

UNIVERSITY OF GRANADA

Department of Computer Science and
Artificial Intelligence



PhD Programme

Probabilistic Models for Artificial Intelligence
and Data Mining

PhD Thesis Dissertation

**New data structures and algorithms
for uncertainty treatment
with Probabilistic Graphical Models**

PhD Student

Cora Beatriz Pérez-Ariza

Advisors

**Andrés Cano Utrera, Manuel Gómez-Olmedo and
Antonio Salmerón**

Granada, June 2013

Editor: Editorial de la Universidad de Granada
Autor: Cora Beatriz Pérez-Ariza
D.L.: GR 383-2014
ISBN: 978-84-9028-780-4

Wyrð bið ful aræð!
(*fate remains wholly inexorable*)

Agradecimientos

Quiero comenzar dándole las gracias a mis directores de tesis Andrés Cano Utrera, Manuel Gómez Olmedo y Antonio Salmerón Cerdán. Ellos han hecho que el proceso de desarrollo de esta tesis doctoral se convirtiese en un viaje de descubrimiento entre amigos. Siempre me han ayudado a superar las dificultades con su sabiduría, su sentido del humor y su infinita paciencia. Quiero agradecer también a Serafín Moral Callejón por su inestimable ayuda y su apoyo durante todos estos años, y también a Andrés Masegosa por haber sido siempre un amigo y algunas veces (muchas) un fantástico profesor para mí.

Deseo extender mi agradecimiento a todos los miembros del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada por el excelente ambiente de trabajo que generan cada día, enfatizando mi agradecimiento a Silvia Acid y a Nicolás Pérez de la Blanca Capilla porque guardo dentro de mí el cariño y los buenos consejos que siempre me han dado. También me gustaría agradecer a mis colegas y amigos de CITIC, especialmente a Antonio Masegosa, Nacho del Amo, Victor Salazar, Gonzalo Milla Millán y Juan Ramón González, porque con ellos siempre había lugar para las risas durante las horas de trabajo.

Un agradecimiento muy especial también para los miembros de los proyectos “Elvira” y “Programo” por su apoyo y por el ejemplo que todos y cada uno de ellos me han dado, especialmente Julia Flores, quien es una excelente profesional y una mejor amiga. Gracias a todos los miembros del Departamento de Matemáticas de la Universidad de Almería por tratarme tan bien cada vez que he ido a visitarles.

Durante la fase de investigación de mi tesis he tenido la oportunidad de visitar dos centros de investigación de alto nivel, primero el Departamento de Ciencias de la Computación de la Universidad de Aalborg, bajo la supervisión del profesor Thomas Nielsen, y después la Escuela de Clayton de Tecnologías de la Información, en la Universidad de Monash, bajo la tutela de la profesora Ann Nicholson. Quisiera agradecerles a ambos el haberme tratado tan bien y el haberme enseñado tanto durante el tiempo que pasé con ellos. Deseo hacer extensivo mi agradecimiento al profesor Kevin Korb por su ayuda y su paciencia durante el

tiempo que pasé en Monash. No puedo olvidar a toda la gente que he conocido durante mis estancias, ya que todos ellos me han ayudado de una manera o de otra. Les estoy muy agradecida. Debo subrayar mi agradecimiento a Carmen Ruiz, Yanir Seroussi, Mona Abdulaziz, Ricardo Lage, Ricardo Coelho, Tales Tonini y Liz Brumer porque todos ellos me han enseñado mucho sobre la amistad, la vida y el cariño. Soy muy afortunada de tenerlos como amigos. Desde aquí le mando mi cariño también a la profesora Ingrid Zukerman y a todos los miembros del Coffee Club, echo mucho de menos vuestra compañía.

Finalmente, estoy muy agradecida a mis amigos por apoyarme siempre y por tener ese sentido del humor tan peculiar. Sin vosotros haciéndome que me ría de mí misma no podría seguir adelante. Todo mi cariño a la hermana que nunca tuve, Toñi García Plaza, no sé decir lo agradecida que estoy por tenerte en mi vida. Todo mi cariño también para Jose Manuel Llamas y su familia, porque estuvieron conmigo cuando más les necesité. Mi más sincero agradecimiento a Juanma Rodríguez Cantos por su cariño y por su apoyo incondicional. Gracias a mi propia familia, que siempre me hace sentir orgullosa. Gracias a mi padre, porque siento que siempre me protege desde donde sea que esté ahora. Y no tengo suficientes palabras para agradecer a mi madre, Matilde Ariza Cuberos, por dármelo todo todos los días.

Esta tesis doctoral ha sido financiada por el Ministerio de Ciencia y Competitividad y por el Fondo Europeo de Desarrollo Regional (FEDER) con el proyecto TIN2010-20900-c04-01 y por la beca FPI BES-2008-002049.

Acknowledgements

I would like to express my gratitude to my supervisors, Andrés Cano Utrera, Manuel Gómez Olmedo and Antonio Salmerón Cerdán. They made the whole process of developing this thesis a fulfilling journey among friends. They helped me to overcome all the problems with their wisdom, sense of humour and infinite patience. I want to thank as well Serafín Moral Callejón for his priceless help and encouragement during all these years, and to Andrés Masegosa for being always a friend and sometimes (many times) a great professor to me.

I want to extend my gratitude to all the members of the Department of Computer Science and Artificial Intelligence of the University of Granada for the excellent working environment that they provide every day, emphasizing my gratitude to Silvia Acid and Nicolás Pérez de la Blanca Capilla because I keep deep inside their affection and good advice. Also, I would like to thank my colleagues and friends from CITIC, specially Antonio Masegosa, Nacho del Amo, Victor Salazar, Gonzalo Milla Millán and Juan Ramón González, because they always found the way of sneaking a good laugh during working hours.

A very special thanks goes out to the members of the projects Elvira and Programo for their support and example, specially to Julia Flores for being such an excellent professional and a better friend. Thanks as well to the members of the Department of Mathematics of the University of Almería, that have treated me so well every time I've visited them.

During my research I had the opportunity to visit two top level research centres, first the Department of Computer Sciences of Aalborg University, under the supervision of professor Thomas Nielsen, and afterwards the Clayton School of Information Technology of Monash University, under the protection of professor Ann Nicholson. I would like to thank both of them for treating me so well and teaching me so much during the time I spent with them. Also I would like to express my gratitude to professor Kevin Korb for his help and patience during my stay at Monash. I cannot forget all the people that I met during my stays that one way or the other helped me, I am grateful to all of them. I must highlight my thanks to Carmen Ruiz, Yanir Seroussi, Mona Abdulaziz, Ricardo

Lage, Ricardo Coelho, Tales Tonini and Liz Brumer as all of them taught me a lot about friendship, life and love. I am lucky to count you guys as friends. A loving thought goes to professor Ingrid Zukerman and all the members of the Coffee Club, I long for your company.

Finally, I thank my friends so very much for being so supportive and for having a great sense of humour. Without you guys making me laugh at myself I couldnt have done it. All my love to the sister I never had, Toñi García Plaza, I cannot say how grateful I am for having you in my life. My love as well to Jose Manuel Llamas and his family, because they were there when I most needed them. My most sincere acknowledgement to Juanma Rodríguez Cantos for all his care and unconditional support. Thanks to my own loving family that makes me feel proud. Thanks to my father, because I feel he protects me from wherever he is now. And I have not enough words to thank my mother, Matilde Ariza Cuberos, for giving me everything every day.

This doctoral thesis has been jointly supported by the Spanish Ministry of Economy and Competitiveness and by the European Regional Development Fund (FEDER) under the project TIN2010-20900-C04-01 and by a FPI scholarship BES-2008-002049.

Contents

I	Introduction	1
1	Introducción (<i>in spanish</i>)	3
1.1	Motivación	3
1.2	Trabajo desarrollado	4
1.3	Líneas futuras	5
2	Introduction	9
2.1	Contributions	9
2.2	Overview	11
3	Background	13
3.1	Graphs	13
3.2	Causal networks and d-separation	16
3.2.1	Serial connections (head to tail)	17
3.2.2	Diverging connections (tail to tail)	17
3.2.3	Converging connections (head to head)	18
3.2.4	d-separation	18
3.3	Probabilistic graphical models	19
3.4	Bayesian networks	20
3.5	Inference in Bayesian networks	21
3.5.1	Probabilistic potentials	22
3.5.1.1	Operations with potentials	23
3.5.1.2	Normalisation of probabilistic potentials	27
3.5.1.3	Factorisation of probabilistic potentials	29
3.5.2	Exact inference algorithms	31

3.5.2.1	Variable elimination technique	31
3.5.2.2	Message passing technique	35
3.5.3	Approximate inference algorithms	36
3.5.3.1	Deterministic algorithms	36
3.5.3.2	Simulation algorithms: Monte Carlo	38
3.5.3.3	Comparing approximate inference algorithms: Fertig and Mann’s divergence	39
3.6	Learning Bayesian networks	40
3.6.1	Structural learning	41
3.6.1.1	Independence tests	41
3.6.1.2	Score and search	42
3.6.2	Parametric learning	44
3.7	Alternative representations of potentials	45
3.7.1	Probability Trees	46
3.7.1.1	Operations over Probability Trees	49
3.7.1.2	Inference with Probability Trees	58
3.7.1.3	Proportional subtrees	60
3.7.2	Binary Probability Trees	61
3.7.3	Canonical models	63
II Recursive Probability Trees		65
4	Recursive Probability Trees	67
4.1	Motivation	67
4.2	Recursive Probability Trees	68
4.3	Expressiveness of RPTs	77
4.3.1	Proportional values	77
4.3.2	Context-specific independencies	79
4.3.3	Multinets	80
4.3.4	Mixtures of conditional distributions	81
4.4	Operations with RPTs	82
4.4.1	Restriction	83

4.4.2	Combination	87
4.4.3	Marginalisation	90
4.4.4	Normalisation of RPTs	108
4.5	Experimental evaluation	111
4.5.1	Generation of a random RPT	112
4.5.2	Fixed RPT generation parameters	114
4.5.3	Varying RPT generation parameters	116
4.6	Conclusions and Future Work	119
III Inference		121
5 Inference with factorised representations		123
5.1	Motivation	124
5.2	Classical factorisation of probability trees	124
5.3	Fast factorisation of probability trees	129
5.3.1	Exact Decomposition	129
5.3.2	Approximate Decomposition	133
5.4	Inference with factorised representations	140
5.5	Experimental evaluation	142
5.6	Conclusions and Future Work	149
IV Learning		151
6 Learning Recursive Probability Trees		153
6.1	Learning RPTs from probabilistic potentials	153
6.1.1	Motivation	154
6.1.2	Building an RPT from a probabilistic potential	154
6.1.2.1	Description of the algorithm	155
6.1.2.2	Computing multiplicative factorisations	158
6.1.2.3	Detecting context-specific independencies	162
6.1.2.4	independentFactorisation algorithm	168
6.1.2.5	splitChainFactorisation algorithm	169

6.1.2.6	Setting the sensitivity of context-specific independencies detection	174
6.1.3	Examples	176
6.1.4	Problem Complexity and Properties of the Algorithm . . .	181
6.1.4.1	Problem Complexity	181
6.1.4.2	Properties of the algorithm	182
6.1.5	Learning from data	188
6.2	Score and Search approach	192
6.2.1	Motivation	193
6.2.2	Search and Score approach	194
6.2.3	Search strategy	197
6.3	Experimental evaluation	201
6.3.1	Learning RPTs from probabilistic potentials	201
6.3.2	Learning RPTs from data	214
6.3.3	Score and Search approach	218
6.4	Conclusions and Future Work	219
V Conclusions		221
7 Conclusions and Future Work		223
7.1	List of Publications	224
7.2	Future Work	226
7.2.1	Structure definition	226
7.2.2	Inference	226
7.2.3	Learning	227
7.2.4	Applications	227
Appendixes		228
References		241

List of Figures

3.1	Directed graph (i), undirected graph (ii) and partially directed graph (iii).	14
3.2	Disconnected graph.	15
3.3	Directed graph. The nodes in grey are the set of neighbours of X_3	15
3.4	Serial connection (head to tail).	17
3.5	Diverging connection (tail to tail).	17
3.6	Converging connection (head to head).	18
3.7	Directed acyclic graph where X_3 has received evidence.	19
3.8	Potential of three binary variables represented as a table.	23
3.9	Potential of three binary variables and the result of marginalising one of them out.	25
3.10	Combination of two potentials.	26
3.11	Potential of three binary variables and the result of restricting it to the configuration $X_1 = 0$	27
3.12	Potential of three binary variables.	28
3.13	Result of normalising the potential in Fig. 3.12.	28
3.14	Potential representing an unnormalised conditional probability distribution.	29
3.15	Result of normalising the potential in Fig. 3.14.	29
3.16	Decomposition of probabilistic potentials.	31
3.17	Bayesian network of three binary variables.	34
3.18	Intermediate step of the Variable Elimination process.	34
3.19	Result of the Variable Elimination algorithm.	34

LIST OF FIGURES

3.20 A Bayesian network (i) and a join tree (ii) associated with it. Probability propagation is carried out sending messages throughout the edges.	35
3.21 Potential of three binary variables with context-specific independencies.	47
3.22 Potential $\phi(X_1, X_2, X_3)$ (i), its representation as a probability tree (ii) and its approximation after pruning several branches (those rounded with a rectangular box) (iii).	48
3.23 A probability tree \mathcal{PT}_1 defining a potential ϕ_1 over the set of variables $\{X_1, X_2, X_3\}$ to be combined with a probability tree \mathcal{PT}_2 that represents a potential ϕ_2 over the variable X_2	51
3.24 Propagation of Alg. 2 to the children of X_1	51
3.25 Intermediate step of Alg. 2.	52
3.26 Probability tree resultant of applying Alg. 2 to combine the probability trees in Fig. 3.23.	52
3.27 A probability tree \mathcal{PT}_1 defining a potential ϕ_1 over the set of variables $\{X_1, X_2, X_3\}$ to be added with a probability tree \mathcal{PT}_2 that represents a potential ϕ_2 over the variable X_2	55
3.28 Propagation of Alg. 4 to the children of X_1	55
3.29 Intermediate step of Alg. 4.	56
3.30 Probability tree resultant of applying Alg. 4 to add the probability trees in Fig. 3.27.	56
3.31 A probability tree defining a potential ϕ over the set of variables $\{X_1, X_2, X_3\}$	57
3.32 Intermediate step of Alg. 3.	57
3.33 Probability tree resultant of applying Alg. 3 to marginalise out the variable X_3 from the probability tree in Fig. 3.31.	58
3.34 Probability tree with proportionalities and context-specific independencies.	60
3.35 Factorisation of the tree in Fig. 3.34 as a product of smaller probability trees.	60
3.36 Probability distribution $P(X_1 X_2)$ as a table, as a probability tree and as binary probability tree	62

LIST OF FIGURES

3.37 Noisy-OR gate for n causes.	63
4.1 An RPT with only <i>Split</i> and <i>Value</i> nodes.	69
4.2 Probability trees with proportional subtrees.	71
4.3 Factorisation of the probability tree in Fig. 4.2 using PTs.	71
4.4 Simple probability tree with proportionalities and its factorisation as an RPT.	72
4.5 Factorisation of the probability tree in Fig. 4.2 as an RPT.	73
4.6 Different RPTs for the probability distribution in a).	73
4.7 An RPT representing a full Bayesian network.	74
4.8 An RPT representing a potential of three variables.	76
4.9 Potential with proportional values.	78
4.10 RPT encoding the potential with proportional values described in Fig. 4.9.	78
4.11 Potential with context-specific independencies and proportionalities.	79
4.12 Bayesian multinet and RPT representation.	80
4.13 RPT representation of a mixture of conditional distributions using an auxiliary variable A	81
4.14 RPT encoding a Bayesian network	84
4.15 RPT restricted to a configuration, operation applied to the List node at the root.	86
4.16 RPT restricted to a configuration, operation applied to a Split node.	86
4.17 Result of restricting an RPT to a configuration using Alg. 8.	87
4.18 Combination of potentials with RPTs.	89
4.19 Combination of potentials with RPTs.	89
4.20 Multiplication of two RPTs (I).	91
4.21 Multiplication of two RPTs (II).	91
4.22 Multiplication of two RPTs (III).	92
4.23 Addition of two RPTs (I).	96
4.24 Addition of two RPTs (II).	96
4.25 Addition of two RPTs (III).	97
4.26 Addition of two RPTs (IV).	97
4.27 Addition of two RPTs (V).	97

LIST OF FIGURES

4.28 Marginalisation of RPTs (I).	100
4.29 Marginalisation of RPTs (II).	101
4.30 Marginalisation of RPTs (III).	101
4.31 Marginalisation of RPTs (IV).	101
4.32 Marginalisation of RPTs (V).	102
4.33 Marginalisation of RPTs (VI).	103
4.34 Marginalisation of RPTs (VII).	103
4.35 Marginalisation of RPTs (VIII).	104
4.36 Marginalisation of RPTs (IX)	104
4.37 Marginalisation of RPTs (X)	105
4.38 Marginalisation of RPTs (XI)	105
4.39 Marginalisation of RPTs (XII)	106
4.40 Marginalisation of RPTs (XIII)	106
4.41 Marginalisation of RPTs (XIV)	106
4.42 Step of Variable Elimination algorithm, X_1 is the variable to be removed.	107
4.43 Reordered RPT achieved in a step of Alg. 12	107
4.44 Result of applying Alg. 12 and Alg. 10.	108
4.45 Normalisation of RPTs, the RPT represents $\phi(X_1, X_2)$. Enclosed in the dashed line is the potential encapsulated in the Potential node.	110
4.46 Normalisation of RPTs, the RPT represents $P(X_1 X_2)$. Enclosed in the dashed line is the potential encapsulated in the Potential node.	111
4.47 Experiment 1: execution time of removing all the variables in the network over RPTs, PTs, unpruned PTs (PTWP) and CPTs. . .	115
4.48 Experiment 2: time average of the Variable Elimination algorithm over probability trees, unpruned PTs (PTWP), CPTs and RPTs, considering different factorisation levels.	117
4.49 Experiment 2: size average, in terms of number of stored probability values, of the biggest structure stored during the inference process over RPTs, CPTs, probability trees and unpruned probability trees (PTWP).	118

LIST OF FIGURES

4.50 Experiment 2: size average, in terms of number of stored nodes, of the biggest structure stored during the inference process over RPTs, CPTs, probability trees and unpruned probability trees (PTWP).	119
5.1 A probability tree with proportional sub-trees.	127
5.2 Decomposition of the tree in Fig. 5.1 using Alg.15.	128
5.3 Probability tree that cannot be factorised by variable X_2 with classical factorisation.	128
5.4 Decomposition of the tree in Fig. 5.3, which cannot be obtained using Alg.15.	129
5.5 Modified decomposition obtained from Fig. 5.4 after making the total mass be equal to the one in the original tree in Fig. 5.3. . . .	133
5.6 Decomposition of the tree in Fig. 5.7, using Alg. 16.	134
5.7 Result of multiplying the two trees in Fig. 5.6.	134
5.8 Decomposability in four real-world examples. The plot shows the distribution of the potentials that, after decomposing, attained different levels of error.	139
5.9 Potential $\phi(X_1, X_2, X_3)$ represented as a pruned probability tree.	140
5.10 Error vs. time for the Barley (top) and Munin (bottom,) networks. The solid line corresponds to method Factorise_VE and the dotted one to Prune_VE	144
5.11 Error vs. time for the Andes (top) and Water (bottom) networks. The solid line corresponds to method Factorise_VE and the dotted one to Prune_VE	145
5.12 Average potential size (in logarithmic scale) vs. time for the Barley (top) and Munin (bottom) networks. The solid line corresponds to method Factorise_VE and the dotted one to Prune_VE . . .	146
5.13 Maximum potential size (in logarithmic scale) vs. time for the Barley (top) and Munin (bottom) networks. The solid line corresponds to method Factorise_VE and the dotted one to Prune_VE . . .	147

LIST OF FIGURES

5.14	Average potential size vs. time for the Andes (top) and Water (bottom) networks. The solid line corresponds to method Factorise_VE and the dotted one to Prune_VE	148
5.15	Maximum potential size (in logarithmic scale) vs. time for the Andes (top) and Water (bottom) networks. The solid line corresponds to method Factorise_VE and the dotted one to Prune_VE	149
6.1	Alg. 18 workflow	156
6.2	Dependencies graph computed during the factorisation process.	159
6.3	RPT representing the situation described in Fig. 6.2.	160
6.4	Step of the algorithm when the removal of X_5 exceeds the threshold of information gain.	166
6.5	Step of the algorithm when we find a division in the graph and S_1 is empty.	167
6.6	Step of the algorithm when we find a division in the graph and we have variables in S_1	168
6.7	Creation of a Split chain.	170
6.8	Creation of a Split chain with a decomposition of the auxiliar graph.	171
6.9	Example of G_ϕ complete division. New recursive calls required for factors ϕ_1, ϕ_2, ϕ_3 and ϕ_4	174
6.10	Potential ϕ to be decomposed in Example 1.	177
6.11	Factors generating ϕ in Example 1.	177
6.12	Factors obtained from the decomposition in Example 1.	178
6.13	RPT learned in Example 1.	179
6.14	Potential ϕ considered in Example 2.	179
6.15	RPT learned in Example 2.	181
6.16	Tying parameters to factorise a CPT	196
6.17	KL divergence and size (logarithm of number of probability values stored) of the representation for models learned with different thresholds.	202
6.18	KL divergence and size (logarithm of number of probability values stored) of the representation for models learned with smaller thresholds.	204

LIST OF FIGURES

6.19	KL divergence and size (logarithm of number of probability values stored) of the representation learning from the same tree (slightly pruned) for different values of ϵ	206
6.20	KL divergence and size (logarithm of number of probability values stored) of the representation learning from the same tree (severely pruned) for different values of ϵ	207
6.21	KL divergence variation between the joint probability distribution of Cancer network and the model learned, and size of the learned model, for different values of the threshold.	210
6.22	RPT learned from the Cancer network	211
6.23	Largest and average structure sizes used during the inference process over different Bayesian networks.	212
6.24	Largest (left) and average (right) structure sizes used during the inference process over Barley Bayesian network.	213
6.25	Average Kullback-Leibler divergence between the original distribution and the learned, for different RPTs and for different database sizes.	215
6.26	Size of the learned RPTs for the database of 2000 instances.	216
6.27	Bayesian Information Criteria for the learned models.	217
6.28	Log likelihood of the learned models.	218
6.29	Size of the learned models, in terms of number of probability values used to represent them.	219
1	UML diagram for the RPTs classes	230
2	UML diagram for the VariableElimination classes	231
3	UML diagram for the ProbabilityTree classes	232
4	UML diagram for the search and score learning algorithm	233

List of Tables

4.1	Average and standard deviation of the size of the biggest structure stored during the inference	116
6.1	Average and standard deviation for the KL divergence values obtained between the posterior distribution obtained for every node of each network, using RPTs and using PTs.	214
1	Details of Hepatitis dataset.	234
2	Details of Glass Identification dataset.	235
3	Details of Ecoli dataset.	235
4	Details of Diabetes dataset.	236
5	Details of Breast Cancer dataset.	236
6	Details of Heart Disease dataset.	237
7	Details of Andes dataset.	237
8	Details of Water dataset.	238
9	Details of Barley dataset.	238
10	Details of Munin dataset.	238
11	Details of Cancer dataset.	239
12	Details of Alarm dataset.	239
13	Details of Pedigree dataset.	239
14	Details of Prostanet dataset.	240

Part I

Introduction

Chapter 1

Introducción (*in spanish*)

Este capítulo presenta una breve reflexión personal sobre la presente tesis, analizando el pasado, presente y futuro del trabajo realizado, en lengua española, a fin de cumplir con lo establecido en la normativa vigente de regulación de las enseñanzas oficiales de Doctorado y del título de Doctor por la Universidad de Granada aprobadas por Consejo de Gobierno de la Universidad de Granada en su sesión de 2 de Mayo del 2012.

1.1 Motivación

Los Modelos Gráficos Probabilísticos son una herramienta de modelado ampliamente extendida por su versatilidad y potencia para razonar y tomar decisiones en problemas que conllevan incertidumbre. Como herramienta de modelado permiten especificar relaciones complejas entre variables, así como incorporar la propia información probabilística en el modelo. Desde un punto de vista computacional, se pueden definir y aplicar sobre ellos algoritmos eficientes para la propia creación de las estructuras, o aprendizaje, y para la toma de decisiones o resolución de consultas, lo que denominamos inferencia. La efectividad y eficiencia con la que se realicen estas operaciones es motivo de estudio por parte de la comunidad científica, buscando siempre métodos más rápidos y que representen mejor y más compactamente las distribuciones de probabilidad asociadas a los problemas y algoritmos que devuelvan mejores soluciones y que lo hagan con un menor coste computacional.

La forma de representar la información probabilística es un tema clave a la hora de trabajar con estos modelos, ya que de la estructura de datos va a depender en gran medida la eficiencia de los algoritmos que trabajen con ella. Además, la estructura de datos utilizada debe ser lo suficientemente flexible para permitir el modelado de relaciones complejas entre variables y de la manera más compacta posible. Es común usar tablas de probabilidad para expresar esta información probabilística, pero al tener que especificar un valor de probabilidad para cada configuración de las variables involucradas, el tamaño de la representación crece exponencialmente con el número de variables. Una solución ampliamente extendida es el uso de árboles de probabilidad, estructuras en forma de árbol que se aprovechan de las independencias sensibles al contexto para compactar la representación de la información. En este contexto nació la idea que dio pie al desarrollo de este trabajo, al analizar las debilidades de esta última estructura.

