core(TM) i7-6600U CPU 2.60GHz LTS. The machine has approximately 6Gb of free memory for use in calculations.

A. Sparse unity tables

We define a sparse unity table, $\mu$, over the variables $U$ to equal the set of level sets $\{L_u\}_{u \in U}$ induced by the domain $J$. Multiplying a sparse table, $\tau = (\Phi, \phi)$, defined over the set of labels $V$ with sparse domain $I$ and the sparse unity table, $\mu$, can be performed as follows. Define $R = U \setminus V$ and $L_R = \times_{r \in R} L_r$ and initialize the product matrix $\Pi$ with $|V| + |R|$ rows and $|I| \cdot |L_R|$ columns. In other words, $\Pi$ has $|L_R|$ times more columns than $\Phi$. For $j = 1, 2, \ldots, |I|$ and $i_R \in L_R$ append $(\Phi[j], i_R)$ to $\Pi$ and set the corresponding value in $\pi$ to $\phi_j$. That is, we multiply a sparse table with a sparse unity table on the fly. When $|I| \gg |I|$ this approach greatly reduces the number of binary operations compared to the dense case; in fact, there are no binary operations. The procedure is summarized in Algorithm 3. However, dividing $\mu$ with $\tau$, one must change line 10 in Algorithm 3 with $\pi_l := 1/\phi_j$. 

Algorithm 3 Multiplication of a Sparse Table with a Sparse Unity Table

1: procedure $(\tau = (\Phi, \phi):$ sparse table, $\mu:$ sparse unity table)
2: $R := U \setminus V$
3: Set $N = |\mathcal{I}| \cdot |\mathcal{L}_R|$
4: Initialize the matrix $\Pi$ with $|\mathcal{V}| + |\mathcal{R}|$ rows and $N$ columns
5: Initialize the vector $\pi$ of dimension $N$
6: $l := 1$
7: for $j = 1, 2, \ldots, |\mathcal{I}|$ do
8:     for $i_R \in \mathcal{L}_R$ do
9:         $\pi[l] := (\Phi[j], i_R)$
10:         $\pi[l] := \phi_j$
11:         $l := l + 1$
12:     end for
13: end for
14: return $(\Pi, \pi)$
15: end procedure

References

Dua D, Graff C (2017). “UCI Machine Learning Repository.” URL http://archive.ics.uci.edu/ml.

Eddelbuettel D, Sanderson C (2014). “RcppArmadillo: Accelerating R with high-performance C++ linear algebra.” Computational Statistics and Data Analysis, 71, 1054–1063. URL http://dx.doi.org/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 Advances in Probabilistic Graphical Models, pp. 127–152. Springer.

Gonzales C, Torti L, Wuillemin PH (2017). “aGrUM: a graphical universal model framework.” In International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, pp. 171–177. Springer.

Højsgaard S, Edwards D, Lauritzen S (2012). Graphical Models with R. Springer Science & Business Media.

Højsgaard S (2012). “Graphical Independence Networks with the gRain Package for R.” Journal of Statistical Software, Articles, 46(10), 1–26. ISSN 1548-7660. doi:10.18637/jss.v046.i10. URL https://www.jstatsoft.org/v046/i10.

Jensen CS, Kong A (1999). “Blocking Gibbs Sampling for Linkage Analysis in Large Pedigrees with Many Loops.” The American Journal of Human Genetics, 65(3), 885 – 901. ISSN 0002-9297. doi:https://doi.org/10.1086/302524. URL http://www.sciencedirect.com/science/article/pii/S0002929707623398.
Konis K (2014). *RHugin: R Package Version 7.8*. URL http://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: Series B (Methodological)*, 50(2), 157–194.

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 Inc.

Lindskou M (2020a). *jti: Junction Tree Inference*. R package version 0.7.1, URL https://CRAN.R-project.org/package=jti.

Lindskou M (2020b). *sparta: Sparse Tables*. R package version 0.7.3, URL https://CRAN.R-project.org/package=sparta.

Lindskou M (2021). *ess: Efficient Stewise Selection in Decomposable Markov Random Fields*. R package version 1.1.0, URL https://CRAN.R-project.org/package=ess.

Masante D (2020). *bnspatial: Spatial Implementation of Bayesian Networks and Mapping*. R package version 1.1.1, URL https://CRAN.R-project.org/package=bnspatial.

Mihaljević, Bojan, Bielza, Concha, Larrañaga, Pedro (2018). “bnclassify: Learning Bayesian Network Classifiers.” *The R Journal*, 10(2), 455–468. URL https://doi.org/10.32614/RJ-2018-073.

Moharil J (2016). *Belief Propagation in Genotype-Phenotype Networks*. R package version 2.0.1, URL https://CRAN.R-project.org/package=geneNetBP.

Pearl J (2013). “A constraint propagation approach to probabilistic reasoning.” *arXiv preprint arXiv:1304.3422*.

Pearl J (2014). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Stanford ISL (2020). *A Julia package for Bayesian Networks*. Julia package version v.3.3.2, URL https://github.com/sisl/BayesNets.jl.
Yu H, Moharil J, Blair R (2020). “BayesNetBP: An R Package for Probabilistic Reasoning in Bayesian Networks.” *Journal of Statistical Software, Articles*, 94(3), 1–31. ISSN 1548-7660. doi:10.18637/jss.v094.i03. URL https://www.jstatsoft.org/v094/i03.