Los Árboles de Probabilidad Recursivos (RPTs) generalizan a los árboles de probabilidad tradicionales, permitiendo expresar descomposiciones multiplicativas dentro de la propia estructura, lo que permite modelar relaciones más complejas de una manera más compacta. También da pie al desarrollo de algoritmos de inferencia que se aprovechen de estas factorizaciones para mejorar la eficiencia del proceso. Una cuestión fundamental al desarrollar una nueva estructura de datos es definir una metodología para poder construir la estructura a partir de una distribución de probabilidad, lo que implica buscar los patrones representables mediante Árboles de Probabilidad Recursivos dentro de la distribución. Esto implica diseñar métodos de aprendizaje que detecten independencias sensibles al contexto y posibles factorizaciones en las relaciones entre variables.

1.2 Trabajo desarrollado

Esta tesis se ha dividido en cinco partes. La primera aporta una introducción al problema junto con una revisión bibliográfica de los conceptos básicos necesarios para enmarcar el trabajo. El cuerpo de la tesis son las tres partes siguientes, la Parte II está dedicada a definir la estructura de los Árboles de Probabilidad Recursivos, analizando su expresividad y detallando la manera de trabajar con ellos mediante el análisis de la forma en que las operaciones básicas necesarias

para la inferencia se llevan a cabo mediante esta forma de representación. En el Capítulo 4 se muestran varios resultados que confirman los beneficios en términos de tiempo de computación y tamaño de almacenamiento al trabajar con RPTs, comparándolos con árboles de probabilidad tradicionales.

La Parte III está centrada en Inferencia. En esta parte se estudian los métodos de factorización clásicos de árboles de probabilidad, y se propone un nuevo método que resuelve algunos de los problemas presentes en la literatura. Los RPTs de manera natural permiten incorporar factorizaciones presentes en las distribuciones de probabilidad y trabajar con ellas de manera eficiente. De esta manera, el método de factorización propuesto se plantea como una herramienta para traducir potenciales en RPTs factorizados y así acelerar el proceso de inferencia. La Parte IV de esta tesis propone varias soluciones al problema de aprender un RPT a partir de una distribución de probabilidad, bien almacenada en otra estructura de datos o bien representada con una base de datos. Los resultados obtenidos con los métodos estudiados en los capítulos señalados avalan los beneficios de los RPTs cuando se trabaja con distribuciones de probabilidad que presentan patrones como independencias sensibles al contexto, valores proporcionales y otros tipos de factorizaciones. En el caso de que las distribuciones no puedan ser expresadas de manera exacta, se consiguen RPTs que contienen aproximaciones compactas de calidad.

La última parte, Parte V, aporta una reflexión general sobre el trabajo desarrollado, enumera las publicaciones obtenidas y discute las numerosas líneas de trabajo futuro.

1.3 Líneas futuras

En este trabajo se recogen algunas respuestas a tres preguntas fundamentales que se pueden definir como tres líneas de investigación que discurren en paralelo: definición de la estructura, inferencia y aprendizaje. Las tres corrientes presentan un amplio abanico de posibilidades para trabajos futuros. A continuación pasamos a detallar algunas de las ideas que se plantean.

Definición de la estructura

Con respecto a la propia estructura de datos, planeamos ampliar su expresividad mediante la incorporación de un nuevo tipo de nodo, el nodo Suma, que permita expresar descomposiciones aditivas de manera sencilla y eficiente. Además pretendemos modificar la estructura para trabajar en un espacio continuo. Hasta ahora se ha trabajado con variables discretas, por lo que la incorporación y el manejo de variables continuas es un tema que abre muchas posibilidades de investigación.

Inferencia

Se plantea seguir desarrollando la idea presentada en el Capítulo 5, para así diseñar algoritmos que incorporen la factorización rápida de potenciales en fases intermedias de la inferencia, donde los propios potenciales adquieran dimensiones demasiado grandes. También se plantea desarrollar nuevos métodos que trabajen directamente sobre RPTs, teniendo en cuenta la naturaleza factorizada de los potenciales representados. Otra línea de investigación relacionada es el desarrollo de algoritmos de inferencia que trabajen sobre RPTs que mezclen variables discretas y continuas.

Aprendizaje

Aparte de diseñar nuevos algoritmos que aprendan la estructura, se plantea refinar los algoritmos planteados para el aprendizaje a partir de datos presentados en el Capítulo 6, especialmente modificar el algoritmo basado en **score and search** añadiendo nuevos operadores, modificando los actuales, añadiendo otros tipos de factorizaciones y modificando la estrategia de búsqueda.

Aplicaciones

Por último, también consideramos como una línea de investigación futura la aplicación práctica de los RPTs. Al tratarse de una estructura de datos que permite representaciones muy compactas de las distribuciones de probabilidad, podrían

1.3 Líneas futuras

aplicarse en casos en los que el espacio de almacenamiento es un tema crítico, como pueden ser los dispositivos móviles.

Chapter 2

Introduction

Probabilistic Graphical Models (PGMs) enable efficient representation of joint distributions exploiting independencies among the variables. The independencies are encoded by means of the *d-separation* criterion [1]. Therefore only explicit dependencies will be represented and quantified. The values measuring the dependencies can be stored using several data structures, being *Conditional Probability Tables* (CPTs) the most common and straightforward. A CPT encoding a potential defined over a set of variables can be seen as a grid with a cell for each combination of values of these variables. This implies an exponential growth of memory space requirements depending on the number of variables. *Probability Trees* (PTs) [2; 3; 4] try to improve CPTs allowing context-specific independencies and usually obtaining memory space savings as a consequence. *Recursive Probability Trees* (RPTs) suppose another step in this direction and can be considered as a generalization of PTs. With this data structure it is possible to cover the modeling capabilities of PTs and to represent proportionalities, multinets [5] and mixtures of conditional distributions as well. Moreover, RPTs try to keep the information as factorised as possible. These features are used during inference in order to speed up the process.

2.1 Contributions

The main contribution of this thesis is the introduction of a new framework for representing and managing probabilistic potentials. Recursive Probability Trees

aim at reflecting some patterns often present within probabilistic potentials, like context specific independencies, proportional values and other types of multiplicative factorisations. This representation is also compact, keeping the potentials factorised while working with them. The full analysis of the structure and its capabilities is given in Chapter 4.

Two main contributions are given in Chapter 5. The first is a new algorithm for factorising probability distributions stored as probability trees (PTs). This method runs in linear time with respect to the number of variables in the tree, and is not dependent on the ordering of the variables, overcoming with this some of the issues of classical approaches to the problem. The described method can be easily incorporated when working with RPTs. The second contribution is a measure called the *factorisation degree*, that provides a heuristic to rank the variables in the domain of a probabilistic potential according to the accuracy of the decompositions that they induce. The computation of such measure is fast enough as to be included in probabilistic inference algorithms, where computing time is a crucial issue.

In Chapter 6 we propose three algorithms for learning RPTs. The first is an algorithm for learning Recursive Probability Trees from a probabilistic potential represented in any other data structure. The algorithm proposed follows a greedy methodology, and search for factorisations and context specific independencies within the original distribution to build a compact RPT. When an exact representation of the original distribution is too large, the algorithm is able to compute an accurate approximation.

The second learning algorithm is in fact a modification of the previous one, this time aiming at learning the structure from a database. The general workflow of the algorithm is preserved, but now the relation between the variables is measured in terms of a Bayesian score, the Bayesian Dirichlet equivalent metric (BDe) [6]. Also, the normalisation of the different factors does not rely on a normalisation constant, as the correspondent potentials are retrieved already normalised from the database.

Finally, the third learning framework is based on a Search and Score approach, where different local operators are defined, along with a specific search technique. As the search space of RPTs is enormous, the methodology explained aims at

reducing it by limiting the possible neighbours to be explored at each step of the search. This is done when defining the local operators, so by modifying them we can explore different parts of the search space, issue that belongs to the proposed future research lines.

2.2 Overview

This dissertation is arranged into five parts. Part I is a introductory section composed of three chapters. Chapter 1 summarises the main conclusions achieved with this work, and it has been written in Spanish to fulfil the requirements given by the University of Granada related to the Doctoral theses that aim to obtain the International mention. Chapter 2 provides an introduction to the topic and explains the main contributions of the thesis. Chapter 3 provides the necessary background to enshrine this work, along with the definition of the notation used throughout the dissertation.

The secont part, Part II, only contains one chapter, Chapter 4. This chapter is devoted to explain and analyse the structure and capabilities of Recursive Probability Trees. Part III focuses on Inference, where Chapter 5 analyses the factorisation of probabilistic potentials and its possible applications on RPTs. Part IV is devoted to Learning. Chapter 6 addresses the problem of transforming probabilistic potentials into RPTs, as well as dealing with the problem of learning RPTs from data using two different approaches.

Finally, Part V contains one last chapter, Chapter 7. This chapter provides a discussion of the main conclusions of the dissertation and states future research lines. Also, a list of publications supporting the contributions of this thesis is provided.

Chapter 3

Background

3.1 Graphs

A graph \mathcal{G} is a data structure composed of a set of nodes $X = \{X_1, \dots, X_n\}$ and a set of edges $L = \{l_1, \dots, l_l\}$. Each edge connects a different pair of nodes from X , and the specification of the edge will define its nature, and therefore, the nature of the graph. An edge that connects two nodes X_i, X_j , can be directed $X_i \rightarrow X_j$ or $X_i \leftarrow X_j$ or undirected $X_i - X_j$. If all the edges in the graph are directed, then the graph is directed. If all the edges are undirected, the graph is called undirected as well. If within a graph there are directed and undirected edges, we will say it is a partially directed graph. In some cases it is useful to disregard the direction of the edges in a directed or partially directed graph, transforming it into an undirected graph. In Fig. 3.1 (i) there is an example of a directed graph, undirected graph (ii) and partially directed graph (iii). The definition of a graph can be more general, allowing several edges to connect the same pair of nodes or even allowing edges from one node to itself. However, for the scope of this dissertation those cases will never be considered.

Whenever we have an edge (directed or undirected) that connects X_i and X_j , we will say that X_i is a neighbour of X_j and vice versa. In Fig. 3.3 we see the set of neighbours of X_3 coloured in grey. The set of all the neighbours of a node is called its *neighbourhood*. It is possible to travel from one node to another following the edges that connect neighbours, if those edges exist. We define a *path* between two nodes X_i and X_k as the set of nodes X_i, \dots, X_k where for each pair X_i, X_{i+1} ,

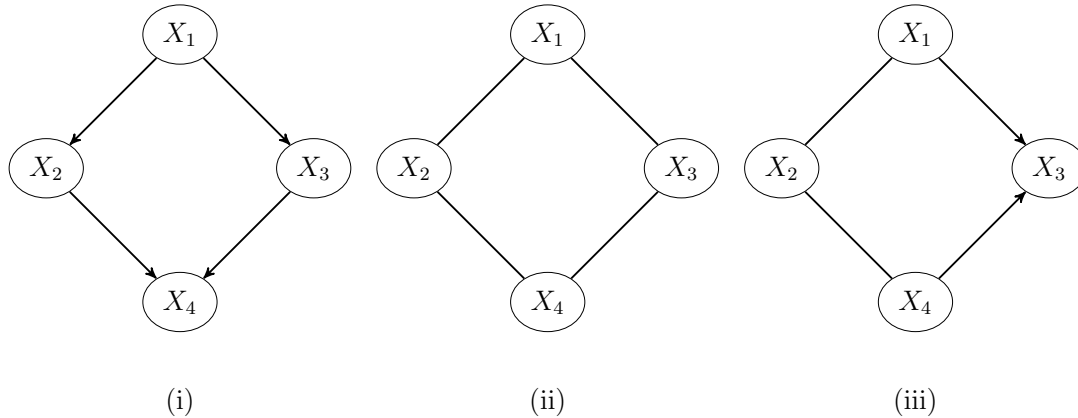


Figure 3.1: Directed graph (i), undirected graph (ii) and partially directed graph (iii).

exists an edge such as $X_i \rightarrow X_{i+1}$ or $X_i - X_{i+1}$. We will say that X_i, \dots, X_k form a *trail* if each pair X_i, X_{i+1} are connected with an edge, independently of its direction. We will say a graph is connected when for each X_i, X_j , exists a trail between them. Otherwise, the graph will be disconnected, and the parts of the graph that remain connected will be called connected components of the graph.

Example 1 Consider the graph in Fig. 3.2 whose set of nodes is $\{X_1, X_2, X_3, X_4, X_5, X_6, X_7\}$. This graph is disconnected, as there are no trails that connect the subsets $\{X_1, X_2, X_3, X_4, X_5\}$ and $\{X_6, X_7\}$. These two sets with their corresponding edges form the two connected components of this graph.

Example 2 The directed graph in Fig. 3.3 is a connected graph, as every node is connected through an edge to the rest of the network. From X_1 we can follow a path to X_5 or X_6 , going through X_3 . The set of nodes $\{X_1, X_3, X_6, X_4\}$ form a trail, as the edge between X_6 and X_4 goes in the reverse direction.

A cycle in a graph is a directed path $\{X_1, \dots, X_k\}$ where $X_1 = X_k$. We will define an acyclic graph as a graph that contains no cycles. If this graph is directed, then we will call it directed acyclic graph (DAG). This will be a key concept along this dissertation as a DAG is the underlying graphical structure of a Bayesian network.

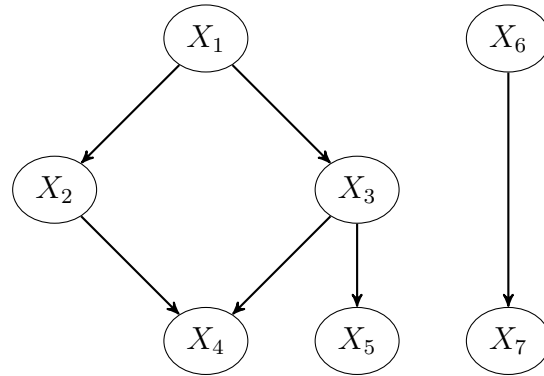
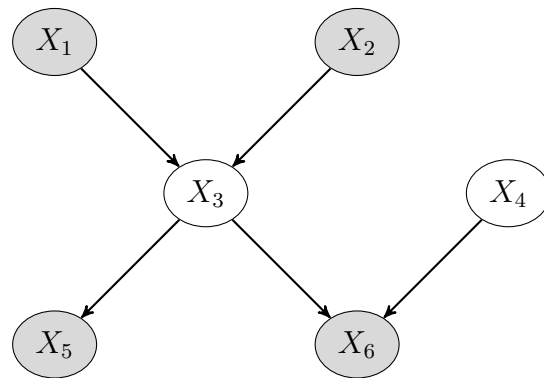


Figure 3.2: Disconnected graph.

Figure 3.3: Directed graph. The nodes in grey are the set of neighbours of X_3 .

We will also define a *tree* as a directed graph that has only one path between each pair of nodes. From this definition we see that a tree is a connected graph, but if any of its edges is removed, then the graph becomes disconnected. Again, a tree does not contain cycles (it is a DAG), but if we add an edge to it, we automatically introduce a cycle in it.

Example 3 *The example in Fig. 3.3 corresponds to a tree, and we see how the removal of any edge disconnects the graph, and the addition of an edge anywhere in the graph creates a cycle. Every node in a tree can be seen as the root node of the subtree rooted at that node.*

3.2 Causal networks and d-separation

A causal network is a DAG that represents the relations of dependency between elements in a given model. The nodes in the DAG represent the elements -from now on denoted as *variables*¹- of the model, and the edges represent the causal relations between them. Given this semantics, we will refer to the nodes in a causal network as variables, and we will rename the edges as arcs. From now on, when talking about the relations between a pair of variables X_i, X_j where there is an arc $X_i \rightarrow X_j$, we will say that X_i is a *parent* of X_j , and therefore X_j is a *child* of X_i . We will say X_i is an *ancestor* of X_k if there is a path from X_i to X_k , and so X_k is called a *descendant* of X_i . We will denote as π_{X_i} to the set of parents of variable X_i in the model. For example, in Fig. 3.3, $\pi_{X_3} = \{X_1, X_2\}$.

In a causal network, every variable represents a set of possible states within the model. At one time point, every variable can be in only one state, and changes in those states can lead to changes in the remaining variables. The knowledge of the certainty of a variable is called *evidence*. Evidence, \mathbf{e} , is an assignment of values to a subset \mathbf{E} of the domain's variables. The evidence can be strong, if it establishes the exact state of the variable that it refers, or soft, in other case. When the state of a variable is known, that is to say we have strong evidence about its state, then we call the variable *instantiated*. The way of how a change of certainty in one variable may change the certainty for other variables is defined by the arcs, following a set of rules that we describe in this section.

¹A *variable* symbolizes a measurable attribute that usually corresponds to a defined entity in the represented model. Variables can be discrete (taking values from a countable set of *states*) or continuous. For the scope of this dissertation, when we refer to variables we will always refer to discrete variables with a finite number of possible states, unless otherwise stated. We will denote a variable as a capital letter, usually X , and we will use x when referring to a particular state of X . The whole set of states for a given variable X will be denoted as Ω_X , and we will note a set of variables in boldface, for example \mathbf{X} (therefore, $\Omega_{\mathbf{X}}$ will be the set of possible combinations of states of the variables in \mathbf{X}). If the set is indexed, we will include a subindex to the boldface label denominating the set of indexes, for example $\mathbf{X}_{\mathbf{I}} = \{X_1, \dots, X_I\}$

3.2.1 Serial connections (head to tail)

A serial connection is represented in Fig. 3.4, where X_1 influences X_2 , and X_2 has an effect on X_3 . If we do not know anything about X_2 , a knowledge of X_1 will influence our knowledge of X_3 through X_2 , and vice versa.

If we have strong evidence about X_2 , the channel between X_1 and X_3 becomes blocked, and they become independent, given X_2 . In this example, we say that X_1 and X_3 are d-separated given X_2 , as there is only one path connecting them. This concept of conditional independence can be generalised as follows:

Given three disjoint sets of variables \mathbf{X} , \mathbf{Y} and \mathbf{Z} , we say that \mathbf{X} and \mathbf{Y} are conditionally independent given \mathbf{Z} (defined as $I(\mathbf{X}, \mathbf{Y}|\mathbf{Z})$) if $P(x|z) = P(x|y, z)$ for each possible value of $\mathbf{x}, \mathbf{y}, \mathbf{z}$ of $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$, such as $P(x, y) > 0$. Otherwise, we say that \mathbf{X} and \mathbf{Y} are conditionally dependent given \mathbf{Z} , and we write it as $D(\mathbf{X}, \mathbf{Y}|\mathbf{Z})$.



Figure 3.4: Serial connection (head to tail).

3.2.2 Diverging connections (tail to tail)

In a diverging connection such as the one presented in Fig. 3.5, the flow of information can travel among the children of X_1 unless X_1 is instantiated. That is, all the children are d-separated given the parent, as there are no other path between them. We say in this situation that the parent is the common cause of all the children.

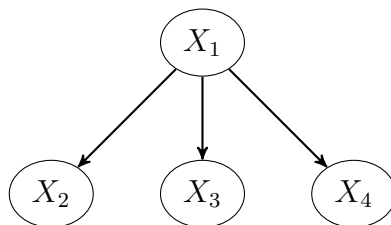


Figure 3.5: Diverging connection (tail to tail).

3.2.3 Converging connections (head to head)

This situation is illustrated in Fig. 3.6. In this case, every parent affects the child X_4 . If we do not have any evidence over the model, all the parents are independent: evidence about one of them cannot influence the state of the others through their common child. However, if we have some evidence about the child, then the information about one of the parents can give some information about the rest. In other words, the parents become conditionally dependent given the child. This information about the child can be direct evidence about it, or evidence about any child of it. So if neither the child nor any of its descendants have received evidence, the parents are d-separated. In this case, we say that the child is the common effect of all the parents.

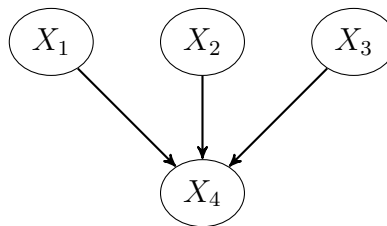


Figure 3.6: Converging connection (head to head).

3.2.4 d-separation

The preceding discussion explains all the possible ways of transmitting the evidence among the variables in a causal model. In general, it is possible to determine if two variables are dependent given some evidence. In summary, the rules of d-separation are the following:

Two variables X_i and X_j in a causal model are d-separated if for each path between X_i and X_j exists a variable X_k such as:

- the connection is serial or diverging and X_k is instantiated, or,
- the connection is converging and neither X_k nor any of its descendant have received evidence.

If X_i and X_j are not d-separated, then we say they are d-connected.

Example 4 For example, consider the DAG in Fig. 3.7. The variable X_3 is instantiated, and to denote this situation, the node is coloured in grey. In this context, the flow of information between X_1 and X_6 is closed, as there is only a serial connection between them, and X_3 is instantiated. The same situation happens between X_5 and X_6 , there is a diverging connection and the flow of information is stopped by the instantiation of X_3 . And finally, if we consider the two diverging connections present in this graph, we see that the flow of information is open between X_1 and X_2 due to the instantiation of X_3 , but it is closed between X_3 and X_4 , as they remain independent as long as X_6 does not receive evidence.

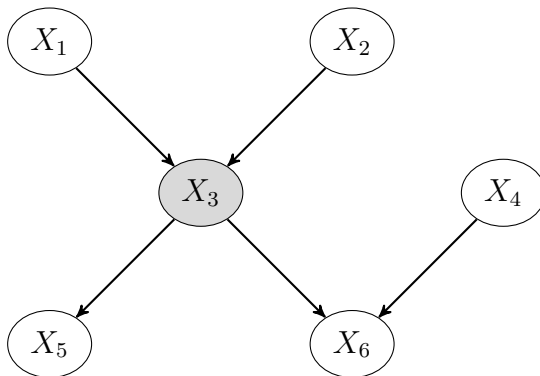


Figure 3.7: Directed acyclic graph where X_3 has received evidence.

3.3 Probabilistic graphical models

A probabilistic graphical model (PGM) is a group of three elements $\{\mathbf{V}, P, \mathcal{G}\}$ where \mathbf{V} is a set of random variables with a joint probability distribution $P(\mathbf{V})$ and \mathcal{G} is a graph that represents the interactions between the variables in the model.

The probability distribution and the graph are related in the sense that all the independencies between variables expressed in \mathcal{G} due to the d-separation criterion are reflected in the probability distribution P . However, it is possible that some independencies present in P are not present in \mathcal{G} . Formally, we will say that $I(X, Y|Z)_{\mathcal{G}} \rightarrow I(X, Y|Z)_P$.

The procedure of building a probabilistic graphical model for a set of variables can be simplified by considering a factorisation of the underlying probability distribution as a product of conditional probability distributions. To do so, it is necessary to know the dependency relations between the variables in the model. This means that the analysis of a probabilistic graphical model has two parts:

- Qualitative: the cause-effect relationships between the variables are represented using a graphical model.
- Quantitative: the dependencies are quantified through probability distributions.

There are several types of probabilistic graphical models. Without being exhaustive and attending to the types of nodes in the model, we can have:

- PGMs that only contains nodes that represent variables:
 - If \mathcal{G} is an undirected graph: Markov network.
 - If \mathcal{G} is a DAG: Bayesian network.
 - if \mathcal{G} is a partially directed graph and has certain restrictions: chain graphs.
- PGMs that contain variables and other types of nodes, as decision and utility nodes:
 - Markov decision models.
 - Influence diagrams.

3.4 Bayesian networks

A Bayesian network is a set $BN = \{\mathbf{V}, \mathcal{G}, \mathbf{P}\}$ where \mathcal{G} is a DAG where every node is associated to one of the variables of the model, $X_i \in \mathbf{V}, i = 1, \dots, n$ and $\mathbf{P} = \{P(X_1|\pi_{X_1}), \dots, P(X_n|\pi_{X_n})\}$ is a set of conditioned probability distributions, one for each variable given its set of parents. The set \mathbf{P} defines an associated

joint probability distribution through the factorisation (known as the **chain rule for Bayesian networks**):

$$P(X) = \prod_{i=1}^n P(X_i | \pi_{X_i}). \quad (3.1)$$

The advantages of explicitly representing a probability distribution over a set of variables using a Bayesian network are many: simplify conditionalisation, ability to plan decisions under uncertainty and capacity to explain the outcome of stochastic processes.

- Building the network becomes easier. Either if we are filling the probability values using a human expert's knowledge or if we are using some kind of automated learning algorithm, it is much easier to fill a set of smaller conditional probability distributions $P(X_i | \pi_{X_i})$ than considering all the possible cases of the joint distribution $P(X_1, \dots, X_n)$ one by one.
- The size of the model decreases. If we consider the joint probability distribution $P(X_1, \dots, X_n)$, we can see how the number of parameters grows exponentially on the number of nodes in the network. However, if we use a factorised representation, we see how for each conditional probability distribution $P(X_i | \pi_{X_i})$ the number of parameters only grows exponentially on the number of parents.
- Inference can be carried out with local operations. Algorithms can compute all the marginals and conditional probability distributions of $P(X)$ without computing the whole joint probability distribution.

3.5 Inference in Bayesian networks

One of the main tasks when working with probabilistic graphical models is **belief updating** or **probabilistic inference**, that consists of the computation of the posterior probability distribution for a set of query nodes given some evidence over the rest of the network. Performing inference in Bayesian networks is a very flexible process, as evidence can be entered about any node while beliefs in any

other nodes are updated. If \mathbf{E} is the set of variables instantiated by the evidence \mathbf{e} , the simplest question would be to know the probability of this evidence:

$$P(e) = \sum_{X_i \notin \mathbf{E}} P(X_1, \dots, X_n, \mathbf{e}). \quad (3.2)$$

However, in the context of Bayesian networks, the most common query will be to compute the posterior probability of a variable (or a set of variables, usually the *unobserved*, those variables not instantiated by the evidence) called *variable(s) of interest*, given the evidence:

$$P(\mathbf{X}|\mathbf{e}) = \frac{P(\mathbf{X}, \mathbf{e})}{P(\mathbf{e})}. \quad (3.3)$$

Inference in Bayesian networks has been a productive field of study for more than 20 years, leading to the development of many different algorithms. We distinguish between **exact algorithms** [7; 8; 9; 10; 11; 12; 13] -those algorithms that compute the probabilities of the nodes without any other error than the introduced by the computer- and **approximate algorithms** [2; 14; 15; 16; 17] -algorithms that use different techniques to obtain approximate values of the probabilities. In general, theoretically both types of algorithms are computationally complex - NP-hard¹ [18; 19]- In practice, the speed of the inference is determined by many factors such as the topology of the network (how dense it is, among others) and, as we will develop thorough this dissertation, the representation in the computer of the probability values contained in the model.

3.5.1 Probabilistic potentials

We need to introduce a new concept that will be used from now on thorough the whole dissertation. It usually happens, when working with probability distributions, that the results of local operations are not normalised. These unnormalised

¹NP stands for “nondeterministic polynomial time”, a term going back to the roots of complexity theory. A problem is said to be NP if we can find a nondeterministic Turing machine that can solve the problem in a polynomial number of nondeterministic moves. That is to say, its solution comes from a finite set of possibilities and it takes polynomial time to verify the correctness of a candidate solution. NP-hard is a class of problems that are, informally, at least as hard as the hardest problems in NP.

probability distributions must be stored and handled during the inference. Informally, a *potential* [20] is a probability distribution that is not necessarily normalised.

Formally, a potential ϕ is a real-valued function over a *domain* of finite variables \mathbf{X} (denoted as $\text{dom}(\phi)$):

$$\phi : \Omega_{\mathbf{X}} \rightarrow \mathbb{R}_0^+ \tag{3.4}$$

where \mathbb{R}_0^+ is the set of non-negative real numbers.

The *size* of a potential ϕ is the number of values needed to fully represent it. If ϕ is defined over $\Omega_{\mathbf{X}}$, its size will be denoted as $|\Omega_{\mathbf{X}}|$.

Example 5 Consider the potential ϕ of three binary variables $\mathbf{X} = \{X_1, X_2, X_3\}$ represented in Fig. 3.8. In this case, $|\Omega_{\mathbf{X}}| = 8$, as we need 8 values to fully represent the potential.

X_1	X_2	X_3	$\phi(X_1, X_2, X_3)$
0	0	0	0.2
0	0	1	0.5
0	1	0	0.7
0	1	1	0.7
1	0	0	0.3
1	0	1	0.5
1	1	0	0.3
1	1	1	0.3

Figure 3.8: Potential of three binary variables represented as a table.

3.5.1.1 Operations with potentials

There are three basic operations over potentials that are needed for performing inference: the *marginalisation* of a potential over a set of variables, the *combination* or multiplication of potentials and the *restriction* of a potential to a given *configuration* of its variables. Given a set of variables $\mathbf{X}_{\mathbf{J}}$, we will denominate a *configuration* of $\mathbf{X}_{\mathbf{J}}$ (denoted as $\mathbf{x}_{\mathbf{J}}$) to an instantiation of all or some of the variables in $\mathbf{X}_{\mathbf{J}}$.

Example 6 For example, in Fig. 3.8, all the possible configurations for the set of binary variables $\{X_1, X_2, X_3\}$ are disclosed (the 8 rows of the table for the first 3 columns).

This representation where the full set of configurations and their corresponding probability value are displayed in a tabular form is called *probability table*, and it is the traditional and more straightforward way of representing a probability potential. If the potential is a conditional probability distribution, the representation is called *conditional probability table*.

Marginalisation

The key operation that we are performing when computing the probability of a subset of variables is that of marginalising out variables from a distribution. In general, given a potential ϕ defined over $\Omega_{\mathbf{X}_I}$ and a subset $\mathbf{J} \subset \mathbf{I}$, with $\mathbf{K} = \mathbf{I} - \mathbf{J}$, we define the *marginalisation* of ϕ over the variables in \mathbf{X}_J (or the *elimination* of the variables with indexes in \mathbf{K}) as a new potential $\phi^{\downarrow \mathbf{X}_J}$ defined over the set of variables \mathbf{X}_J as:

$$\phi^{\downarrow \mathbf{X}_J}(\mathbf{X}_J) = \sum_{\mathbf{X}_K} \phi(\mathbf{X}_J, \mathbf{X}_K). \quad (3.5)$$

Informally, to *marginalise out* a variable from a potential is to add the values corresponding to the configurations in the potential that only differ in the state of that variable, for each possible state it can reach. The result is a new potential with a reduced dimension, as the variable marginalised out will not be part of it.

Example 7 An example of this is shown in Fig. 3.10, where we consider a potential of three binary variables $\phi(X_1, X_2, X_3)$, and the potential resulting after marginalising out X_1 . The new potential is defined over the set $\{X_2, X_3\}$ and therefore its size is smaller than the original one, being necessary only 4 values to fully define it.

A modification to this operation is the *marginalisation by maximum*, that instead of adding the values for the configurations, it chooses the maximum probability value for each configuration and for each state of the variable. This is


X_1	X_2	X_3	$\phi(X_1, X_2, X_3)$		X_2	X_3	$\sum_{\mathbf{x}_1} \phi(X_1, X_2, X_3)$
0	0	0	0.2		0	0	$0.2 + 0.3 = 0.5$
0	0	1	0.5		0	1	$0.5 + 0.5 = 1$
0	1	0	0.7		1	0	$0.7 + 0.3 = 1$
0	1	1	0.7		1	1	$0.7 + 0.3 = 1$
1	0	0	0.3				
1	0	1	0.5				
1	1	0	0.3				
1	1	1	0.3				

Figure 3.9: Potential of three binary variables and the result of marginalising one of them out.

useful when we are interested in obtaining the most probable explanation for a given evidence of the model, whilst the marginalisation by addition is used for probability propagation.

Combination

In general, if we have a set of r potentials ϕ_1, \dots, ϕ_r , defined over the sets $\Omega_{\mathbf{X}_{\mathbf{I}_1}}, \dots, \Omega_{\mathbf{X}_{\mathbf{I}_r}}$ respectively, the combination of all of them will be a new potential defined over the set of variables with indexes in $\mathbf{I} = \bigcup_{i=1}^r \mathbf{I}_i$ given by the expression:

$$\phi(\mathbf{X}_{\mathbf{I}}) = \prod_{i=1}^r \phi_i(\mathbf{X}_{\mathbf{I}}^{\downarrow \mathbf{I}_i}). \quad (3.6)$$

The result of this operation is a new potential defined over the set of possible states of the union of the variables of each individual multiplied potential. The dynamics of the operation is simple: for each configuration of the new potential, multiply the values of every individual potential that are consistent with the configuration, and repeat for every configuration.

Example 8 *This is illustrated in Fig. 3.10, where a potential of only one variable, X_1 , is combined with a potential of two variables X_2, X_3 resulting a potential*

of three variables X_1, X_2, X_3 .

					X_1	X_2	X_3	$\phi(X_1, X_2, X_3)$
					0	0	0	$0.2 \cdot 0.2 = 0.04$
			X_2	X_3	$\phi(X_2, X_3)$	0	0	$0.2 \cdot 0.5 = 0.1$
X_1	$\phi(X_1)$	0	0	0.2	0	1	0	$0.2 \cdot 0.7 = 0.14$
0	0.2	0	1	0.5	0	1	1	$0.2 \cdot 0.3 = 0.6$
1	0.7	1	0	0.7	1	0	0	$0.7 \cdot 0.2 = 0.14$
		1	1	0.3	1	0	1	$0.7 \cdot 0.5 = 0.35$
					1	1	0	$0.7 \cdot 0.7 = 0.49$
					1	1	1	$0.7 \cdot 0.3 = 0.21$


Figure 3.10: Combination of two potentials.

Restriction

We define the *restriction* of a potential ϕ , defined over $\Omega_{\mathbf{X}_I}$, to a configuration \mathbf{x}_J , where $J \subseteq I$, as a new potential $\phi^{R(\mathbf{x}_J)}$ obtained by getting the values from ϕ that are consistent with \mathbf{x}_J .

Example 9 This operation is illustrated in the example in Fig. 3.11, where a potential of three binary variables $\{X_1, X_2, X_3\}$ is restricted to the configuration $\{X_1 = 0\}$, where X_1 takes its first value. The result is a potential defined over the remaining variables, $\{X_2, X_3\}$, nevertheless, implicitly it is known that those values correspond to the context of the restrictive configuration.

X_1	X_2	X_3	$\phi(X_1, X_2, X_3)$
0	0	0	0.2
0	0	1	0.5
0	1	0	0.7
0	1	1	0.7
1	0	0	0.3
1	0	1	0.5
1	1	0	0.3
1	1	1	0.3



X_2	X_3	$\phi^{R(X_1=0)}(X_2, X_3)$
0	0	0.2
0	1	0.5
1	0	0.7
1	1	0.7

Figure 3.11: Potential of three binary variables and the result of restricting it to the configuration $X_1 = 0$.

3.5.1.2 Normalisation of probabilistic potentials

A probabilistic potential can be turned into a probability distribution by making its values add up to one. This process is called *normalisation*, and it is an essential operation when working with probabilistic graphical models. The basic idea is to divide every value in the potential by the total sum of all its values.

Definition 1 We define the sum of a potential ϕ , denoted as sum_ϕ , as the addition of all the values in the potential.

$$sum_\phi = \sum_{\forall \mathbf{x} \in \mathbf{X}} \phi(\mathbf{x}). \tag{3.7}$$

Definition 2 Therefore, the normalisation of a probabilistic potential ϕ , denoted as $normalise(\phi)$, can be defined as the division of each and every value in the potential by sum_ϕ .

$$normalise(\phi) = \frac{\phi(\mathbf{x})}{sum_\phi}, \forall \mathbf{x} \in \mathbf{X}. \tag{3.8}$$

Example 10 Consider the potential ϕ in Fig. 3.12. We compute sum_ϕ by applying Eq.(3.7), which gives us the value 3.5. According to Eq.(3.8), the next step

3.5 Inference in Bayesian networks

of the algorithm is to go through all the configurations in ϕ , dividing every value by sum_ϕ , which gives us the potential ϕ_N , represented in Fig. 3.13.

X_1	X_2	X_3	$\phi(X_1, X_2, X_3)$
0	0	0	0.175
0	0	1	0.525
0	1	0	0.28
0	1	1	0.07
1	0	0	0.35
1	0	1	0.7
1	1	0	1.05
1	1	1	0.35

Figure 3.12: Potential of three binary variables.

X_1	X_2	X_3	$\phi(X_1, X_2, X_3)$
0	0	0	$0.175/3.5 = 0.05$
0	0	1	$0.525/3.5 = 0.15$
0	1	0	$0.28/3.5 = 0.08$
0	1	1	$0.07/3.5 = 0.02$
1	0	0	$0.35/3.5 = 0.1$
1	0	1	$0.7/3.5 = 0.2$
1	1	0	$1.05/3.5 = 0.3$
1	1	1	$0.35/3.5 = 0.1$

Figure 3.13: Result of normalising the potential in Fig. 3.12.

However, it may be the case that a potential ϕ represents a conditional probability distribution of the variables in \mathbf{X}_c given the set of parents \mathbf{X}_p . In this case, the potential must add up to one for each configuration of the parents. To normalise ϕ we have to independently normalise the potentials (using Eq. (3.8))

resultant of restricting ϕ to each configuration of the set of parents \mathbf{X}_p :

$$c_normalise(\phi) = normalise(\phi(\mathbf{X}_c)^{R(\mathbf{x}_p)}), \forall \mathbf{x}_p \in \mathbf{X}_p. \quad (3.9)$$

Example 11 Consider the potential of three binary variables shown in Fig. 3.14. Note that we use here a different representation for the conditional probability table, as this display makes it easier grouping together the values of X_1 given the different configurations of the set of parents $\{X_2, X_3\}$. The potential in the figure is an unnormalised representation of the conditional probability distribution $P(X_1|X_2, X_3)$. To normalise it, we use Eq. (3.9), where we basically normalise each column as if they were independent potentials of only one variable, in this example, X_1 . Fig. 3.15 shows the normalised potential, where every column adds up to 1.

X_1	X_2, X_3			
	00	01	10	11
0	0.09	0.02	0.66	0.14
1	0.36	0.18	0.44	0.06

Figure 3.14: Potential representing an unnormalised conditional probability distribution.

X_1	X_2, X_3			
	00	01	10	11
0	0.2	0.1	0.6	0.7
1	0.8	0.9	0.4	0.3

Figure 3.15: Result of normalising the potential in Fig. 3.14.

3.5.1.3 Factorisation of probabilistic potentials

When working with PGMs we seek an accurate representation of the probability distributions involved in the models. Besides accuracy, we will be interested in

compacting the information both to save storage space and to try to benefit the inference.

The division of potentials can lead to storage savings, as we aim at representing potentials over several variables as a multiplicative factorisation of smaller potentials. The inference in this case also may gain efficiency, as we can design algorithms that take into account these factorisations to minimize the complexity of the operations.

As discussed above, a probabilistic potential represented as a table specifies the probability value associated to every possible configuration of the variables in the potential, which means that a probability table that stores a potential ϕ defined over a set of variables $\{X_1, \dots, X_n\}$ will need $|\Omega_{X_1}| \cdot \dots \cdot |\Omega_{X_n}|$ values to fully represent it. If we represent the potential as a combination of two smaller potentials, the total number of values that we need for the representation would be the addition of the sizes of the factors. The size of the factorisation can be smaller than the size of the original potential, for example if we seek *disjoint factorisations*, that would be those decompositions¹ where the subsets of variables of the factors do not share any variable among them.

Definition 3 A potential ϕ defined for a set of variables \mathbf{X} is said to be factorisable with respect to ϕ_1 and ϕ_2 if:

$$\phi(\mathbf{x}) = \phi_1(\mathbf{x}^{\mathbf{I}}) \cdot \phi_2(\mathbf{x}^{\mathbf{J}}), \forall \mathbf{x} \in \mathbf{X} \text{ such that } \mathbf{X} = \mathbf{I} \cup \mathbf{J}.$$

Additionally, if we observe that $\mathbf{I} \cap \mathbf{J} = \emptyset$, we will say that ϕ is decomposable with respect to ϕ_1 and ϕ_2 .

Example 12 For example, consider the potential ϕ_1 represented in Fig. 3.16. This potential is defined over three binary variables $\mathbf{X} = \{X_1, X_2, X_3\}$ and so, the total number of values needed to be specified in the probability table are $2^3 = 8$ values. We can decompose ϕ_1 into the multiplication of two smaller potentials, ϕ_2 and ϕ_3 , as specified in Fig. 3.16. These two potentials are defined over two disjoint subsets of \mathbf{X} which makes the decomposition a smaller representation of the original potential. Adding the number of parameters of each factor, the total number of values needed to represent ϕ_1 is 6.

¹We will specifically denominate a factorisation as a *decomposition* if we know that the domains of the factors are disjoint. Otherwise, we will use the general term.

X_1	X_2	X_3	$\phi_1(X_1, X_2, X_3)$		X_2	X_3	$\phi_3(X_2, X_3)$
0	0	0	0.02		0	0	0.1
0	0	1	0.04		0	1	0.2
0	1	0	0.08		1	0	0.4
0	1	1	0.06	=	0	0.2	·
1	0	0	0.08		1	0.8	
1	0	1	0.16			1	0.3
1	1	0	0.32				
1	1	1	0.24				

Figure 3.16: Decomposition of probabilistic potentials.

3.5.2 Exact inference algorithms

Performing exact inference over a Bayesian network consists of computing the marginal distributions of each variable modifying the information of its neighbours using exact computation. When the information in a node is modified, this change affects its neighbours, and so on, making the message-passing process an *NP-complete*¹ problem [18].

There are some methods that preserve the original structure of the network, being the most important of them the inference algorithm over *polytrees*² of Judea Pearl [21; 22]. The methods that change the structure of the network can be divided into two categories: those based on the *variable elimination technique* [23; 24; 25], and the algorithms that are based on making *subgroups of nodes and passing messages between them* [10; 20; 26].

3.5.2.1 Variable elimination technique

This technique is based on successively removing variables from a Bayesian network while maintaining its ability to answer queries of interest. Consider a

¹NP-complete is a class of decision problems where each problem C is in NP and every problem in NP is reducible to C in polynomial time.

²A polytree is a tree where the nodes can have more than one parent.

3.5 Inference in Bayesian networks

Bayesian network defined over a set of variables $\mathbf{X} = \{X_1, \dots, X_n\}$, and some evidence \mathbf{e} . The goal would be to obtain the posterior distribution of a variable of interest, X_i , given the evidence. To do so, an ordering for removing the variables is defined, being X_i the last one. For each step of the algorithm, a variable is removed and its information is incorporated to the simplified network. At the end of the algorithm we obtain a potential over the variable of interest, and the posterior probability that we were looking for is proportional to this potential.

The detailed process is defined in Algorithm 1, where we start from a Bayesian network $\mathcal{B} = \{\mathcal{G}, \mathbf{P}\}$ defined over a set of variables $\mathbf{X} = X_1, \dots, X_n$, being the variable of interest X_i , and we have some evidence \mathbf{e} . The first step is to integrate the evidence in the network (line 4). To do so, every probability distribution in \mathbf{P} is restricted to the evidence. The next step would be to define an ordering of the variables (line 5) such as the last variable is X_i . The search for the optimal node elimination sequence is, in general, NP-hard, so we need to rely on heuristics to solve it. The choice must be carefully taken because the ordering will critically affect the efficiency of the method [27; 28; 29; 30]. The next step of the algorithm would be to iterate over the variables, following the order established, combining the potentials related to each one (lines 6 to 12). When the loop ends, we obtain a list of potentials over exclusively X_i . These potentials are then combined to obtain a single potential (line 13) that after normalising (line 14) becomes the probability distribution of X_i given the evidence \mathbf{e} .

```

1 Variable_Elimination((B),Xi,e)
   Input: A BN (B) = {X, G, P} defined over a set of variables
           X = {X1, ..., Xn}, the evidence e and a variable of interest Xi.
   Output: P(Xi|e).
2 begin
3   Let L be a list of potentials with all the distributions in P, {φ1, ..., φn};
4   Integrate e;
5   Define an elimination ordering σ that contains all the variables but Xi;
6   foreach k = 1 until (n - 1) do
7     Xk ← σ(k);
8     Let F be the subset of potentials from L that are defined over Xk;
9     L = L - F;
10    φ' = ∑Xk(∏φ∈F φ);
11    L = L ∪ φ';
12  end
13  Combine in φ all the potentials in L;
14  Normalise φ to obtain P(Xi|e);
15  return P(Xi|e);
16 end

```

Algorithm 1: Variable elimination algorithm

Example 13 Consider the Bayesian network detailed in Fig. 3.17. We want to apply the Variable Elimination algorithm (Alg. 1) to compute the posterior probability distribution of X_3 given the evidence $e = \{X_2 = 0\}$.

The first step would be to integrate the evidence (line 4 of Alg. 1), so we reduce the complexity of the problem as we only take into account the values in the potentials that are consistent with the configuration $e = \{X_2 = 0\}$. After this step we can eliminate X_2 from the Bayesian network by multiplying the corresponding potentials, obtaining the model in Fig. 3.18.

Now we only have X_1 left to be removed from the network, so the next step would be to combine both remaining potentials, as they both include X_1 in their domains. After the combination, we marginalise out X_1 (line 10 of Alg. 1). The result of this step is the posterior probability distribution of X_3 given the evidence $e = \{X_2 = 0\}$ as shown in Fig. 3.19.

3.5 Inference in Bayesian networks

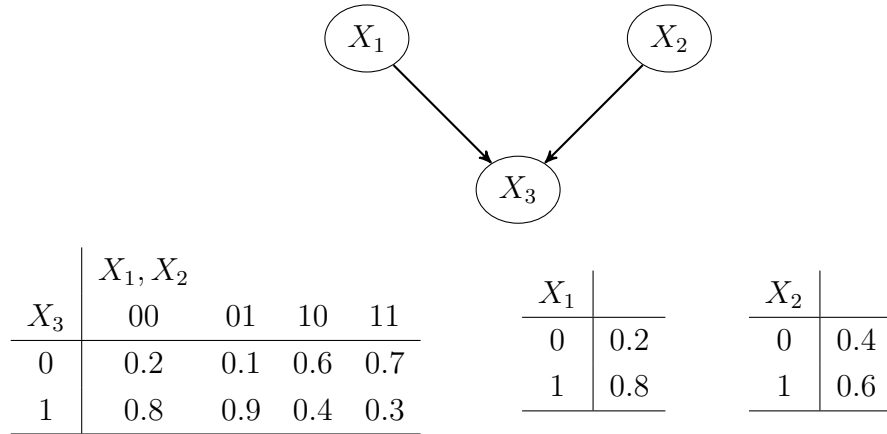


Figure 3.17: Bayesian network of three binary variables.

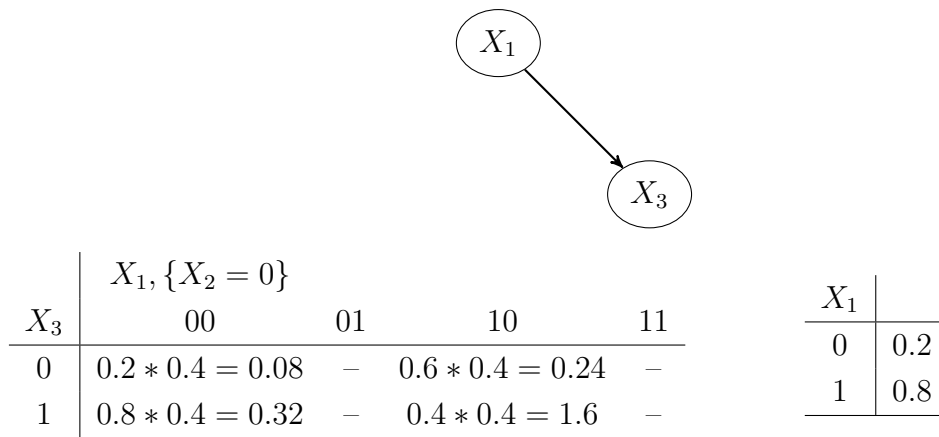


Figure 3.18: Intermediate step of the Variable Elimination process.

X_3	
0	0.52
1	0.48

Figure 3.19: Result of the Variable Elimination algorithm.

3.5.2.2 Message passing technique

The algorithms that use this technique are also known as *grouping methods*, as their underlying idea is to build an alternative representation of the graph where each supernode contains a subset of the original nodes, capturing the local structures associated with the original graph, and then propagate over this structure, where the computations will be local (dependent of a smaller group of variables).

This auxiliary structure is called *join tree* [10]. A *join tree* is a tree where each node V is a subset of the variables in the network, and such that if a variable is in two different nodes, V_1 and V_2 , then it is also in every node in the path between V_1 and V_2 . Every potential in the original Bayesian network (i.e. every conditional distribution) is assigned to a node V_j containing the variables involved in the conditional distribution. A potential constantly equal to 1 (unity potential) is assigned to nodes which did not receive any conditional distribution. In this way, attached to every node V_i there will be a potential ϕ_i defined over the set of variables V_i and which is equal to the product of all the potentials assigned to it.

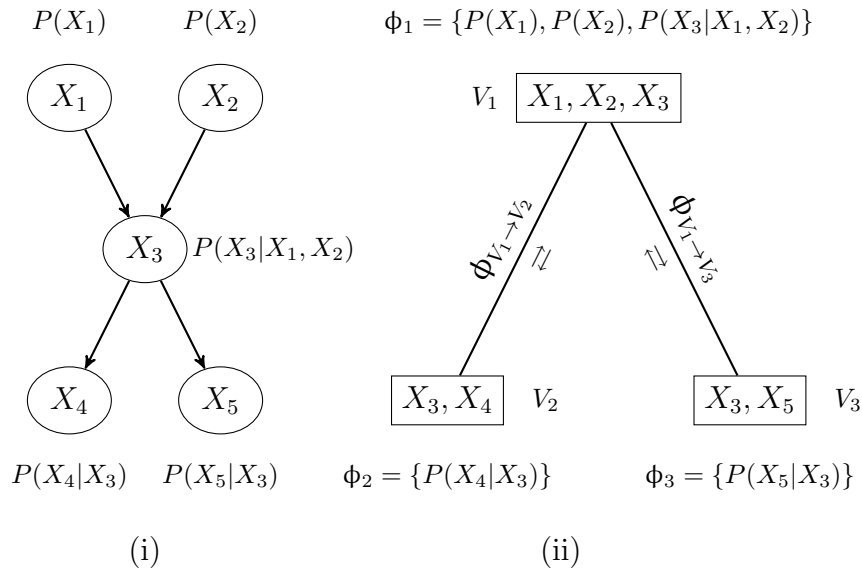


Figure 3.20: A Bayesian network (i) and a join tree (ii) associated with it. Probability propagation is carried out sending messages throughout the edges.

It is possible to keep the potentials assigned to a node as a list instead of multiplying them initially [11; 14] in what is called *lazy propagation*. This methodology

tries to save computation effort by postponing the actual combinations of potentials until the result is required. Figure 3.20 illustrates the process of probability propagation: A join tree (ii) is constructed from a Bayesian network (i) and then propagation is carried out by a flow of messages through the edges of the join tree. Observe that, in this example, the potentials in the nodes are kept as a list.

A message from one node V_i to one of its neighbours, V_j is a potential defined for the variables contained in $V_i \cap V_j$, and is obtained as the result of removing from the potentials attached to V_i all the variables not in V_j . A variable is removed by multiplying the potentials containing it and then summing the variable out. This is precisely the step in which the complexity of probability propagation arises, as it happened in the Variable Elimination algorithm: The domain of the potential resulting from the product above mentioned may become so large that a huge amount of memory would be necessary to store it.

3.5.3 Approximate inference algorithms

The methodology explained above may become infeasible because of the complexity of the networks, so an approximation is the tradeoff between computational cost and accuracy of the results. The main algorithms developed for approximate inference in Bayesian networks can be classified into two main categories: *deterministic algorithms*, those which obtain an approximation for $P(X_i|\mathbf{e})$ that is always the same for any execution of the algorithm given equal inputs; and the category containing the algorithms based on *simulation*, also called *Monte Carlo algorithms*. In general, a Monte Carlo algorithm generates a sample from $P(\mathbf{X}|\mathbf{e})$, and then estimates $P(X_k|\mathbf{e})$ as the relative frequency of the values of X_k in the sample.

3.5.3.1 Deterministic algorithms

Deterministic algorithms can be classified into two subgroups: the algorithms based on simplifying the model, and the search algorithms.

The algorithms in the first category simplify the model in order to perform exact propagation afterwards, but now over a tractable version of the model.

There are several approaches for the deterministic algorithms based on building a simplification of the model:

- Replace by zero small probability values, also called *annihilating small probabilities* [31].
- Remove arcs in the graph [32; 33]. The approach consists of deleting arcs between variables that are almost independent in order to simplify the topology of the network. They use the *Kullback-Leibler divergence* to quantify the *strength* of the arcs. The Kullback-Leibler (KL) divergence is a non-symmetric measure of the difference between two probability distributions P and an approximation P' . For discrete probability distributions P and P' , the KL divergence of P' from P is defined to be:

$$D_{KL}(P||P') = \sum_i \log\left(\frac{P(i)}{P'(i)}\right)P(i). \quad (3.10)$$

- Remove variables (the localized partial evaluation algorithm [34]). This algorithm consists of removing the variables that are too far from the variables of interest. Its basic version uses the message passing scheme by Pearl.
- Reduce the cardinality of the conditional probability distributions, or *state space abstraction* [35]. The algorithm can be used when the original variables were continuous and discretized, and it consists of propagating successively using an increasing number of states in the discretized variables.
- Use **alternative representations** for the conditional probability distributions, other than tables, where similar probabilities are merged. This methodology can be based on rules [36; 37], or on tree-based structures like probability trees [2; 14; 38]. This last option is the methodology followed during the whole dissertation, where we present a generalisation of this framework.

The second approach among the deterministic inference algorithms is to follow the hypothesis that a relatively small fraction of the joint probability distribution contains the majority of the probability mass. These algorithms look for the

configurations with high probability and use them to obtain the approximations. There are several examples:

- The search of the N configurations with higher probability [39].
- Search methods for high-probability configurations [40].
- Approximate search method using *conflicts*¹ [41; 42; 43].

3.5.3.2 Simulation algorithms: Monte Carlo

As introduced at the beginning of this section, the algorithms based upon *simulation* consist of generating a sample from $P(\mathbf{X}|\mathbf{e})$ and then estimating $P(X_i|\mathbf{e})$ as the relative frequency of X_i in the sample. So the process of stochastic simulation of a Bayesian network can be described as follows: estimate $P(X_i|\mathbf{e})$ by sampling a large number of random configurations over all the noninstantiated variables in the Bayesian network. The configurations that are inconsistent with \mathbf{e} are discarded, and if n is the total number of resulting cases, then $P(X_i|\mathbf{e}) \approx \frac{n(X_i)}{n}$, computed for each possible state of X_i .

We can distinguish two groups of Monte Carlo algorithms: those based on *Gibbs sampling* [44; 45] and those based on *importance sampling* [16; 17; 23; 46; 47; 48; 49; 50].

- Gibbs sampling: this method differs from the general stochastic simulation in the way the samples are generated. In Gibbs sampling, a sample is generated by starting from a valid configuration (for instance, the result of a sweep of stochastic simulation) and randomly changing the state of the variables, following the topological order. In this algorithm there is no instance discarding, but the issue is to choose a correct first instance.
- Importance sampling: the idea behind this methodology is that certain values of the input variables in the simulation have more impact on the parameter being estimated than others, and so, the method try to emphasize

¹In this context, a *conflict* is an assignment of a subset of variables that has a probability value close to zero.

this peculiarity by sampling more frequently those values, using a different distribution. This introduced error is compensated by weighting the simulation outputs.

Particular extensions of these methodologies are blocking Gibbs sampling [44] and importance sampling based on approximate precomputation [49; 50]. In both cases, the goal is to draw samples from a probability distribution that is extremely difficult to manage because of its size.

3.5.3.3 Comparing approximate inference algorithms: Fertig and Mann’s divergence

Further in this dissertation we will need to compare different approximate inference algorithms, and to do so we have chosen Fertig and Mann’s divergence [51]. The metric is defined as follows. For one variable X , the error is computed as

$$G(X) = \sqrt{\frac{1}{|\Omega_X|} \sum_{x \in \Omega_X} \frac{(\hat{P}(x|\mathbf{e}) - P(x|\mathbf{e}))^2}{P(x|\mathbf{e})(1 - P(x|\mathbf{e}))}}}, \quad (3.11)$$

where $P(x|\mathbf{e})$ is the exact *posterior* probability, $\hat{P}(x|\mathbf{e})$ is the approximate value and $|\Omega_X|$ is the number of possible values of variable X . For a set of variables \mathbf{X} , the error is:

$$G(\mathbf{X}) = \sqrt{\sum_{X \in \mathbf{X}} G(X)^2}. \quad (3.12)$$

Fertig and Mann’s divergence is an appropriate measure for comparing approximate inference algorithms, as it takes into account the magnitude of the exact value when evaluating the approximation. More precisely, it gives more weight to errors made when estimating extreme probabilities (close to 0 or 1), as it can be easily checked that the denominator in Eq. 3.11 is maximised for $P(x|\mathbf{e}) = 0.5$.

3.6 Learning Bayesian networks

In this section we define the basic steps to build a Bayesian network, and we give details on some of the most widely known algorithms. In general, when defining a problem using a Bayesian network, there are three main points to take into account:

1. *Definition of the network variables and their values:* we need to define all the entities that interact in the situation we are modelling, that will become the nodes in the DAG.
2. *Definition of the network structure:* at this step, we need to determine the interactions between the variables, or in other words, the edges in the DAG.
3. *Definition of the network's conditional probability distributions:* at this point we have to specify all the probability values that fill the conditional probability distributions defined by the relations between the variables, specified in the previous step.

This process could be performed entirely by hand, however, even a modestly sized network requires a skilled knowledge engineer spending a considerable amount of time with one or more domain experts. This approach has several problems: in some domains, the amount of knowledge required to model the problem is just too large, or the expert's time is just too expensive. In other domains, there are simply no experts that have enough understanding of the problem to define even the more subtle relations. Also, some domains change over time, and we cannot expect to have an available expert to redesign the whole system all over again.

The usual situation is to have a database available with a record of several instantiations of a set of variables altogether, so we can build the Bayesian network from it, with or without the help of the expert¹. This database can be *preprocessed* in order to identify, for example, a subset of variables that is more relevant for predicting than the whole set in order to obtain a smaller and more

¹If we have an expert available, we can combine an automated learning process with the incorporation of expert knowledge in order to increase the quality of the representation.

accurate representation. In the preprocessing stage it is usually necessary as well to take care of non valid or non existent values, that can exist in the database due, for instance, to errors in the measurement equipment. After preprocessing the database, we obtain the set of variables with their correspondent domains and a clean set of values such that we can build a Bayesian network from it.

We differentiate two stages in the automated learning of Bayesian networks: learning the structure of the network and learning the parameters. Both tasks are dependent on each other, and usually they are performed iteratively. In the following we give an explanation of both tasks and give examples of the most widely known algorithms to perform such operations.

3.6.1 Structural learning

The task of learning the structure of the network can be performed using different methodologies. Here we describe the two main ones: methods that use *independence tests* to identify the relations between the variables, and methods that search in the space of all possible networks trying to optimize a *quality measure*.

3.6.1.1 Independence tests

The algorithms that fall into this category try to detect the relationships between the variables using independence tests. Generally, they start with the *complete graph*: a graph that contains all the variables as its nodes and all and every one of them are connected to each other, and afterwards some arcs are deleted according to the result of independence tests.

A classic algorithm that follows this methodology is the PC algorithm [52]. The general idea behind this algorithm is to begin with the complete graph for the model, and remove arcs according to the results of independence tests that grow in complexity on each sweep of the algorithm: it first tests all the pairs of variables, checking the conditional independence of level 0. After checking all the pairs, it tests conditional independencies of level 1, which means whether two variables are independent given a third one, that must be adjacent to either one of the variables being tested. The algorithm keeps adding complexity to the independence tests until reaching a certain level m .

3.6.1.2 Score and search

This methodology has two main elements: on one hand it has a *search technique* used to explore the space of *candidate networks*, and on the other hand there is an evaluation measure or *metric* that each specific algorithm uses for evaluating each candidate. This metric measures the degree of fitness between the graph and the available data, and therefore the learning of an structure can be seen as an optimization procedure where a candidate network with optimal fitness is searched.

Metrics

A valid metric should be able to assign a score to all the candidate networks in the domain so we can define an ordering with the goal of choosing the best network. The metric should take into account how well the available data fits into the candidate network, how well the network fits within the expert information (in case that we have it) and finally, the metric should also take into account the network complexity, so that in case that we had two networks that equally fit the data, we would always prefer the less complex. Besides, some metrics are *score equivalent*, which means that equivalent DAG models have the same marginal likelihood [6; 53]. This is a desirable property for a good metric, along with decomposability: it is preferable to be able to decompose the metric such as small changes in the candidate network can be evaluated without having to reevaluate the whole network.

- The K2 metric [6; 53; 54]. It is not score equivalent, but it is efficient and obtains good results in practice. The quality of a graph \mathcal{G} given the data \mathcal{D} is defined, according to the K2 metric, as:

$$K2(\mathcal{G}|\mathcal{D}) = \sum_{i=1}^n \left[\sum_{k=1}^{s_i} \left[\log \frac{\Gamma(r_i)}{\Gamma(N_{ij} + r_i)} + \sum_{j=1}^{r_i} \log \Gamma(N_{ijk} + 1) \right] \right] \quad (3.13)$$

where:

- Γ is the *Gamma* function.

- $r_i = |\Omega_{X_i}|$, the number of cases of variable X_i .
 - x_{ik} is the k -th value of X_i .
 - π_{ij} is the j -th value of π_{X_i} , the set of parents of X_i .
 - $s_i = \prod_{x_j \in \pi_{X_i}} r_j = |\pi_{X_i}|$, the number of cases of π_{X_i} .
 - $N_{ijk} = n(x_{ik}|\pi_{ij})$, the number of cases in the database consistent with the given configuration of X_i and its parents.
 - $N_{ij} = \sum_{k=1}^{r_i} N_{ijk}$.
- The Bayesian Information Criterion (BIC metric) [6; 55]. It is a likelihood criterion penalized by the model complexity, measured as the number of free parameters. It is defined as:

$$BIC(\mathcal{G}|\mathcal{D}) = \sum_{i=1}^n \sum_{j=1}^{r_i} \sum_{k=1}^{s_i} N_{ijk} \log \frac{N_{ijk}}{N_{ik}} - \frac{1}{2} C(\mathcal{G}) \log N. \quad (3.14)$$

where $C(\mathcal{G}) = \sum_{i=1}^n (r_i - 1)s_i$ is a complexity measure of the network and N is the total number of cases in the database.

- The Bayesian Dirichlet equivalent (BDe) metric [6], which is based on the concept of sets of likelihood equivalent network structures, where all members in a set of equivalent networks are given the same score.

Search technique and search space

In general, we need to define a search space where we will look for possible candidate networks to be evaluated. Examples of these search spaces are the *DAG space*, the *order space* or the space of *equivalence classes*¹. A set of operators is defined to be able to travel from one candidate to the next in the defined search space. The usual operators used when learning Bayesian networks are:

- The **addition** of a new arc in the network.

¹The space of equivalence classes are graphs that represent the same (in)dependence relations.

- The **removal** of an existing arc in the network.
- The **reversal** of an arc. This operator can be seen as a composition of the removal of an arc and the addition of a new one that goes in the opposite direction.

Once the search space and the operators to explore it are defined, the next step is to choose a search technique. There are several classical techniques available, which are based in local search: hill climbing, tabu search and simulated annealing, for instance. Another approach is to define an ordering of the variables and use it during the search. An example of this methodology is the K2 algorithm, that starts with the *empty network*¹ and adds arcs when the addition produces an increment in the K2 metric.

3.6.2 Parametric learning

Once the structure of the network is defined, and so the set of parents of each variable, we need to estimate the conditional probability distributions associated to each node in the network that define the joint probability distribution of the variables. Again, the probabilities could be elicited by one or more experts, but this task can become infeasible when the networks are very large or the relationships between variables are too subtle for the expert's understanding. There are several methods to automate the task of estimating the probabilities from a database which are outlined below.

Maximum likelihood approach

This first approach is based on the *likelihood principle*, which favours those estimates that have a maximal likelihood, that is, ones that maximize the probability of observing the given data set. This is probably the most common approach, where each probability is computed from the sample as the relative frequency of

¹An *empty network* has only nodes and no arcs.

3.7 Alternative representations of potentials

the values. Let $n(x_i)$ be the number of times that the variable X_i takes the value x_i , then the computation of the estimator is defined as:

$$P(x_i|\pi_{x_i}) = \frac{n(x_i, \pi_{x_i})}{n(\pi_{x_i})}. \quad (3.15)$$

This methodology has two main drawbacks: on one hand, the sample needs to be long, and if a given configuration has not been observed, its probability is estimated as zero (furthermore, Eq. 3.15 might not be even defined for some configurations of the parents). On the other hand, this estimator tends to overfit the data, leading to poor prediction power in models learned from small samples.

Laplace smoothing

Laplace smoothing [56] tries to overcome the problems of the maximum likelihood approach, by redefining the estimator as:

$$P(x_i|\pi_{x_i}) = \frac{n(x_i, \pi_{x_i}) + 1}{n(\pi_{x_i}) + |\Omega_{X_i}|}. \quad (3.16)$$

Equation 3.16 is always defined, and has the following properties:

- If the sample is small, it is close to a uniform distribution. So if it is the case that a given configuration of the parents has no observations, the estimator corresponds exactly to a uniform distribution.
- If the sample is large enough, the estimator defined in Eq. 3.16 tends to the value of the maximum likelihood one, as the values 1 and $|\Omega_{X_i}|$ are insignificant compared to the frequencies.

3.7 Alternative representations of potentials

In this section we review some data structures to represent potentials. Along this Chapter, we have used *probability tables* to illustrate the concept of potential and the operations performed with them. This representation is exhaustive, which means that we declare all the possible configurations of the variables in a given potential, and specify the probability values associated to them. Therefore, the

3.7 Alternative representations of potentials

size of the representation can become very large as the number of variables in the potential grows, and if we are representing a *conditional probability distribution*, the size of the representation grows exponentially in the number of parents of the distribution.

Also, the graphical structure of a Bayesian network can only capture independence relations of the form $I(\mathbf{X}, \mathbf{Y}|\mathbf{Z})$, that is, independencies holding for any configuration of the variables in \mathbf{Z} . However, we are often interested in independencies that hold only in certain contexts. This situation is called *context-specific independency* [57], and if we used a probability table to represent a distribution with context-specific independencies within it, it would mean that the table would store repeated probability values in several rows, depending on how precise the context where the independence is defined is.

Example 14 *For example, if we consider the potential fully defined in the left part of Fig. 3.21, we see that there are many repeated values in the table. The full potential can be defined with only four different values, as expressed in the right part of Fig. 3.21. There, the context-specific independencies (for the contexts $\{X_1 = 0, X_3 = 0\}$ and $\{X_1 = 1\}$) are emphasized. Note that we use reduced configurations to express context-specific independencies, as the value for $\{X_1 = 0, X_3 = 0\}$ is the same for $\{X_1 = 0, X_3 = 0, X_2 = 0\}$ and for $\{X_1 = 0, X_3 = 0, X_2 = 1\}$. The same happens for the value of the configuration $\{X_1 = 1\}$ in Φ , we do not need to specify the states of X_2 and X_3 because the value of the configuration is the same for each possible combination of the two of them.*

In this Section we review some alternative representations for probabilistic potentials that attempt to reduce the number of parameters used to define the potentials involved in probabilistic graphical models, either taking advantage of context-specific independencies present in the data, or reformulating the nature of the model.

3.7.1 Probability Trees

Probability trees [17] have been used as a flexible data structure that enables the specification of *context-specific independencies* [57] as well as using exact or

3.7 Alternative representations of potentials

X_1	X_2	X_3	$\phi(X_1, X_2, X_3)$					
0	0	0	0.1					
0	0	1	0.8					
0	1	0	0.1					
0	1	1	0.2	<div style="display: flex; align-items: center; justify-content: center;"> <div style="font-size: 2em; margin-right: 10px;">➤</div> <table style="border-collapse: collapse; border: none;"> <tr style="border-top: 1px solid black; border-bottom: 1px solid black;"> <td style="padding: 5px;">$\phi(X_1 = 0, X_3 = 0) = 0.1$</td> </tr> <tr> <td style="padding: 5px;">$\phi(X_1 = 0, X_2 = 0, X_3 = 1) = 0.8$</td> </tr> <tr> <td style="padding: 5px;">$\phi(X_1 = 0, X_2 = 1, X_3 = 1) = 0.2$</td> </tr> <tr style="border-bottom: 1px solid black;"> <td style="padding: 5px;">$\phi(X_1 = 1) = 0.3$</td> </tr> </table> </div>	$\phi(X_1 = 0, X_3 = 0) = 0.1$	$\phi(X_1 = 0, X_2 = 0, X_3 = 1) = 0.8$	$\phi(X_1 = 0, X_2 = 1, X_3 = 1) = 0.2$	$\phi(X_1 = 1) = 0.3$
$\phi(X_1 = 0, X_3 = 0) = 0.1$								
$\phi(X_1 = 0, X_2 = 0, X_3 = 1) = 0.8$								
$\phi(X_1 = 0, X_2 = 1, X_3 = 1) = 0.2$								
$\phi(X_1 = 1) = 0.3$								
1	0	0	0.3					
1	0	1	0.3					
1	1	0	0.3					
1	1	1	0.3					

Figure 3.21: Potential of three binary variables with context-specific independencies.

approximate representations of potentials. A *probability tree* \mathcal{PT} is a directed labelled¹ tree, in which each internal node represents a variable and each leaf² represents a non-negative real number (in this context, we will say that a node in a probability tree is labelled either with a variable or with a probability value). Each internal node has one outgoing edge for each state of the variable that labels that node; each state labels one edge. The *size* of a tree \mathcal{PT} , denoted by $size(\mathcal{PT})$, is defined as its number of leaves.

Formally, a probability tree \mathcal{PT} on variables $\mathbf{X}_I = \{X_i | i \in I\}$ represents a potential $\phi : \Omega_{\mathbf{X}_I} \rightarrow \mathbb{R}_0^+$ if for each $\mathbf{x}_I \in \Omega_{\mathbf{X}_I}$ the value $\phi(\mathbf{x}_I)$ is the number stored in the leaf node that is reached by starting from the root node and selecting the child corresponding to coordinate x_i for each internal node labelled X_i . A subtree of \mathcal{PT} is called a *terminal tree* if it contains only one node labelled with a variable, and all the children are numbers (leaf nodes).

A probability tree is usually a more compact representation of a potential

¹In general, a *labelled* graph is a graph whose edges have a label associated to it. In the case of probability trees, the labels of the edges correspond to the states of the variable represented by the source node.

²In a probability tree, there is only one node that has no parents, and it is called the *root* node. At the same time, there can be several nodes that have no children. These nodes are called *leaves* of the probability tree. A probability tree is generally traversed from root to leaves.

3.7 Alternative representations of potentials

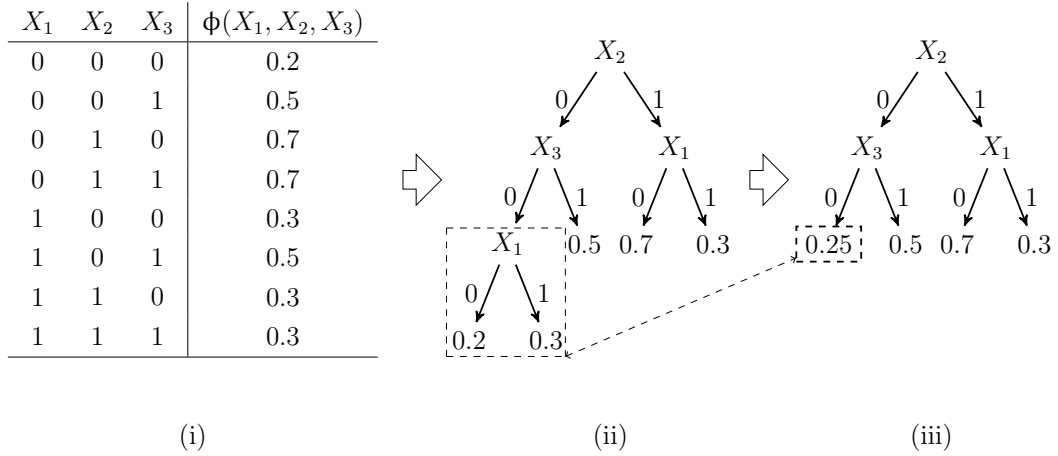


Figure 3.22: Potential $\phi(X_1, X_2, X_3)$ (i), its representation as a probability tree (ii) and its approximation after pruning several branches (those rounded with a rectangular box) (iii).

than a table.

Example 15 *This situation is illustrated in Fig. 3.22, which displays a potential $\phi(X_1, X_2, X_3)$ stored as a probability table (i) and its representation using a probability tree (ii). It can be seen that the values in the potential are independent of X_3 in the context $\{X_2 = 1\}$. This context-specific independence is reflected in the tree in Fig. 3.22 (ii): it contains the same information as the table, but only requires five values rather than eight.*

Furthermore, trees enable even more compact representations in exchange for loss of accuracy. There are several ways of constructing and *approximate tree* [58]. One of the alternatives consists of adding nodes until an exact representation is achieved or a maximum number of nodes is reached. Other alternative consists of building the full tree (an exact representation of the potential) and *prune* it afterwards. The *pruning* of a probability tree is a mechanism that consists of removing certain leaves and replacing them, for instance, with the average value, as shown in Fig. 3.22 (iii). Again, there are several alternatives to decide when to prune a subtree, being the most straightforward to prune when the

3.7 Alternative representations of potentials

values on the leaves are too similar. We will denominate *slight pruning* when the operation is only performed when the difference between the values is small. In contrast, we are performing *severe pruning* when we prune subtrees whose leaves are significantly different.

3.7.1.1 Operations over Probability Trees

The basic operations (*combination*, *marginalisation* and *restriction*) in potentials that were introduced in Section 3.5.1 can be performed directly on probability trees [17].

Restriction

If \mathcal{PT} is a probability tree on \mathbf{X}_I and $\mathbf{X}_J \subseteq \mathbf{X}_I$, $\mathcal{PT}^{R(\mathbf{x}_J)}$ (probability tree restricted to the configuration \mathbf{x}_J) denotes the *restriction operation* which consists of returning the part of the tree which is consistent with the values of the configuration $\mathbf{x}_J \in \Omega_{\mathbf{X}_J}$.

Example 16 For example, in Fig. 3.22 (ii), $\mathcal{PT}^{R(X_2=0, X_3=0)}$ represents the terminal tree enclosed by the lined square.

This operation is an important part of both combination and marginalisation operations and is used for conditioning potentials to a given configuration.

Combination

The combination of two probability trees can be performed recursively using the procedure explained in Algorithm 2. Given two probability trees \mathcal{PT}_1 and \mathcal{PT}_2 , the result of the combination of both is a probability tree \mathcal{PT}_c defined over a set of variables that is the union of the sets of variables where \mathcal{PT}_1 and \mathcal{PT}_2 are defined, respectively. The underlying idea is to restrict both trees to a configuration of the variables in \mathcal{PT}_c until we reach a pair of values that are consistent with the configuration, and then multiply those values. By doing this for each possible configuration of the variables of \mathcal{PT}_c , we fill its leaves, making the new probability tree the result of the combination of \mathcal{PT}_1 and \mathcal{PT}_2 .

3.7 Alternative representations of potentials

```

1 combine( $\mathcal{PT}_1, \mathcal{PT}_2$ )
   Input: Two probability trees  $\mathcal{PT}_1$  and  $\mathcal{PT}_2$ 
   Output: A probability tree  $\mathcal{PT}_c$ 
2 begin
3   Let  $L_1$  and  $L_2$  be the labels of the root nodes of  $\mathcal{PT}_1$  and  $\mathcal{PT}_2$ ,
   respectively.;
4   if  $L_1$  and  $L_2$  are numbers then
5     | Make  $L_1 \cdot L_2$  label the root of  $\mathcal{PT}_c$ ;
6   end
7   if  $L_1$  is a number and  $L_2$  is a variable then
8     | Make  $L_2$  label the root of  $\mathcal{PT}_c$ ;
9     foreach subtree  $\mathcal{PT}'_2$  child of  $\mathcal{PT}_2$  do
10    | Make  $\mathcal{PT}'_1 = \text{combine}(\mathcal{PT}_1, \mathcal{PT}'_2)$  be a child of  $\mathcal{PT}_c$ ;
11    end
12  end
13  if  $L_1$  is a variable ( $X_k$ ) then
14    | Make  $X_k$  label the root of  $\mathcal{PT}_c$ ;
15    foreach  $x_k$  state of  $X_k$  do
16    | Make  $\mathcal{PT}'_1 = \text{combine}(\mathcal{PT}_1^{R(x_k)}, \mathcal{PT}_2^{R(x_k)})$  be a child of  $\mathcal{PT}_c$ ;
17    end
18  end
19  return  $\mathcal{PT}_c$ ;
20 end

```

Algorithm 2: Combination of two probability trees

Example 17 Consider the two probability trees in Fig. 3.23. We want to apply Alg. 2 to combine them into a single probability tree.

3.7 Alternative representations of potentials

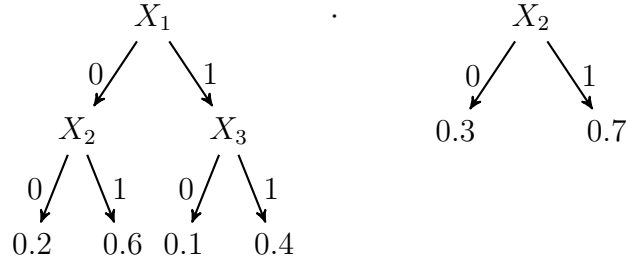


Figure 3.23: A probability tree \mathcal{PT}_1 defining a potential ϕ_1 over the set of variables $\{X_1, X_2, X_3\}$ to be combined with a probability tree \mathcal{PT}_2 that represents a potential ϕ_2 over the variable X_2 .

As the root of the first probability tree is labelled by a variable, X_1 , the root of the resultant probability tree will be labelled with X_1 (lines 13 and 14 of Alg. 2). For each state of the variable, we apply the algorithm recursively with both probability trees restricted to each context of X_1 , as shown in Fig. 3.24 (lines 15 to 17 of Alg. 2).

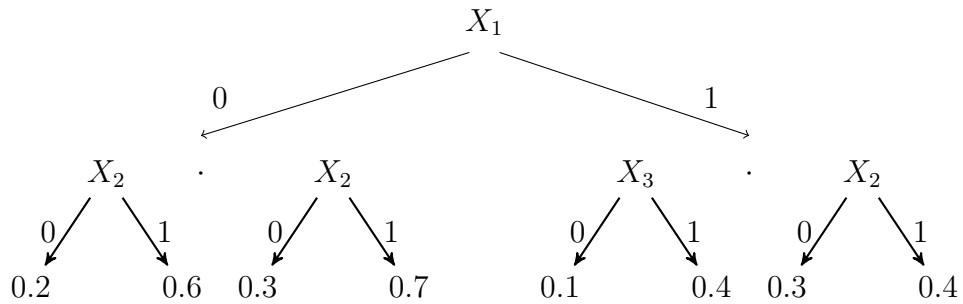


Figure 3.24: Propagation of Alg. 2 to the children of X_1 .

Both parts of the tree are now solved independently. In the left part of the tree, a new subtree rooted by variable X_2 is created (lines 13 and 14 of Alg. 2), and the recursive call ends by multiplying the numbers in the leaves (lines 4 to 6 of Alg. 2). In the right part of the tree, the new subtree is rooted by variable X_3 , and in this case every one of its children (in this example they are both numbers) are combined with a subtree rooted by X_2 . This scenario corresponds to lines 7 to 12 of Alg. 2, where a subtree rooted by X_2 is created, and is illustrated in Fig. 3.25.

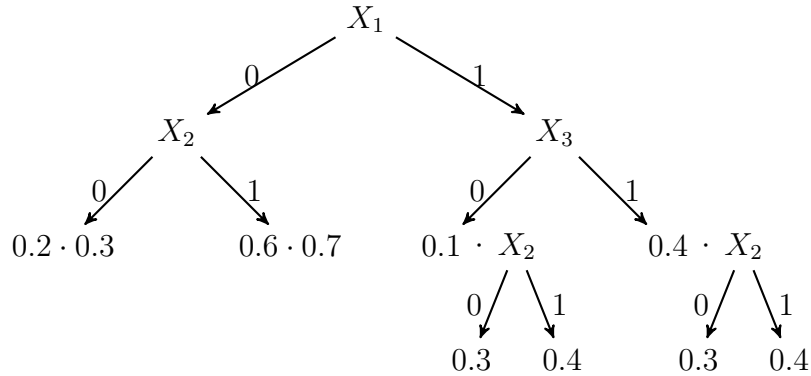


Figure 3.25: Intermediate step of Alg. 2.

The result of the operation is a new probability tree over the set of variables $\{X_1, X_2, X_3\}$ as shown in Fig. 3.26.

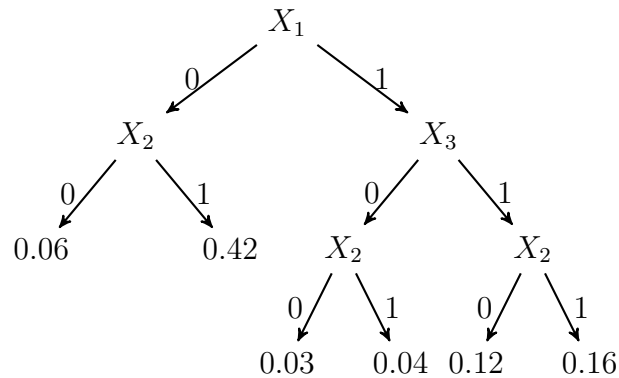


Figure 3.26: Probability tree resultant of applying Alg. 2 to combine the probability trees in Fig. 3.23.

Marginalisation

The procedure to marginalise out a variable from a probability tree is explained in Algorithm 3. The idea is to traverse the tree until we find the variable to marginalise out, then it is replaced by the addition of its children. The addition of two probability trees is explained in Algorithm 4, where the methodology is the same as in the general combination operation, but in the case of the addition, when we reach two probability values we add them instead of multiplying them. Both operations, marginalisation and addition of probability trees, are recursive.

3.7 Alternative representations of potentials

```

1 marginalise( $\mathcal{PT}, X_i$ )
  Input: A probability tree  $\mathcal{PT}$  and a variable  $X_m$  from the set of variables
           where  $\mathcal{PT}$  is defined.
  Output: A probability tree  $\mathcal{PT}_m$ .
2 begin
3   Let  $L$  be the label of the root of  $\mathcal{PT}$ ;
4   if  $L$  is a number then
5     | Make  $L \cdot |\Omega_{X_m}|$  label the root of  $\mathcal{PT}_m$ ;
6   else
7     | Let  $X_k$  be the variable corresponding to  $L$ ;
8     | if  $X_k == X_m$  then
9       | Let  $\mathcal{PT}_1, \dots, \mathcal{PT}_s$  be the children of  $\mathcal{PT}$ ;
10      | Let  $\mathcal{PT}_m = \mathcal{PT}_1$ ;
11      | foreach child  $\mathcal{PT}_i$  of  $\mathcal{PT}$  starting from  $i = 2$  do
12        | Make  $\mathcal{PT}_m = \text{add}(\mathcal{PT}_i, \mathcal{PT}_m)$  using Alg. 4;
13      | end
14    | else
15      | Create  $\mathcal{PT}_m$  with  $X_k$  labelling its root;
16      | foreach state  $x_k$  of  $X_k$  do
17        | Make  $\mathcal{PT}_h = \text{marginalise}(\mathcal{PT}^{R(x_k)}, X_i)$  be the next child of
18        |  $\mathcal{PT}_m$ ;
19      | end
20    | end
21  | return  $\mathcal{PT}_m$ ;
22 end

```

Algorithm 3: Marginalisation of a variable from a probability tree.

3.7 Alternative representations of potentials

```

1 add( $\mathcal{PT}_1, \mathcal{PT}_2$ )
   Input: Two probability trees  $\mathcal{PT}_1$  and  $\mathcal{PT}_2$ .
   Output: A probability tree  $\mathcal{PT}_a$ .
2 begin
3   Let  $L_1$  and  $L_2$  be the labels of the root nodes of  $\mathcal{PT}_1$  and  $\mathcal{PT}_2$ ,
   respectively.;
4   if  $L_1$  and  $L_2$  are numbers then
5     | Make  $L_1 + L_2$  label the root of  $\mathcal{PT}_a$ ;
6   end
7   if  $L_1$  is a number and  $L_2$  is a variable then
8     | Make  $L_2$  label the root of  $\mathcal{PT}_a$ ;
9     foreach subtree  $\mathcal{PT}'_2$  child of  $\mathcal{PT}_2$  do
10    | Make  $\mathcal{PT}'_1 = \text{add}(\mathcal{PT}_1, \mathcal{PT}'_2)$  be a child of  $\mathcal{PT}_a$ ;
11    end
12  end
13  if  $L_1$  is a variable ( $X_k$ ) then
14    | Make  $X_k$  label the root of  $\mathcal{PT}_a$ ;
15    foreach  $x_k$  state of  $X_k$  do
16    | Make  $\mathcal{PT}'_1 = \text{add}(\mathcal{PT}_1^{R(x_k)}, \mathcal{PT}_2^{R(x_k)})$  be a child of  $\mathcal{PT}_a$ ;
17    end
18  end
19  return  $\mathcal{PT}_a$ ;
20 end

```

Algorithm 4: Computation of the addition of two probability trees

Example 18 *The methodology to add two probability trees is almost equivalent with the combination of probability trees, as it can be seen in the following example. Consider the two probability trees in Fig. 3.27. We want to apply Alg. 4 to add them into a single probability tree.*

3.7 Alternative representations of potentials

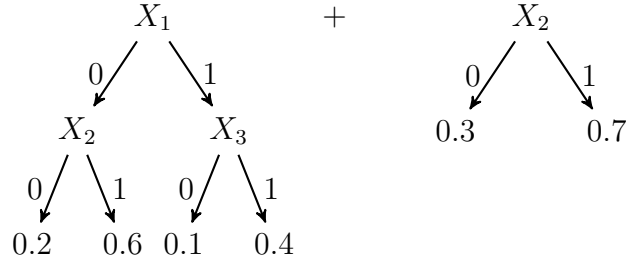


Figure 3.27: A probability tree \mathcal{PT}_1 defining a potential ϕ_1 over the set of variables $\{X_1, X_2, X_3\}$ to be added with a probability tree \mathcal{PT}_2 that represents a potential ϕ_2 over the variable X_2 .

As the root of the first probability tree is labelled by a variable, X_1 , the root of the resultant probability tree will be labelled with X_1 (lines 13 and 14 of Alg. 4). For each state of the variable, we apply the algorithm recursively with both probability trees restricted to each context of X_1 , as shown in Fig. 3.28 (lines 15 to 17 of Alg. 4).

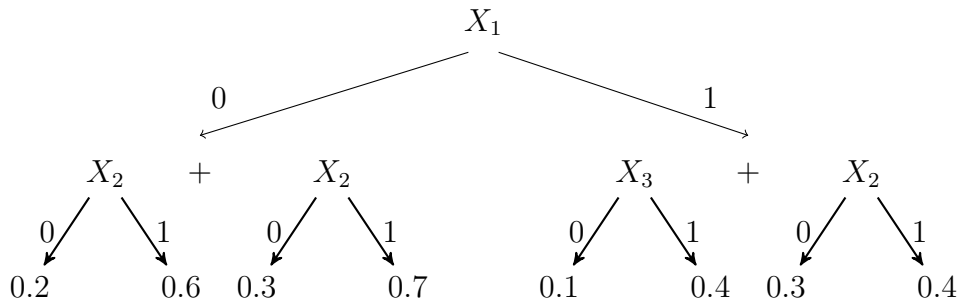


Figure 3.28: Propagation of Alg. 4 to the children of X_1 .

Both parts of the tree are now solved independently. In the left part of the tree, a new subtree rooted by variable X_2 is created (lines 13 and 14 of Alg. 4), and the recursive call ends by adding the numbers in the leaves (lines 4 to 6 of Alg. 4). In the right part of the tree, the new subtree is rooted by variable X_3 , and in this case every one of its children (in this example they are both numbers) are added with a subtree rooted by X_2 . This scenario corresponds to lines 7 to 12 of Alg. 4, where a subtree rooted by X_2 is created, and is illustrated in Fig. 3.29.

3.7 Alternative representations of potentials

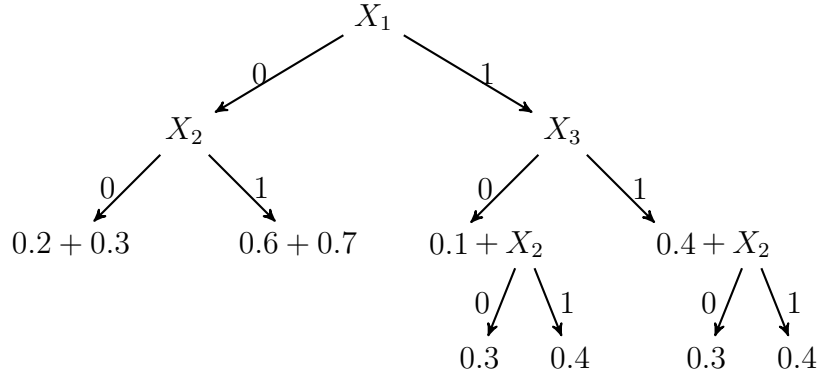


Figure 3.29: Intermediate step of Alg. 4.

The result of the operation is a new probability tree over the set of variables $\{X_1, X_2, X_3\}$ as shown in Fig. 3.30.

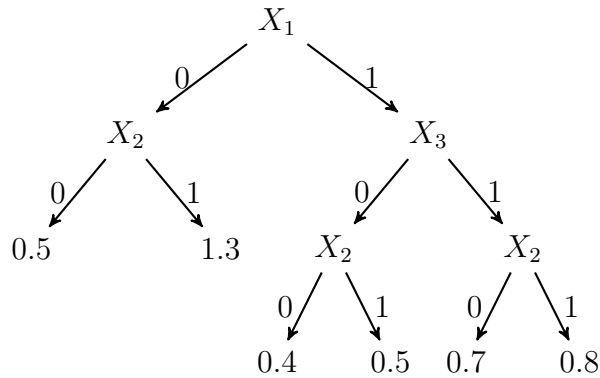


Figure 3.30: Probability tree resultant of applying Alg. 4 to add the probability trees in Fig. 3.27.

Example 19 Consider the probability tree over the set of variables $\{X_1, X_2, X_3\}$ shown in Fig. 3.31. We want to use Alg. 3 to marginalise out variable X_3 , so the result of the operation will be a probability tree defined over the reduced set of variables $\{X_1, X_2\}$.

3.7 Alternative representations of potentials

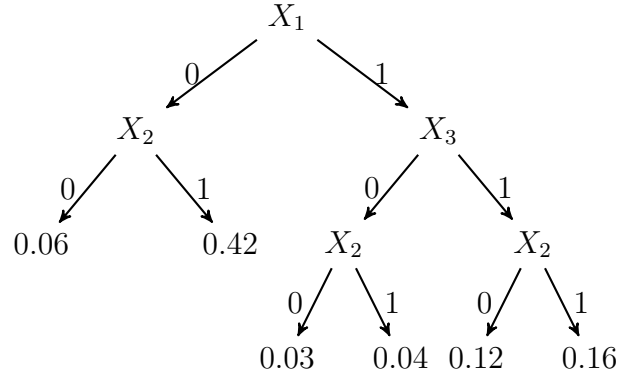


Figure 3.31: A probability tree defining a potential ϕ over the set of variables $\{X_1, X_2, X_3\}$.

Algorithm 3 starts by analysing the root of the tree, that is a variable different than the variable of interest X_3 . In this case, we build the root of the resultant tree with variable X_1 , and then Alg. 3 is recursively called for every context of the root (lines 15 to 19 of Alg. 3). Now, both the left and right parts of the tree are computed separately. In the left part of the tree, we find a subtree rooted by a variable different than X_3 , so we follow the same procedure. As its children are all values, the algorithm multiplies them by the number of states of the variable of interest (lines 4 and 5 of Alg. 3). In the right part of the tree, we find that the root of the subtree is our variable of interest, so we apply Alg. 4 to add all its children. The current state of the structure is shown in Fig. 3.32.

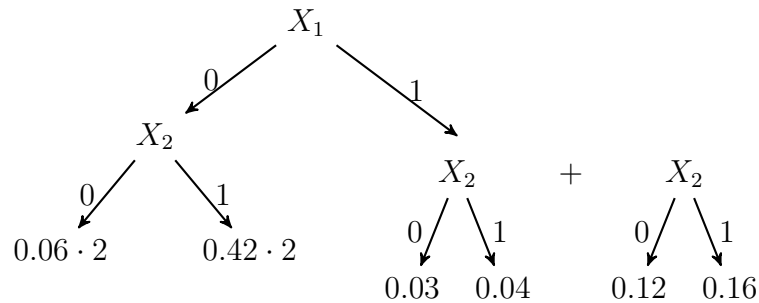


Figure 3.32: Intermediate step of Alg. 3.

After solving both parts of the tree, the final result is the probability tree shown in Fig. 3.33.

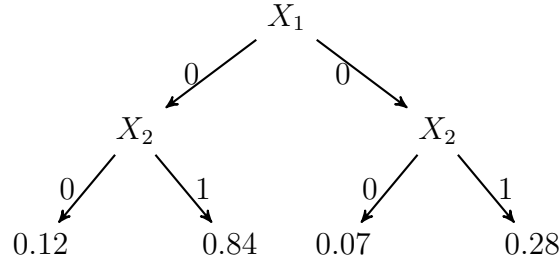


Figure 3.33: Probability tree resultant of applying Alg. 3 to marginalise out the variable X_3 from the probability tree in Fig. 3.31.

3.7.1.2 Inference with Probability Trees

Traditional inference algorithms as those introduced in Section 3.5 can be modified to take advantage of the nature of probability trees, as it is the case of the Variable Elimination algorithm [24; 25] explained in Section 3.5.2.1. Algorithm 5 shows the general workflow of the Variable Elimination algorithm, that is equivalent as the methodology explained earlier in this chapter, but adapted to work with probability trees instead of general potentials. This basically means that now the potentials are stored as probability trees, and when working with them we will be applying the basic operations (i.e. combination, restriction and marginalisation) specifically designed for probability trees, as explained in Section 3.7.1.1.

Observe that Algorithm 5 is an exact inference algorithm, as it does not apply any pruning over the probability trees that represent the distributions. Algorithm **Prune_VE**($\mathbf{B}, \mathbf{E}, \mathbf{e}, \alpha$) (Alg. 6) is an extension of Alg. 5 that carries out the approximation by pruning the probability trees corresponding to the initial conditional distribution in the network. The pruning is controlled by parameter α . Intuitively, this parameter indicates that sub-trees whose entropy is higher than the entropy of the binary probability distribution $\{0.5 - \alpha, 0.5 + \alpha\}$ will be replaced by a single value equal to the average of all the values in the subtree [17]. This approximation method based on tree pruning has been successfully used as the fundamental of various approximate algorithms [2; 17].

3.7 Alternative representations of potentials

```

1 Variable_Elimination( $\mathbf{X}, W, \mathbf{E}, \mathbf{e}, P$ )
   Input: The variables in the network ( $\mathbf{X}$ ), an observation  $\mathbf{E} = \mathbf{e}$ , the target
           variable ( $W$ ) and a set of probability trees,  $\mathbf{P}$ , over the variables in
            $\mathbf{X}$ .
   Output: The posterior distribution of  $W$  given  $\mathbf{E} = \mathbf{e}$ .
2 Let  $\mathcal{PT}_i, i = 1, \dots, k$ , be the probability trees in  $P$ .
3  $\mathbf{T} = \{\mathcal{PT}_i^{R(\mathbf{E}=\mathbf{e})}, i = 1, \dots, k\}$ .
4 foreach  $U \in \mathbf{X} \setminus \mathbf{E} \setminus \{W\}$  do
5   |  $\mathbf{T}_U = \{\mathcal{PT} \in \mathbf{T} \mid U \in \text{dom}(\mathcal{PT})\}$ .
6   | Let  $g_U$  be the product of the trees in  $\mathbf{T}_U$ .
7   | Let  $r_U$  be the result of marginalising out variable  $U$  from  $g_U$ .
8   |  $\mathbf{T} = (\mathbf{T} \setminus \mathbf{T}_U) \cup \{r_U\}$ .
9 end
10 Let  $\mathcal{PT}_f$  be the product of the trees in  $\mathbf{T}$ .
11 Normalise  $\mathcal{PT}_f$  in order to make it add up to 1.
12 return  $\mathcal{PT}_f$ .

```

Algorithm 5: Variable Elimination algorithm over probability trees.

```

1 Prune_VE( $\mathbf{B}, \mathbf{E}, \mathbf{e}, \alpha$ )
   Input: A Bayesian network  $\mathbf{B}$  and an observation  $\mathbf{E} = \mathbf{e}$ . A threshold  $\alpha$ 
           for pruning the initial distributions.
   Output: The posterior distribution of all the unobserved variables in the
           network, given  $\mathbf{E} = \mathbf{e}$ .
2 Let  $\mathbf{X}$  be the variables in  $\mathbf{B}$ .
3 Let  $\mathbf{PT} = \{\mathcal{PT}_i, i = 1, \dots, n\}$ , be the probability trees representing the
   conditional distributions in  $\mathbf{B}$ .
4 foreach  $\mathcal{PT} \in \mathbf{PT}$  do
5   | Let  $\mathcal{PT}'$  be the result of pruning  $\mathcal{PT}$  according to parameter  $\alpha$ .
6   |  $\mathbf{PT} \leftarrow (\mathbf{PT} \setminus \{\mathcal{PT}\}) \cup \{\mathcal{PT}'\}$ .
7 end
8  $\mathbf{R} = \emptyset$ .
9 foreach  $W \in \mathbf{X} \setminus \mathbf{E}$  do
10  |  $\mathcal{PT} \leftarrow \text{Variable\_Elimination}(\mathbf{X}, W, \mathbf{E}, \mathbf{e}, \mathbf{PT})$ .
11  |  $\mathbf{R} \leftarrow \mathbf{R} \cup \{\mathcal{PT}\}$ .
12 end
13 return  $\mathbf{R}$ .

```

Algorithm 6: Variable Elimination algorithm with tree pruning of the initial distributions.

3.7 Alternative representations of potentials

3.7.1.3 Proportional subtrees

We may also find within a probability tree that some subtrees are proportional. In this case, we can factorise the tree into a product of smaller trees, where the division may lead into an efficiency gain when performing inference.

Formally, given a probability tree \mathcal{PT} defined over Σ_X , let \mathbf{x}_c be a configuration that defines a path from the root to a variable X_i in \mathcal{PT} . We will say that \mathcal{PT} is proportional under X_i in the context \mathbf{x}_c if there is an $x_0 \in \Sigma_X$ such as for each $x_i \in \Sigma_X$, $\exists \alpha_i > 0$ such as:

$$\mathcal{PT}^{R(\mathbf{x}_c, x_i)} = \alpha_i \cdot \mathcal{PT}^{R(\mathbf{x}_c, x_0)}.$$

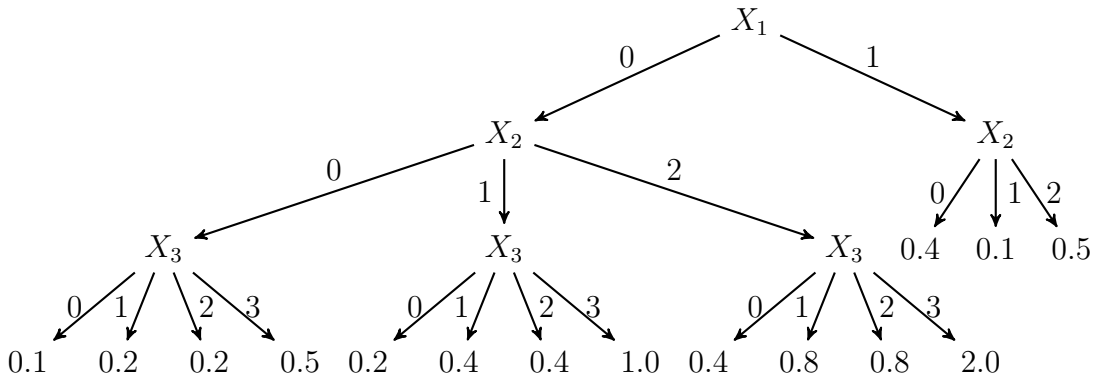


Figure 3.34: Probability tree with proportionalities and context-specific independencies.

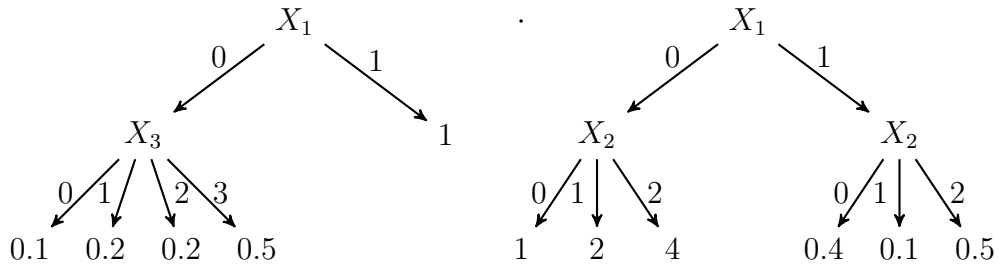


Figure 3.35: Factorisation of the tree in Fig. 3.34 as a product of smaller probability trees.

3.7 Alternative representations of potentials

Example 20 For example, the probability tree in Fig. 3.34 shows on one hand that the represented potential has some context-specific independencies, as for $X_1 = 1$, the value of the potential is independent of the value of X_3 . On the other hand, if we check the context $X_1 = 0$, we see that the values for X_3 are proportional for the different states of X_2 , being the proportionality factors (1, 2, 4) for each state of X_2 , sequentially. This probability tree can be divided into a product of two smaller probability trees, as shown in Fig. 3.35. As it can be seen in the figure, both factors together are smaller than the original probability tree, with the advantage that not all the variables are in both factors, and so the inference can benefit from this scenario to gain efficiency.

3.7.2 Binary Probability Trees

A *binary probability tree* \mathcal{BT} [59] is similar to a probability tree in the sense that it is also a directed labelled tree, where each internal node is labelled with a variable, and each leaf is labelled with a non-negative real number. It also allows a potential for a set of variables \mathbf{X}_I to be represented. The main differences with respect to a probability tree are that for a binary probability tree each internal node has always two outgoing arcs and a variable can appear more than once labelling the nodes in the path from the root to a leaf node. Another difference is that, for an internal node labelled with X_i , the outgoing arcs can generally be labelled with more than one state of the domain of X_i , Ω_{X_i} .

At a given node t of \mathcal{BT} , labelled with variable X_i , we denote with $\Omega_{X_i}^t$, $\Omega_{X_i}^t \subseteq \Omega_{X_i}$, the set of *available states* of X_i at node t . In general, this set is a proper subset of Ω_{X_i} . The available states of X_i at node t will be distributed between two subsets, in order to label its two outgoing arcs. We denote with $L_{lb(t)}$ and $L_{rb(t)}$ the labels (two subsets of $\Omega_{X_i}^t$) of the left and right branches of t . We denote with t_l and t_r the two children of t .

Example 21 For example, Fig. 3.36 (ii) shows a probability tree for the table in (i), and its equivalent as a binary probability tree is shown in (iii). In this last figure, the root is labelled with variable X_1 . The domain of X_1 , Ω_{X_1} , is the set of states $\{0, 1, 2\}$. At root node we have $\Omega_{X_1} = \{0, 1, 2\}$. That is, the available states of X_1 at the root node coincides with its domain. The left branch of the

3.7 Alternative representations of potentials

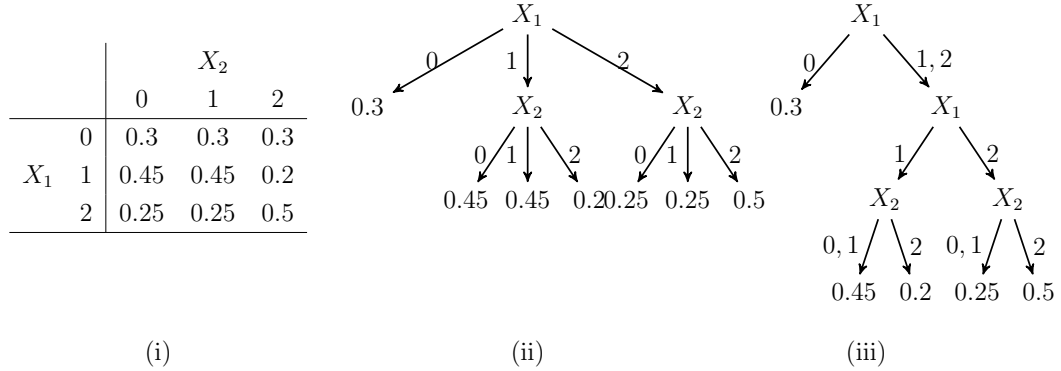


Figure 3.36: Probability distribution $P(X_1|X_2)$ as a table, as a probability tree and as binary probability tree

root is labelled with $\{0\}$ and the right branch with $\{1, 2\}$. If we traverse the tree following the right path, we reach a node also labelled with X_1 but this time the set of states is $\Omega_{X_1} = \{1, 2\}$. If we keep going down the tree, the nodes labelled with X_2 divide their states as well into two groups: $\Omega_{X_2}^{left} = \{0, 1\}$ and $\Omega_{X_2}^{right} = \{2\}$. It can be seen that the binary probability tree contains only five leaves, whereas the probability tree contains seven.

Another difference with probability trees is that the labelling LP_t of a path from the root to a descendant node t now determines an *extended configuration* $\mathbf{A}_{\mathbf{X}_I^t}$ for the variables in $\mathbf{X}_I^t, \mathbf{X}_I^t \subseteq \mathbf{X}_I$, rather than a standard configuration \mathbf{x}^t . This new concept is required in binary probability trees in order to express that a variable X_i in \mathbf{X}_I belongs to a subset of Ω_{X_i} , instead of stating that $X_i = x_i$. Extended configurations will be denoted with \mathbf{A} or $\mathbf{A}_{\mathbf{X}_I}$. Thus, an extended configuration $\mathbf{A}_{\mathbf{X}_I}$ defines a set of configurations $\mathbf{S}_{\mathbf{A}_{\mathbf{X}_I}}$ for \mathbf{X}_I , which is obtained with the Cartesian product of the subsets of states in $\mathbf{A}_{\mathbf{X}_I}$.

Example 22 For example, an extended configuration for the set of variables $\{A, B\}$ could be $\{\{a_3\}, \{b_1, b_2\}\}$. This means that A is a_3 and B can be b_1 or b_2 . Therefore, it determines the set of configurations $\{\{a_3, b_1\}, \{a_3, b_2\}\}$.

Example 23 For example, the extended configuration $\{X_1 = \{1, 2\}, X_2 = \{2\}\}$ corresponds to the labelling of the path from the root to the rightmost node in Fig. 3.36 (iii), corresponding to the probability value 0.5.

A^t denotes the *associated extended configuration* for node t .

3.7.3 Canonical models

Causal interaction models, called *canonical models* [22], were developed in order to simplify both the construction of Bayesian networks and probabilistic inference, as they reduce the number of parameters to be acquired, and so the size of the CPTs in the model. The most famous example is the *noisy-OR* model, where each *cause* X_i acts independently of the other causes to produce the effect Y . For each cause X_i , exists an *inhibitory mechanism* that can prevent the cause to produce the effect with a certain probability. A noisy-OR gate can be decomposed as shown in Fig. 3.37. This model requires only one parameter per parent to fully define the distribution.

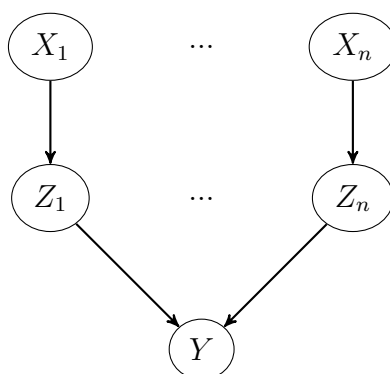


Figure 3.37: Noisy-OR gate for n causes.

This model has been extended in many ways:

- The noisy-MAX model [60] is a generalisation for *graded variables*¹ of the noisy-OR gate.
- The noisy-AND [61] differs from the noisy-OR gate in that the inhibitory mechanism for a single condition can make Y be false even if the other conditions are satisfied.

¹A graded variable X can be either absent or present with δ_X degrees of intensity.

3.7 Alternative representations of potentials

- The noisy-MIN model [62] is a generalization for graded variables of the noisy-AND gate.
- The noisy-ADD model, also known as noisy-addition [63], assumes that the effect of causes on the symptom is the addition of the effects caused by each cause independently.
- The recursive noisy-OR (RNOR) model [64], instead of allowing only probability parameters of the effect given each single-cause as the input, allows probability parameters of the effect given subsets of causes.
- NIN-AND trees, that stands for non-impeding noisy-AND trees [65], is a generalization of all the previous models, in the sense that can represent both *reinforcement and undermining*¹ in a recursive mixture, allowing multi-cause input as well.

¹When multiple causes are present, they may reinforce each other, that is, the more causes are active, the more likely the effect is to occur. Alternatively, multiple causes may as well undermine each other, making the effect less likely when more causes are present. This last interaction can only be modelled by NIN-AND trees among the causal models presented here.

Part II

Recursive Probability Trees

Chapter 4

Recursive Probability Trees

Recursive Probability Trees (RPTs) are a data structure that can be used for representing several types of potentials involved in Probabilistic Graphical Models (PGMs). The RPT structure improves the modelling capabilities of previous structures (like probability trees (PTs) or conditional probability tables (CPTs)). These capabilities can be exploited in order to gain savings in memory space and/or computation time during inference. This chapter describes the modelling capabilities of RPTs as well as how the basic operations required for making inference on Probabilistic Graphical Models operate on them. The performance of the inference process with RPTs is examined with some experiments using the Variable Elimination algorithm for Bayesian networks.

4.1 Motivation

The design of this data structure was directed to take advantage of different patterns that are present within probabilistic potentials, in order to build smaller representations of the probability distributions and speed up the inference process. In Chapter 3.7 we discussed different representations of probabilistic potentials, emphasizing the benefits of using tree-based structures (like probability trees) to represent potentials, as these structures allow the capture of patterns like context-specific independencies that help compact the representation. Also, it is easy to reduce the size of these structures by representing an approximation

of the distribution, strategy that is useful when the size of the potentials makes them unmanageable.

The problem is that frequently potentials present different patterns beyond context-specific independencies like proportionalities, as explained in Chapter 3, that are difficult to represent with existent data structures. Furthermore, as probabilistic graphical models develop, we find ourselves in the need of representing complex situations like sets of independencies that change for different contexts of subsets of variables (Bayesian multinets [5]), or probability distributions that are mixtures of smaller conditional distributions. Recursive Probability Trees are a tool conceived to be able to efficiently represent different types of patterns in the local structure of PGMs. Moreover, the basic operations to manage probabilistic potentials, as introduced in Chapter 3.5.1, are optimized to take advantage of the compacted representations in order to save computational effort.

4.2 Recursive Probability Trees

The chosen data structure to hold the potentials in PGMs has a direct impact on the performance of inference algorithms. The magnitude of this effect will depend on the probability distribution being represented and on the ability of the data structure to capture as many patterns present in the distribution as possible. We discussed in Chapter 3.7 some alternatives to potential representation other than probability tables, emphasizing the capabilities of probability trees to obtain (exact or approximate) compacted representations.

In this Chapter we present the *Recursive Probability Tree* (RPT) as a generalization of probability trees. RPTs are developed with the aim of enhancing PTs' flexibility and so they are able to represent different kinds of patterns that so far were out of the scope of probability trees.

RPTs as a generalization of PTs

An RPT \mathcal{RT} is an extension of a PT, and so it is a directed tree, to be traversed from root to leaves, where the nodes (both inner nodes and leaves) play different roles depending on their nature. In the simplest case, an RPT is equivalent to a

PT, where the inner nodes represent variables, and the leaf nodes are labelled by numbers. In the context of RPTs, we will call this type of inner nodes as *Split* nodes and this kind of leaves as *Value* nodes.

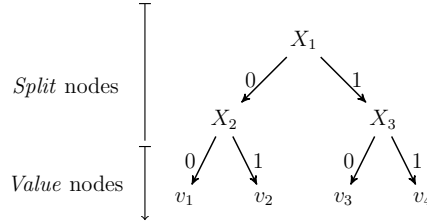


Figure 4.1: An RPT with only *Split* and *Value* nodes.

In Fig. 4.1 we can see an RPT for a potential of three binary variables. The three inner nodes are Split nodes, and they are labelled with the name of each variable. For each Split node, there are as many outgoing arcs as states the variable has, and each arc is labelled with the identifier of the correspondent state. Each children of a Split node is an RPT that encodes the corresponding potential to the context defined by the parent. In the leaves we have four Value nodes that are labelled with a single real number each.

In this example the RPT has the same shape as a PT and in general we can state that every potential that can be represented using a probability tree can be represented as an RPT of, at least, the same size as the original probability tree. This implies that RPTs can capture at least the same patterns as PTs do, like context-specific independencies.

Example 24 In Fig. 4.1 we can check that the RPT is representing a potential with context-specific independencies, as in the example it holds that the potential is independent of X_3 in the context $\{X_1 = 0\}$, and also it is independent of X_2 in the context $\{X_1 = 1\}$.

For a formal definition, consider an RPT \mathcal{RT} with $dom(\mathcal{RT}) = \mathbf{X}_{\mathbf{I}}$. A *Split* node is labelled with a variable $X_i \in \mathbf{X}_{\mathbf{I}}$, and it has outgoing arcs for each value $x_i \in \Omega_{X_i}$. Its children will be new RPTs encoding potentials defined over $\mathbf{X}_{\mathbf{J}} \subset \mathbf{X}_{\mathbf{I}}$. Let ϕ be the full potential represented by \mathcal{RT} , and ϕ_i be the potential corresponding to the child for $X_i = x_i$ then $\phi(\mathbf{x}_{\mathbf{J}}, x_i) = \phi_i(\mathbf{x}_{\mathbf{J}})$.

Example 25 *In the example presented in Fig. 4.1, we have a potential ϕ with $\text{dom}(\phi) = \{X_1, X_2, X_3\}$. The root of the RPT in the figure is a Split node labelled by variable X_1 , and as it is a binary variable, it has two outgoing arcs, labelled with each state that the variable can present: 0 and 1. Following the path where $X_1 = 0$, the children is an RPT that represents a potential ϕ_1 defined over the subset of variables $\mathbf{X}_J = \{X_2, X_3\}$ where $\mathbf{X}_J \subset \text{dom}(\phi)$. The left part of the RPT in Fig. 4.1 corresponds then to $\phi(\mathbf{x}_J, X_1 = 0)$ that is equivalent to saying $\phi_i(\mathbf{x}_J)$.*

In an RPT \mathcal{RT} we have to distinguish between its *theoretical domain*, that is the variables for which the tree is defined ($\text{dom}(\mathcal{RT})$) and the variables explicitly appearing in the nodes of the tree, which will be denoted as $\text{dom}^*(\mathcal{RT})$. In general, we will have that $\text{dom}^*(\mathcal{RT}) \subseteq \text{dom}(\mathcal{RT})$.

If in an RPT \mathcal{RT} the variables in $\text{dom}(\mathcal{RT})$ are not equal to the explicit variables $\text{dom}^*(\mathcal{RT})$, it will be assumed that the potential represented by the RPT is the potential defined on the variables $\text{dom}^*(\mathcal{RT})$ combined with an implicit potential defined for the variables $\text{dom}(\mathcal{RT}) \setminus \text{dom}^*(\mathcal{RT})$ being identically equal to 1, that is a potential with a value of 1 associated to each possible configuration of the variables in $\text{dom}(\mathcal{RT}) \setminus \text{dom}^*(\mathcal{RT})$.

This distinction will be relevant when implementing operations with RPTs, as the fact that $\text{dom}(\mathcal{RT}) \neq \text{dom}^*(\mathcal{RT})$ means that the potential represented with \mathcal{RT} is independent on the variables in $\text{dom}(\mathcal{RT}) \setminus \text{dom}^*(\mathcal{RT})$ and sometimes this information can be relevant.

Example 26 *For example, as we mentioned before, in Fig. 4.1 the RPT rooted with a Split node of X_2 corresponds to a potential ϕ_1 with $\text{dom}(\phi_1) = \{X_2, X_3\}$ and $\text{dom}^*(\phi_1) = \{X_2\}$.*

Factorisations with RPTs

In Chapter 3.7.1.3 we discussed proportionalities, a pattern usually present within probability distributions. Using probability trees, we saw how the representation of proportionalities implied the storage of a list of smaller probability trees that, when combined, represented the original potential. Besides the problems of having to handle the lists specifically when incorporating this kind of divisions in

4.2 Recursive Probability Trees

algorithms that work with probability trees, sometimes it is impossible to reduce the size of the factors in the factorisation. Consider the potential represented in Fig. 4.2 as a probability tree. This potential presents proportionalities, as the branch correspondent to the context $\{X_1 = 0, X_2 = 1\}$ is twice the branch correspondent to the context $\{X_1 = 0, X_2 = 0\}$. However, the branch corresponding to the context $\{X_1 = 0, X_2 = 2\}$ does not present any proportional factor with respect to its siblings. If we factorise it using probability trees, we are not able to lower the complexity of both factors, as seen in Fig. 4.3 where we still have a factor with three variables, and furthermore, we need to increase the number of values to represent the potential to ten, when the original only needs nine values.

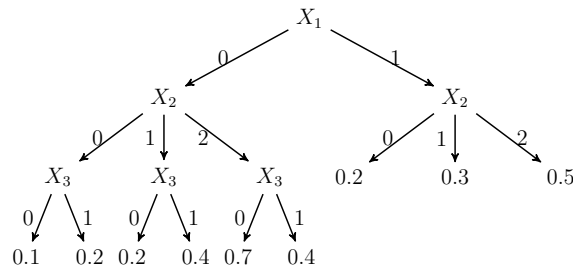


Figure 4.2: Probability trees with proportional subtrees.

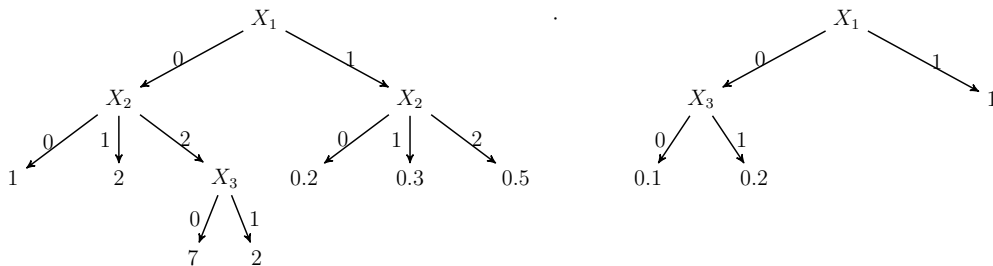


Figure 4.3: Factorisation of the probability tree in Fig. 4.2 using PTs.

Recursive Probability Trees propose to include the factorisations within the data structure by incorporating a type of inner node which mission is to list together all the factors. Therefore, a *List* node represents a multiplicative factorisation by listing all the factors making up the division. If a *List* node stores a factorisation of k factors of a potential ϕ defined on $\mathbf{X}_{\mathbf{J}}$, and every factor i (that

is an RPT as well) encodes a potential ϕ_i for a subset of variables $\mathbf{X}_{J_i} \subseteq \mathbf{X}_J$, then ϕ can be obtained as $\prod_{i=1}^k \phi_i(\mathbf{X}_{J_i})$.

Example 27 For example, in Fig. 4.4(i) we can observe a probability tree that contains proportional subtrees, as the subtree for the context $\{X_1 = 0, X_2 = 1\}$ is twice the subtree correspondent to the context $\{X_1 = 0, X_2 = 0\}$. This can be represented using RPTs in a single structure using a List node under the context $\{X_1 = 0\}$, as shown in Fig. 4.4(ii).

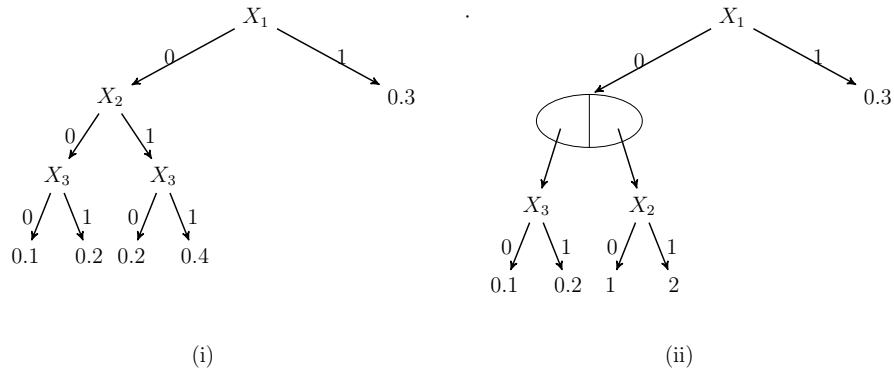


Figure 4.4: Simple probability tree with proportionalities and its factorisation as an RPT.

Example 28 For the pattern shown in Fig. 4.2, RPTs can offer a solution as the structure in Fig. 4.5. In this RPT, the factorisation is incorporated in the structure under the context $X_1 = 0$, and using the same number of parameters as the original representation.

RPTs for Bayesian networks

When necessary, RPTs will include a fourth type of node denominated *Potential* node. This is a leaf node and its purpose is to encapsulate a full potential within the leaf in an internal structure. This internal structure usually will not be an RPT, but a probability tree or a probability table instead. In fact, as long as the internal structure of a Potential node supports the basic operations on potentials as explained in Chapter 3.5.1, it is accepted within the RPT representation.

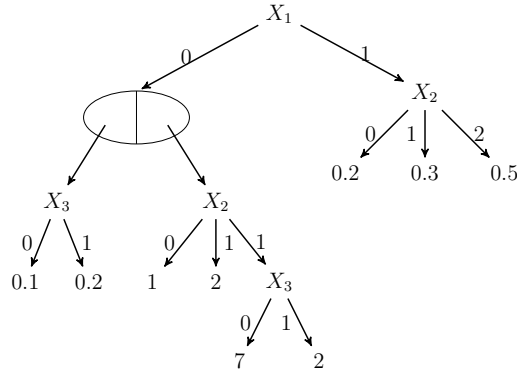


Figure 4.5: Factorisation of the probability tree in Fig. 4.2 as an RPT.

In summary, an RPT can have four kind of nodes in total: *Split* or *List* nodes as inner nodes, and *Value* or *Potential* nodes as leaves. We can combine them in very different ways in order to find the structure that best fits the potential to be represented, making RPTs an extremely flexible framework to work with. Fig. 4.6 shows a single CPT (i) and three alternative RPTs representations: (ii) an RPT with a single *Potential* node as root; (iii) RPT with a *Split* node for X and two *Value* nodes (this representation is equivalent to the corresponding PT); and (iv) RPT with a *List* node encoding the trivial factorisation $1 \cdot \phi(X)$.

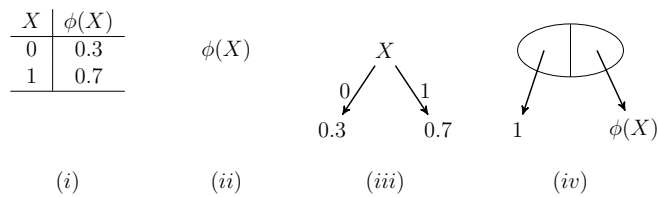


Figure 4.6: Different RPTs for the probability distribution in a).

An RPT is able to represent a full model like a Bayesian network. The more straightforward way of representing a BN would be to join with a List node all the conditional probability distributions defined by the chain rule for Bayesian networks. Every factor could be represented independently looking for patterns within it.

Example 29 For example, consider the BN defined in Fig. 4.7 (i). We can resort to the factorisation of its joint probability distribution by means of a set of conditional probability distributions to build an RPT such as the one presented in Fig. 4.7 (ii).

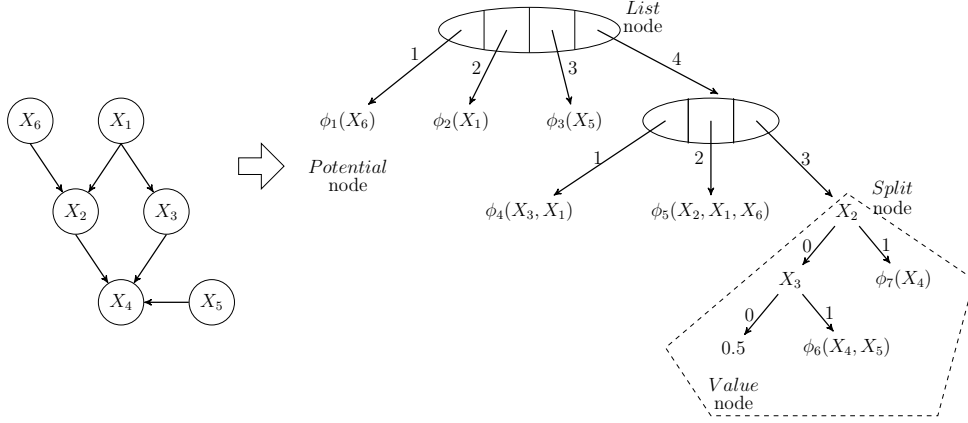


Figure 4.7: An RPT representing a full Bayesian network.

In the figure, the RPT has a List node as root that contains a child for every conditional probability distribution defined by the BN. Note that the three leftmost children are Potential nodes for the marginal potentials: $\phi_1(X_6)$, $\phi_2(X_1)$ and $\phi_3(X_5)$. The fourth child encodes the factors $\phi_4(X_3, X_1)$, $\phi_5(X_2, X_1, X_6)$ and $\phi_7(X_4, X_2, X_3, X_5)$ through a new List node. The first two factors are represented as Potential nodes and the third one with a Split node for capturing some context-specific independencies imposed by X_2 and X_3 (see the branch enclosed with the dashed line). These independencies are represented with Split nodes for X_2 and X_3 (both of them are binary variables). The right child for X_2 contains a Potential node for $\phi_7(X_4)$. The left one contains a Split node for X_3 : if $X_3 = 0$ then the potential does not depend on X_4 and X_5 and a Value node is enough (with 0.5 as value). The potential corresponding to $X_3 = 1$ is contained into a Potential node.

Formal definition of RPTs

Formally, it can be stated that an RPT (\mathcal{RT}) defined over a set of variables \mathbf{X}_I represents the potential $\phi_{\mathcal{RT}}(\mathbf{X}_I) : \Omega_{\mathbf{X}_I} \rightarrow \mathbb{R}_0^+$ if for each $\mathbf{x}_I \in \Omega_{\mathbf{X}_I}$ the

value $\phi_{\mathcal{RT}}(\mathbf{x}_{\mathbf{I}})$ is the number obtained with the recursive procedure explained in Algorithm 7.

The procedure starts from the root of the RPT, traversing recursively the full structure to the leaves. It works by applying a different action depending on the kind of node. If *root* is a: *Value* node, just returns the corresponding value (lines 6 and 7 of Alg. 7.); *Potential* node: gets the value for the selected configuration \mathbf{x} (lines 8 to 10 of Alg. 7.); *Split* node: the procedure is recursively called to the child consistent with the given configuration (lines 11 to 15 of Alg. 7.); *List* node: multiplies the results obtained from new recursive calls for every child (lines 16 to 18 of Alg. 7.).

```

1  probValue( $\mathcal{RT}, \mathbf{x}_{\mathbf{I}}$ )
   Input:  $\mathcal{RT}$ : an RPT defined on  $\mathbf{X}_{\mathbf{I}}$ ,  $\mathbf{x}_{\mathbf{I}}$ : a configuration;
   Output:  $P_{\mathcal{RT}}(\mathbf{x}_{\mathbf{I}})$ : a value;
2  begin
3      $root \leftarrow$  root node of  $\mathcal{RT}$ ;
4      $type \leftarrow$  type of  $root$  (Value, Potential, Split or List)
5     switch  $type$  do
6         case Value
7             return label of  $root$ 
8         case Potential
9             Let  $P(\mathbf{X}_{\mathbf{J}} \subseteq \mathbf{X}_{\mathbf{I}}) \leftarrow$  be the potential labelling the  $root$ ;
10            return  $P(\mathbf{x}_{\mathbf{I}})$ 
11        case Split
12            Let  $X_i$  be the variable labelling the  $root$ ;
13            Let  $x_i$  be  $\mathbf{x}_{\mathbf{I}}^{\downarrow\{X_i\}}$ ;
14             $ch_i(\mathcal{RT}) \leftarrow$  child of  $root$  for  $x_i$  value ;
15            return probValue( $ch_i(\mathcal{RT}), \mathbf{x}_{\mathbf{I}}$ ) ;
16        case List
17             $ch_1(\mathcal{RT}), ch_2(\mathcal{RT}) \dots ch_n(\mathcal{RT})$  children of  $root$ ;
18            return  $\prod_{i=1}^n \mathbf{probValue}(ch_i(\mathcal{RT}), \mathbf{x}_{\mathbf{I}})$ 
19        endsw
20    endsw
21 end

```

Algorithm 7: Algorithm to compute the correspondent probability value from an RPT given a configuration of its variables.

Example 30 For example, consider the RPT in Fig. 4.8. We want to recover the value correspondent to a configuration $\mathbf{x}_I = \{X_1 = 1, X_2 = 0, X_3 = 1, X_4 = 0\}$ from the RPT using Algorithm 7. We start from the root of the tree (line 3 of Alg. 7), that is a Split node of variable X_1 (line 4 of Alg. 7). The switch structures leads us to the specific functionality for Split nodes (line 11 of Alg. 7). The projection¹ of configuration \mathbf{x}_I to variable X_1 (line 13 of Alg. 7) gives us the value 1, so we apply Alg. 7 recursively to the child corresponding to the right branch (lines 14 and 15 of Alg. 7). The new root is a Value node, so the algorithm will return the value that labels it (line 7 of Alg. 7). In this example, the value retrieved by the algorithm is 0.7, that is the correspondent value in the potential for configuration $\{X_1 = 1, X_2 = 0, X_3 = 1, X_4 = 0\}$.

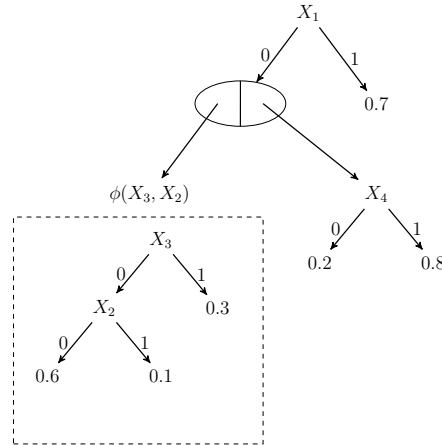


Figure 4.8: An RPT representing a potential of three variables.

Example 31 Let now consider the configuration $\mathbf{x}_I = \{X_1 = 0, X_2 = 1, X_3 = 1, X_4 = 0\}$. In this case, the algorithm begins the same as in the last example, but now the projection of configuration \mathbf{x}_I to variable X_1 (line 13 of Alg. 7) gives us the value 0, so we apply the algorithm recursively to the child corresponding to the left branch (line 14 and 15 of Alg. 7). The child is a List node, so we need to recursively apply the algorithm to every one of the List node's children, and

¹The projection of a configuration to a variable $\mathbf{x}_I^{\downarrow\{X_i\}}$ corresponds to the retrieval of the state defined in a configuration \mathbf{x}_I for a given variable X_i . We will denote it as the identifier of the configuration with a superindex that specifies the variable to project.

then multiply the individual obtained values (lines 17 and 18 of Alg. 7). Working from left to right, the first child is a Potential node, and the potential that it encapsulates is disclosed within the dashed line in Fig. 4.8. As we can appreciate, the structure that holds the potential within the potential node is a probability tree. The algorithm delegates the retrieval of the correspondent value to the probability tree (lines 9 and 10 of Alg. 7), that will return 0.3 as the leaf consistent with \mathbf{x}_I . The second child of the list is a Split node, so we proceed in the same way as in the last example, retrieving the value 0.2. The final result of the algorithm will be the multiplication of the obtained values: $0.3 \cdot 0.2 = 0.06$, that is the correspondent value in the potential for configuration $\{X_1 = 0, X_2 = 1, X_3 = 1, X_4 = 0\}$.

4.3 Expressiveness of RPTs

As it was mentioned before, RPTs can be considered as a general representation for potentials involved in PGMs. In this section we analyse in detail some patterns for which RPTs provide an efficient representation, and how they take advantage of the local structure, achieving more compact representations than probability trees and probability tables.

4.3.1 Proportional values

Proportional values are sometimes present within probability distributions. An example of this was given in Section 3.7.1.3, where we dealt with the presence of *proportional subtrees* in probability trees by factorising the original structure into two smaller trees. This solution, however, is somehow limited and requires the specific managing of the lists of factors within the inference algorithms in order to take advantage of the factorisations. Recursive probability trees offer a more flexible framework, being able to represent and manage the factorisations within the structure itself.

Consider the potential $\phi(X_1, X_2)$ encoded as a CPT defined in Fig. 4.9. This potential presents proportional values, as the values consistent with the configuration $\{X_1 = 1\}$ are 4 times the values correspondent with the configuration

$\{X_1 = 0\}$, for each state of X_2 respectively. Therefore, the values of ϕ consistent with the configuration $\{X_1 = 0\}$ can be considered as a *base* potential for a factorisation.

If we represent this potential as a probability tree, we do not obtain a more compact representation unless we build a product of two factors. The same procedure can be followed to build an RPT, as shown in Fig. 4.10, with the added advantage that RPTs are optimized to work with this factorised representations, so the efficiency gain can be increased. Observe that the RPT stores a reduced number of parameters than the CPT.

X_1	X_2	$\phi(X_1, X_2)$
0	0	0.025
0	1	0.075
0	2	0.075
0	3	0.025
1	0	0.1
1	1	0.3
1	2	0.3
1	3	0.1

Figure 4.9: Potential with proportional values.

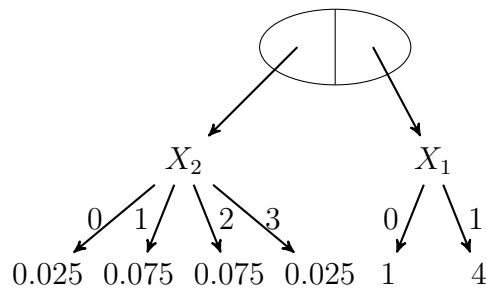


Figure 4.10: RPT encoding the potential with proportional values described in Fig. 4.9.

4.3.2 Context-specific independencies

The concept of context-specific independency was introduced in Section 3.7, where we discussed the benefits of tree-based representations when representing the potentials in a PGM. Fig. 4.11 shows an example where a potential ϕ with $dom(\phi) = \{X_1, X_2, X_3\}$ some values sometimes do not depend on the whole set of variables of the domain, presenting context-specific independencies.

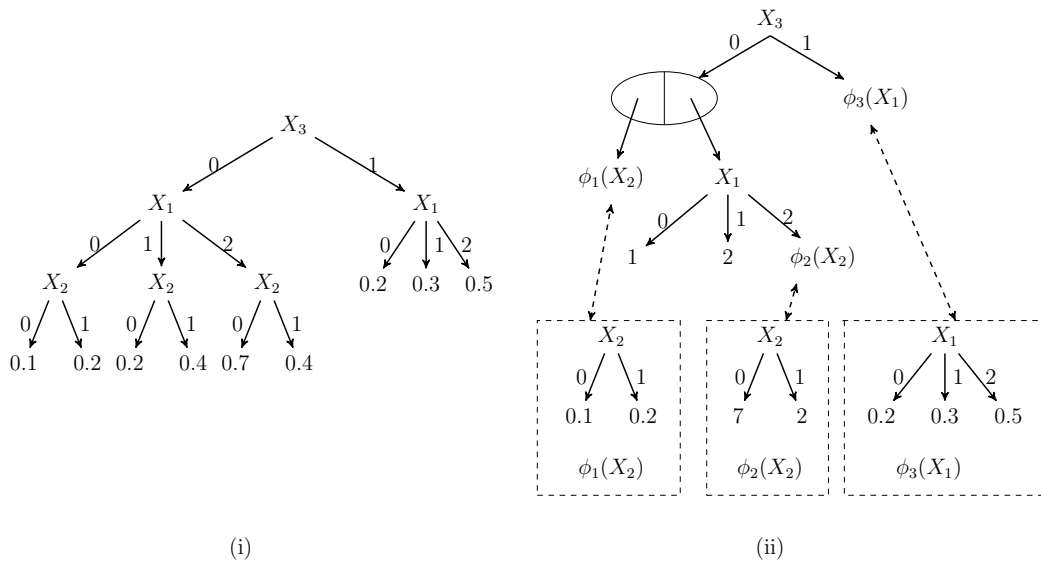


Figure 4.11: Potential with context-specific independencies and proportionalities.

The left part (i) in Fig. 4.11 shows a probability tree representing the potential $\phi(X_1|X_2, X_3)$ where some of the values (right branch for $X_3 = 1$) do not depend on X_2 , that is to say that the potential ϕ is independent of X_2 (it only depends on X_1) in the context $\{X_3 = 1\}$. Moreover, this potential includes proportional values as well. The configuration given by $X_3 = 0, X_1 = 0$ points to a potential whose values can be used to generate the values for $X_3 = 0, X_1 = 1$ (multiplying by 2) and for $X_3 = 0, X_1 = 2$ (respectively multiplying by 7 and 2). All of these features are represented with the RPT in Fig. 4.11 (right part, (ii)).

4.3.3 Multinets

An advantage of Bayesian networks is that they can specify dependencies only when necessary, leading to a significant reduction in the cost of inference. Bayesian multinets [5] further generalize Bayesian networks and can further reduce computation. A multinet can be thought of as a network where edges can appear or disappear depending on the values of certain nodes in the graph. Consider a network with four nodes X_1, X_2, X_3, X_4 where the conditional independencies among these variables depend on the values of X_4 , a binary variable. In fact this conveys the need to consider two different Bayesian networks, one for each state of X_4 , (see Fig. 4.12 (i)). The complete multinet can be represented with an RPT, as shown in Fig. 4.12 (ii), where the root node is a Split node that defines the context for the different groups of independencies, and each child is a List node that represents the multiplicative factorisation of the conditional probability distributions present for each context.

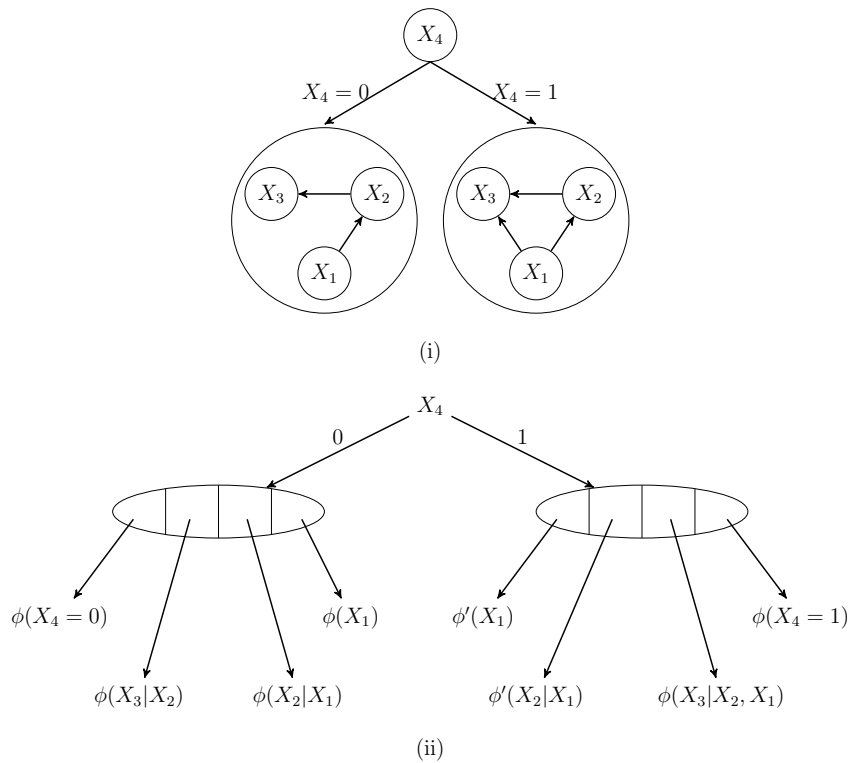


Figure 4.12: Bayesian multinet and RPT representation.

Therefore, any inference algorithm that works with multinets could be adapted to take advantage of the RPT representation.

4.3.4 Mixtures of conditional distributions

Imagine a variable X_4 with X_1, X_2 , and X_3 as parents. Let us assume that a probability distribution for these variables is given by a convex combination of two terms: on one hand, a conditional distribution of X_4 given X_1, X_2 ; and on the other, a conditional distribution of X_4 given X_2, X_3 . That is, the considered probability distribution is defined as:

$$P(X_4|X_1, X_2, X_3) = \alpha P(X_4|X_1, X_2) + (1 - \alpha)P(X_4|X_2, X_3).$$

Recursive probability trees can easily represent multiplicative factorisations, but not additions such as the one defined in the mixture above. However, it is possible to build an RPT that represent the mixture of conditional distributions proceeding in the following way: we define an auxiliary variable A with as many states as terms the mixture has. In the example considered here, the mixture only has two terms, so A would be a binary variable. The idea now is to express the mixture as a combination of the states of A , such as if we marginalise out A from the final model, we obtain the addition of the terms in the mixture.

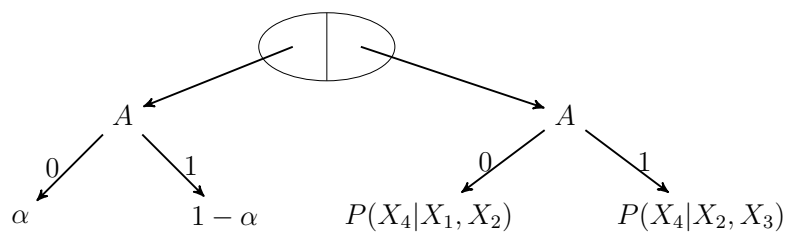


Figure 4.13: RPT representation of a mixture of conditional distributions using an auxiliary variable A .

Fig. 4.13 shows the RPT encoding the mixture of conditional probability distributions explained here, where the root is a list node that encodes the mul-

tiplication of two factors where A labels both roots. In the first factor, at the left side of the figure, A has two Value nodes as children with the weights of the convex combination, α and $1 - \alpha$. In the right part of the figure, A has two Potential nodes containing both conditional probability distributions present in the division: $P(X_4|X_1, X_2)$ and $P(X_4|X_2, X_3)$ respectively. As we will explain in the next section, the marginalisation over RPTs, that runs from root to leaves, first solves the List nodes that are up the tree by multiplying the factors where the variable to marginalise out is involved. This ensures the correctness of the RPT representation of mixtures of conditional distributions explained here, as the states of the auxiliary variable A will be added only after multiplying both factors.

4.4 Operations with RPTs

The basic operations on potentials: *restriction*, *combination* and *marginalisation*, that were described in Sec. 3.5.1 can be adapted to be supported by RPTs, following a similar procedure as the described in Sec. 3.7.1.1 over probability trees. As RPTs are a generalisation of PTs, the operations described in Sec. 3.7.1.1 are extended to deal with the multiplicative factorisations introduced within the RPTs by the List nodes.

In this section we describe how to carry out these operations directly on the RPT data structure. The operations over Potential nodes will depend on the particular data structure used to represent the distribution within it, for example, if two leaves are Potential nodes represented by probability trees, then it is possible to combine them by multiplication using the operations described in Sec. 3.7.1.1 [2].

In addition, sometimes it is also needed to transform a potential represented with an RPT into a probability distribution or a conditional probability distribution. This can be achieved through the *normalisation* and *conditional normalisation* operations.

4.4.1 Restriction

Let \mathcal{RT} be an RPT and $\mathbf{x}_{\mathbf{J}}$ a configuration for the variables in $\mathbf{X}_{\mathbf{J}}$. The restriction operation of $\mathcal{RT}^{R(\mathbf{x}_{\mathbf{J}})}$ (\mathcal{RT} restricted to $\mathbf{x}_{\mathbf{J}}$) will be performed recursively from root to leaf nodes acting on nodes according to their type. In the simplest case, that is when the RPT only contains Split and Value nodes, the operation is applied exactly as explained in Sec. 3.7.1.1 when working with probability trees: Value nodes will remain unaltered, while in Split nodes the operation will be transmitted to the children if the variable labelling the Split node is not in $\mathbf{X}_{\mathbf{J}}$, in any other case the leaf representing the value contained in $\mathbf{x}_{\mathbf{J}}$ will be kept (and promoted to the place of its parent node) and the rest will be removed.

The addition of the two extra types of nodes present in RPTs increases the complexity of the operation. Potential nodes will produce the restriction of their potentials if needed (their domains include any of the variables in $\mathbf{X}_{\mathbf{J}}$). Therefore, the data structure used to represent the potentials within the Potential nodes should support the restriction operation. Finally, List nodes will transmit the operation to their children, continuing recursively with the procedure.

Example 32 *The pseudocode for this procedure is shown in Algorithm 8. To illustrate its workflow, this operation will be applied to the RPT presented in Fig. 4.14, where part (i) represents the BN and the set of potentials involved and the RPT representation is below (ii). Suppose we want to compute the value of the potential restricted to the configuration $\mathbf{x}_{\mathbf{J}} = \{X_1 = 0, X_2 = 1, X_3 = 0\}$.*

4.4 Operations with RPTs

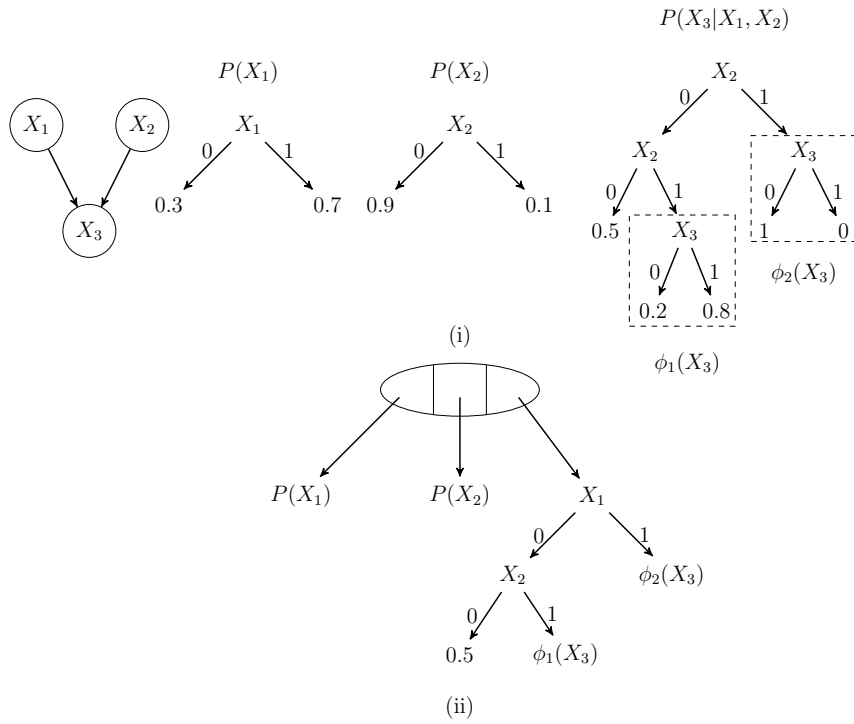


Figure 4.14: RPT encoding a Bayesian network

```

1  restrict( $\mathcal{RT}, \mathbf{x}_J$ )
   Input: An RPT  $\mathcal{RT}$ , a configuration of variables  $\mathbf{X}_J = \mathbf{x}_J$ 
   Output: An RPT  $\mathcal{RT}^{R(\mathbf{x}_J)}$ 
2  begin
3      Let root be the root of  $\mathcal{RT}$ ;
4      if root is a Value node labelled with  $r$  then return  $r$  ;
5      else if root is a Potential node labelled with  $P$  then return  $P^{R(\mathbf{x}_J)}$  ;
6      else if root is a Split node labelled with  $X_i$  then
7          if  $X_i \in \mathbf{X}_J$  then
8              Let  $ch_i(\mathcal{RT})$  be the child of root corresponding to the value of  $X_i$  in  $\mathbf{x}_J$ ;
9              return restrict( $ch_i(\mathcal{RT}), \mathbf{x}_J$ );
10         else
11             Make a new RPT  $\mathcal{RT}'$  with  $X_i$  as root;
12             foreach child of the root of  $\mathcal{RT}$ ,  $ch_i(\mathcal{RT})$  do
13                  $\mathcal{RT}'_i \leftarrow$  restrict( $ch_i(\mathcal{RT}), \mathbf{x}_J$ );
14                 Set  $\mathcal{RT}'_i$  as  $i$ -th child of  $\mathcal{RT}'$  root;
15             end
16             return  $\mathcal{RT}'$ ;
17         end
18     end
19     else if root is a List node then
20         Make a new RPT  $\mathcal{RT}'$  with a List node as root;
21         foreach child of the root of  $\mathcal{RT}$ ,  $ch_i(\mathcal{RT})$  do
22              $\mathcal{RT}'_i \leftarrow$  restrict( $ch_i(\mathcal{RT}), \mathbf{x}_J$ );
23             Set  $\mathcal{RT}'_i$  as  $i$ -th child of  $\mathcal{RT}'$  root;
24         end
25         return  $\mathcal{RT}'$ ;
26     end
27 end

```

Algorithm 8: Algorithm to restrict an RPT to a given configuration of variables.

When the restriction operation $R(\mathbf{x}_J)$ is invoked on the root node of the RPT, as shown in Fig. 4.15 (i), as the root is a List node, the operation is propagated downwards to its children (lines 19 to 26 of Alg. 8). Every children is restricted independently of the others: the first and second children are Potential nodes that are represented internally as probability trees, so after restricting them (line 5 of Alg. 8) we obtain values that are stored in Value nodes within the RPT, as shown

in Fig. 4.15 (ii). The last child is a Split node of a variable that is contained in $\mathbf{x}_{\mathbf{J}}$, so the operation is propagated in the consistent branch (lines 7 to 9 of Alg. 8). We reach another Split node whose variable is also in $\mathbf{x}_{\mathbf{J}}$, so again the operation is propagated in the consistent branch, as shown in Fig. 4.16 (i). At this point, we reach a Potential node that also includes a variable present in the restricting configuration, so applying the restriction operation over the node returns a value that is enclosed in a Value node, as pictured in Fig. 4.16 (ii).

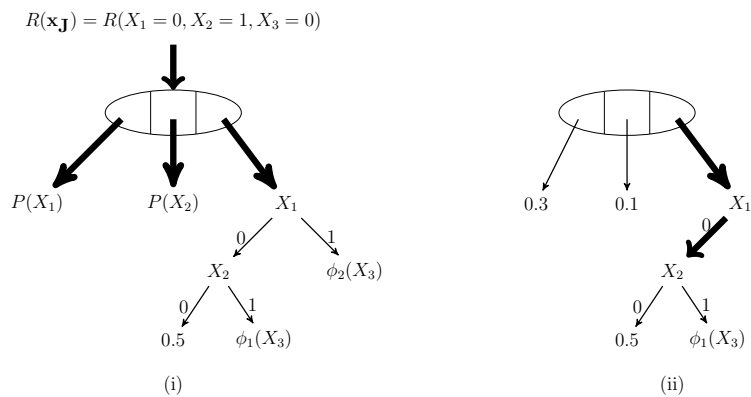


Figure 4.15: RPT restricted to a configuration, operation applied to the List node at the root.

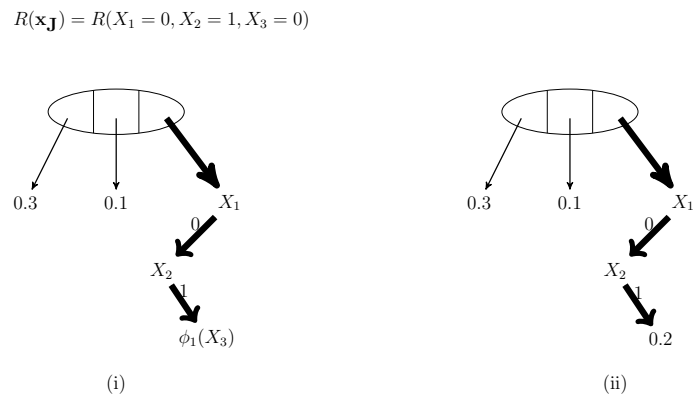


Figure 4.16: RPT restricted to a configuration, operation applied to a Split node.

Finally, the structure is compacted into the RPT shown in Fig. 4.17: a List node containing three values. As the RPT assumes a multiplicative factorisation,

the probability value for the configuration will be given by multiplying these values: 0.006.

$$R(\mathbf{x}_J) = R(X_1 = 0, X_2 = 1, X_3 = 0)$$

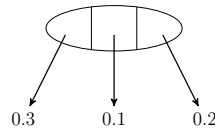


Figure 4.17: Result of restricting an RPT to a configuration using Alg. 8.

Observe that this operation is performed recursively from root to leaves, so the time consumed by the algorithm will be linear on the number of nodes of the RPT.

4.4.2 Combination

The possibility of expressing multiplicative factorisations within RPTs by means of the List nodes makes the *combination* a very simple operation, as we will merely join the factors using a List node that will root the resultant RPT. This idea is described in Algorithm 9. Note that the algorithm is formulated in a general way, so that it is designed for combining two RPTs (lines 12 to 14 of Alg. 9) but also for combining an RPT with another kind of potential and an RPT with a real value. In these last two cases, when the second factor is a constant, it is stored in a *Value* node (lines 4 to 7 of Alg. 9), whilst when the factor is a potential, it is enclosed within a *Potential* node (lines 8 to 11 of Alg. 9). These transformations are done prior to storing the factors as children of the *List* node (lines 5 and 9 of Alg. 9, respectively).

Example 33 A very simple example of the application of this operation is shown in Fig. 4.18. In both parts we are combining a simple RPT \mathcal{RT} (that contains just a *Potential* node) with a value v (i) and with another potential $P(X_1)$ that could be an RPT with a *Potential* node as a root, or simply a potential stored in any

```

1 combine( $\mathcal{RT}$ ,  $f$ )
   Input: An RPT  $\mathcal{RT}$ , a factor  $f$  (another RPT, a potential or a value).
   Output: The result of combining  $\mathcal{RT}$  and  $f$ .
2 begin
3   | Make a new RPT  $\mathcal{RT}'$  with a List node  $\mathcal{L}$  as root;
4   | Put  $\mathcal{RT}$  as child of  $\mathcal{RT}'$ ;
5   | if  $f$  is a numeric value then
6   |   | Make a new Value node labelled with  $f$ ;
7   |   | Put the Value node as child of  $\mathcal{L}$ ;
8   | end
9   | if  $f$  is a potential then
10  |   | Make a new Potential node labelled with  $f$ ;
11  |   | Put the Potential node as child of  $\mathcal{L}$ ;
12  | end
13  | if  $f$  is an RPT then
14  |   | Put RPT as child of  $\mathcal{L}$ ;
15  | end
16  | return  $\mathcal{RT}'$ ;
17 end

```

Algorithm 9: Algorithm to combine two RPTs

other data structure, like a probability tree. After applying Alg. 9, in Fig. 4.18 (i) we obtain an *RPT* with a *List* node as a root (lines 5 to 8 of Alg. 9), that has two children: one is \mathcal{RT} , and the other is a *Value* node that contains v . Again, in Fig. 4.18 (ii), we obtain an *RPT* with a *List* node as a root (lines 9 to 12 of Alg. 9), that has two children: one is \mathcal{RT} , and the other is a *Potential* node that contains $P(X_1)$.

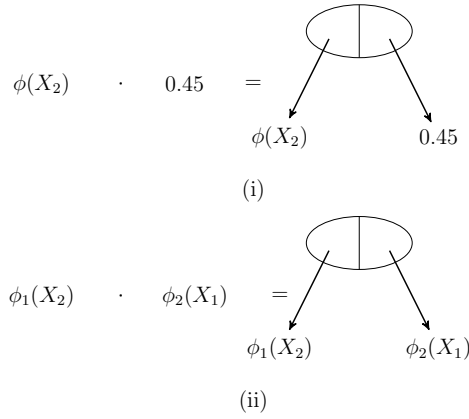


Figure 4.18: Combination of potentials with RPTs.

Example 34 *The last part of Alg. 9 (lines 13 to 15) deals with the combination of two RPTs. An example is illustrated in Fig. 4.19, where the result of the operation is again an RPT with a List node as a root, that has as children both RPTs.*

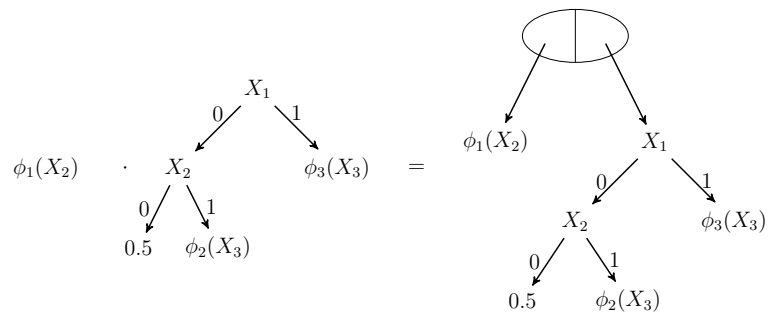


Figure 4.19: Combination of potentials with RPTs.

An important feature of the combination process as described in Alg. 9 is that it does not really require the actual computation of any product of numbers. In this sense, RPTs are compatible with inference schemes based on lazy propagation [11; 12].

4.4.3 Marginalisation

The *marginalisation* of a potential ϕ to a set of variables \mathbf{X}_J consists, as explained in Chapter 3.5.1, of summing out all the variables not in \mathbf{X}_J . This is noted as $\phi^{\downarrow \mathbf{X}_J}$. Again this operation can be performed directly over RPTs, but this operation is not trivial. For summing out a variable from an RPT we need to define first the *multiplication* and the *addition* of two RPTs.

Multiplying RPTs

The multiplication of RPTs is explained in Alg. 10. The difference between the *combination* of RPTs as explained in Section 4.4.2 through Alg. 9 and the multiplication of RPTs as defined in Alg. 10 is that when combining RPTs it may not be necessary to actually perform the numerical computations, being sufficient to store the factors in a List node to be multiplied afterwards. In Alg. 10, even though a grouping of factors is performed, the idea is to divide the RPT into smaller factors consistent with each other until we encounter two factors that can be multiplied, i.e. Value or Potential nodes.

The algorithm takes two RPTs RT_1 and RT_2 and acts in accordance to the type of their roots, $root_1$ and $root_2$ respectively. If either root is a List node (lines 3 to 19 of Alg. 10), the algorithm will return a new RPT RT' with a List node at the root that will put together the factors of $root_1$ and $root_2$. If both roots were List nodes, then the RT' will have as children all the factors of $root_1$ and $root_2$ together (lines 3 to 8 of Alg. 10). If either $root_1$ or $root_2$ is not a List node, then the root of RT' will father all the factors of the RPT rooted by a List node, and the root of the other RPT (lines 9 to 19 of Alg. 10).

Example 35 Consider Fig. 4.20, where Alg. 10 is performed upon two RPTs, one rooted by a List node and the second rooted with a Split. The result of the operation, as shown in the right part of the figure, would be an RPT with a List node at the root, and fathering all the factors of the first RPT plus the whole second RPT.

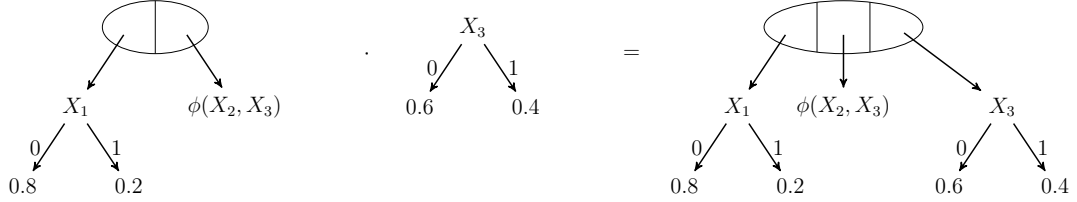


Figure 4.20: Multiplication of two RPTs (I).

Example 36 Consider now Fig. 4.21 where Alg. 10 is performed upon two RPTs, both rooted by a List node. The result of the operation, as shown in the right part of the figure, would be an RPT with a List node at the root, and fathering all the factors of both RPTs.

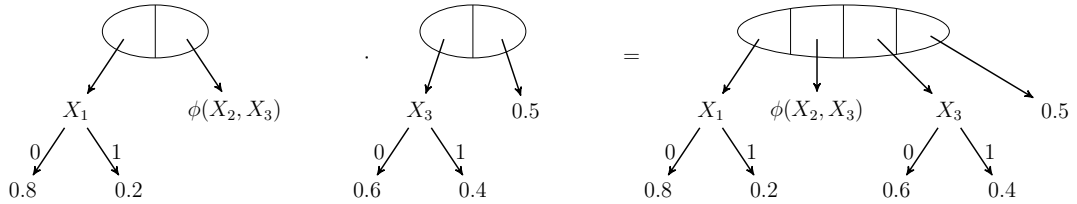


Figure 4.21: Multiplication of two RPTs (II).

If $root_1$ is a Split node labelled by a variable X_i , then the root of RT' will be a Split node of X_i (line 21 of Alg. 10) with a List node for each of its children. Each List node i will contain two factors: one will be the child correspondent to the i -th state of X_i in $root_1$, and the second will be $RT_2^{R(X_i=x_i)}$, that is the second RPT restricted to the correspondent context of the variable X_i (line 24 of Alg. 10). In the case that $root_1$ is not a Split node but $root_2$ is, the procedure is the same (lines 29 to 36 of Alg. 10).

Example 37 The next example is illustrated with Fig. 4.22. Now we are multiplying two RPTs RT_1 and RT_2 that both have Split nodes as roots. The resultant RPT has a Split node of the variable X_3 that labels the first multiplicand, RT_1 . For $X_3 = 0$, we have a List node in the resultant RPT that contains two factors:

the part of RT_1 consistent with $\{X_3 = 0\}$, that is a Value node labelled by 0.6, and the part of RT_2 consistent with the same configuration, that is a Split node of X_1 with two Value nodes as children. Note that this structure is the result of applying the restrict operation, as explained in Section 4.4.1, to RT_2 for the configuration $\{X_3 = 0\}$. The second child of the resultant RPT's root is the same, but this time using the configuration $\{X_3 = 1\}$.

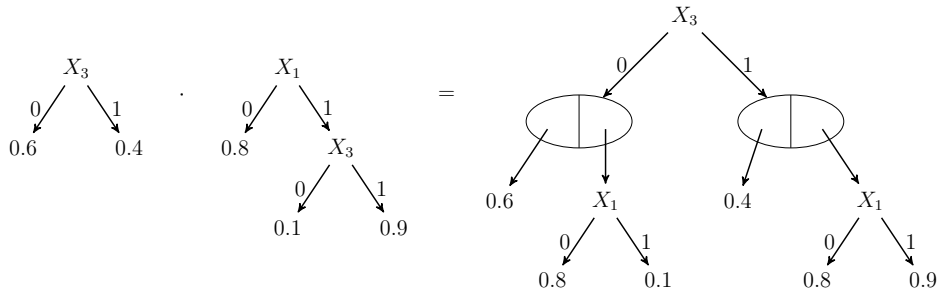


Figure 4.22: Multiplication of two RPTs (III).

Note that at this point, $root1$ and $root2$ can either be Value or Potential nodes, as the possibility of either of them being a List or a Split node has already been covered by the algorithm. Now the multiplication is a direct operation (lines 38 and 39 of Alg. 10), if both $root1$ and $root2$ are Value nodes, the result will be a Value node labelled by the result of the multiplication of the labels of both roots. If either $root1$ or $root2$, or both are Potential nodes, then the result will be a Potential node labelled by the product of both potentials. In the case that both roots were Potential nodes, then we have to be careful because it may happen that the data structure that represents the potential in $root_1$ is not the same as the data structure used within $root_2$. We have handled this scenario by making the resultant Potential node always contain a probability tree, but other approaches could be developed for specific data structures.

```

1 multiply( $\mathcal{R}\mathcal{T}_1, \mathcal{R}\mathcal{T}_2$ )
   Input: Two RPTs  $\mathcal{R}\mathcal{T}_1$  with root  $root_1$  and  $\mathcal{R}\mathcal{T}_2$  with root  $root_2$ 
   Output: The multiplication of  $\mathcal{R}\mathcal{T}_1$  and  $\mathcal{R}\mathcal{T}_2$ 
2 begin
3   if  $root_1$  is a List node then
4     Make a new RPT  $\mathcal{R}\mathcal{T}$  with a List node as root;
5     Append all the children of  $root_1$  as children of the root of  $\mathcal{R}\mathcal{T}$ ;
6     if  $root_2$  is a List node then
7       Put all the children of  $root_2$  as children of the root of  $\mathcal{R}\mathcal{T}$ ;
8     end
9     else
10      Append the factor rooted by  $root_2$  as a child of the root of  $\mathcal{R}\mathcal{T}$ ;
11    end
12    return  $\mathcal{R}\mathcal{T}$ ;
13  end
14  else if  $root_2$  is a List node then
15    Let  $\mathcal{R}\mathcal{T}$  be a new RPT with a List node as root;
16    Append all the children of  $root_2$  as children of the root of  $\mathcal{R}\mathcal{T}$ ;
17    Append the factor rooted by  $root_1$  as a child of the root of  $\mathcal{R}\mathcal{T}$ ;
18    return  $\mathcal{R}\mathcal{T}$ ;
19  end
20  else if  $root_1$  is a Split node then
21    Let  $X_i$  be the label of  $root_1$ , let  $\mathcal{R}\mathcal{T}$  be a new RPT with  $X_i$  labeling the root;
22    foreach child of  $root_1$ ,  $ch_i(\mathcal{R}\mathcal{T}_1)$  do
23      Let  $\mathcal{R}\mathcal{T}_i$  be a new RPT with a List node as root;
24      Append  $ch_i(\mathcal{R}\mathcal{T}_1)$  and  $\mathcal{R}\mathcal{T}_2^{R(X_i=x_i)}$  as children of  $\mathcal{R}\mathcal{T}_i$ ;
25      Put  $\mathcal{R}\mathcal{T}_i$  as child of  $\mathcal{R}\mathcal{T}$ ;
26    end
27    return  $\mathcal{R}\mathcal{T}$ ;
28  end
29  else if  $root_2$  is a Split node then
30    Let  $X_i$  be the label of  $root_2$ , let  $\mathcal{R}\mathcal{T}$  be a new RPT with  $X_i$  labeling the root;
31    foreach child of  $root_2$ ,  $ch_i(\mathcal{R}\mathcal{T}_2)$  do
32      Make a new RPT  $\mathcal{R}\mathcal{T}_i$  with a List node as root;
33      Append  $ch_i(\mathcal{R}\mathcal{T}_2)$  and  $\mathcal{R}\mathcal{T}_1^{R(X_i=x_i)}$  as children of  $\mathcal{R}\mathcal{T}_i$ ;
34      Put  $\mathcal{R}\mathcal{T}_i$  as child of  $\mathcal{R}\mathcal{T}$ ;
35    end
36    return  $\mathcal{R}\mathcal{T}$ ;
37  end
38  Let  $f_1$  and  $f_2$  be the factors (Value or Potential nodes) in  $\mathcal{R}\mathcal{T}_1$  and  $\mathcal{R}\mathcal{T}_2$ ;
39  return  $f_1 \cdot f_2$  (in this case  $f_1$  and  $f_2$  can be multiplied directly);
40 end

```

Algorithm 10: Algorithm to multiply two RPTs

Adding RPTs

The addition of two RPTs RT_1 and RT_2 rooted with $root_1$ and $root_2$ respectively is explained in Algorithm 11, and again the algorithm will perform different operations depending on the nature of the roots.

If $root_1$ is a List node, we will first compute the pairwise multiplication of all its children using Alg. 10 (line 5 of Alg. 11), obtaining a new RPT RT' . The next step will be to perform a recursive call to Alg. 11 this time with RT' and RT_2 (line 6 of Alg. 11). If $root_1$ is not a List node but $root_2$ is, the procedure is the same as described, multiplying this time the children of $root_2$ instead (lines 8 to 11 of Alg. 11).

If $root_1$ is a Split node labelled by variable X_i , then RT' will have as a root $root'$ a Split node labelled also with X_i . For every state j of X_i , $root'$ will have as child the result of a recursive call to Alg. 11 using as parameters the j -th child of $root_1$ and $RT_2^{R(X_i=x_j)}$, that is RT_2 restricted to the correspondent context of X_i . If $root_1$ is not a Split node but $root_2$ is, the procedure is the same, but being this time the label of $root_2$ the variable that labels the root of RT' (lines 20 to 26 of Alg. 11).

As in the *multiplication* operation, note that at this point, $root1$ and $root2$ can either be Value or Potential nodes, as the possibility of either of them being a List or a Split node has already been covered by the algorithm. Now the addition is a direct operation (lines 28 and 29 of Alg. 11), if both $root1$ and $root2$ are Value nodes, the result will be a Value node labelled by the result of the addition of the labels of both roots. If either $root1$ or $root2$, or both are Potential nodes, then the result will be a Potential node labelled by the sum of both potentials. In the case that both roots were Potential nodes we have the same problem as with the multiplication. Again we have handled this scenario by making the resultant Potential node always contain a probability tree, but other approaches could be developed for specific data structures.

Example 38 Consider the two simple RPTs defined in Fig. 4.23. We are now going to apply Alg. 11 to add them, considering them, from left to right, RT_1 and

```

1  sum( $\mathcal{RT}_1, \mathcal{RT}_2$ )
   Input: Two RPTs  $\mathcal{RT}_1$  and  $\mathcal{RT}_2$ 
   Output: A new RPT with the sum of  $\mathcal{RT}_1$  and  $\mathcal{RT}_2$ 
2  begin
3      Let  $root_1$  and  $root_2$  be the root of  $\mathcal{RT}_1$  and  $\mathcal{RT}_2$  respectively;
4      if  $root_1$  is a List node then
5          | Let  $\mathcal{RT}'$  be the product of all of the children of  $root_1$ , calculated by pairwise
6          | multiplication with Algorithm 10;
7          | return sum( $\mathcal{RT}', \mathcal{RT}_2$ );
8      end
9      else if  $root_2$  is a List node then
10         | Let  $\mathcal{RT}'$  be the product of all of the children of  $root_2$ , calculated by pairwise
11         | multiplication with Algorithm 10;
12         | return sum( $\mathcal{RT}', \mathcal{RT}_1$ );
13     end
14     else if  $root_1$  is a Split node then
15         | Make a new RPT  $\mathcal{RT}$  with a Split node as root;
16         | Label the root of  $\mathcal{RT}$  with the same label  $X_i$  of  $root_1$ ;
17         | foreach child of  $root_1$ ,  $ch_i(\mathcal{RT}_1)$  do
18         | | Obtain the child  $i$  of the root of  $\mathcal{RT}$  as sum( $ch_i(\mathcal{RT}_1), \mathcal{RT}_2^{R(X_i=x_i)}$ );
19         | end
20         | return  $\mathcal{RT}$ ;
21     end
22     else if  $root_2$  is a Split node then
23         | Make a new RPT  $\mathcal{RT}$  with a Split node as root;
24         | Label the root of  $\mathcal{RT}$  with the same label  $X_i$  of  $root_2$ ;
25         | foreach child of  $root_2$ ,  $ch_i(\mathcal{RT}_2)$  do
26         | | Obtain the child  $i$  of the root of  $\mathcal{RT}$  as sum( $ch_i(\mathcal{RT}_2), \mathcal{RT}_1^{R(X_i=x_i)}$ );
27         | end
28         | return  $\mathcal{RT}$ ;
29     end
30     Let  $f_1$  and  $f_2$  be the factors (Value or Potential nodes) in  $\mathcal{RT}_1$  and  $\mathcal{RT}_2$ ;
31     return The sum of  $f_1$  and  $f_2$  ;
32     // In this case  $f_1$  and  $f_2$  are potential or Value nodes and can be
33     // added directly
34 end

```

Algorithm 11: Algorithm to add two RPTs.

RT_2 respectively. As $root_1$ is a List node, the first step would be to multiply all its children using Alg. 10 (line 5 of Alg. 11), and afterwards we recursively apply Alg. 10 with the result of the multiplication and RT_2 , as we see in Fig. 4.24.

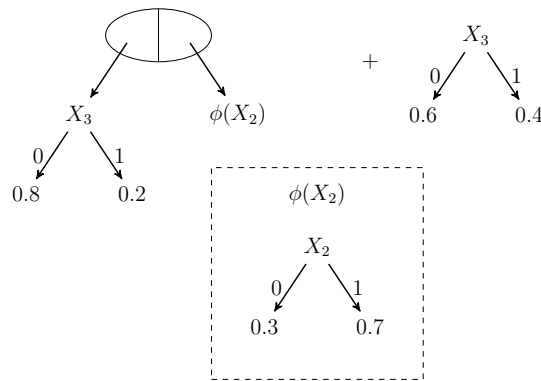


Figure 4.23: Addition of two RPTs (I).

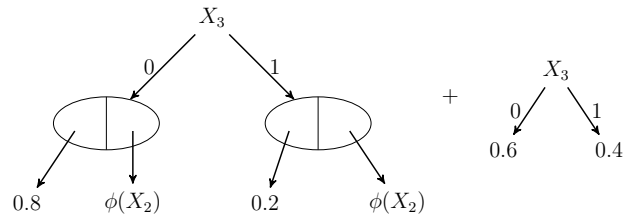


Figure 4.24: Addition of two RPTs (II).

The new RT_1 has a Split node as a root, so Alg. 11 tells us to build a Split node as a root for the resultant RPT, and recursively compute the sum of each of the children of $root_1$ with RT_2 restricted to every branch of the new Split node (lines 12 to 18 of Alg. 11). This step of the algorithm is explained in Fig. 4.25, where within the dashed line we can see the two recursive calls to Alg. 11 with the correspondent factors. After solving the List nodes in the recursive calls using Alg. 10, Alg. 11 is applied over the factors shown in Fig. 4.26. The final result corresponds to the RPT displayed in Fig. 4.27.

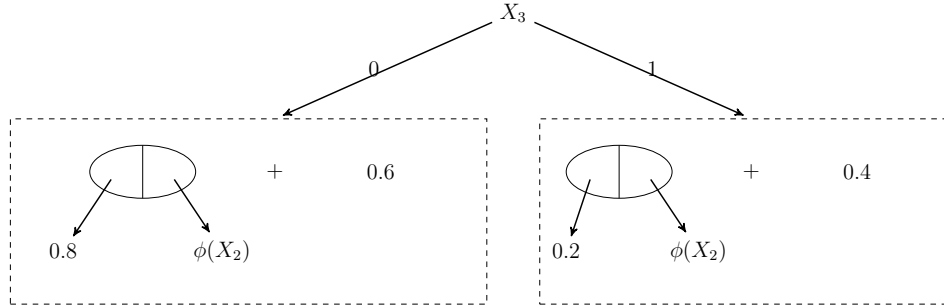


Figure 4.25: Addition of two RPTs (III).

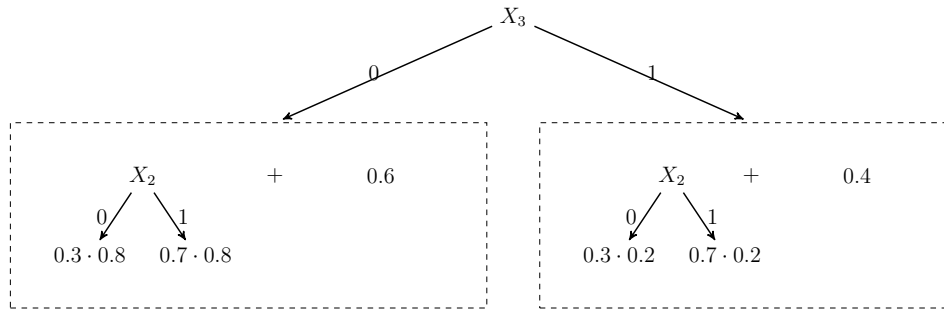


Figure 4.26: Addition of two RPTs (IV).

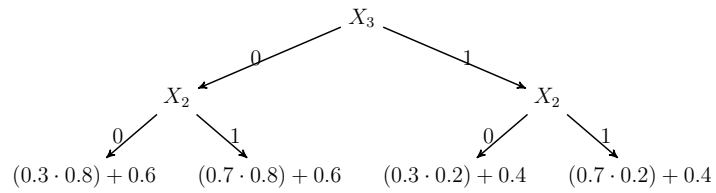


Figure 4.27: Addition of two RPTs (V).

Summing out a variable from an RPT

Given a Recursive Probability Tree \mathcal{RT} representing a potential ϕ defined for a set of variables $\mathbf{X}_{\mathbf{I}}$, and $\mathbf{J} \subseteq \mathbf{I}$, the *marginalisation* of \mathcal{RT} over $\mathbf{X}_{\mathbf{J}}$, denoted as $\mathcal{RT}^{\downarrow \mathbf{X}_{\mathbf{J}}}$, is a new RPT that represents the potential ϕ marginalised to $\mathbf{X}_{\mathbf{J}}$, that is, $\phi^{\downarrow \mathbf{X}_{\mathbf{J}}}(\mathbf{x}_{\mathbf{J}}) = \sum_{\mathbf{x}_{\mathbf{I}-\mathbf{J}}} \phi(\mathbf{x}_{\mathbf{J}}, \mathbf{x}_{\mathbf{I}-\mathbf{J}})$. The marginalisation is obtained by *deleting* from \mathcal{RT} all the variables $\{X_{\iota}\}$ where $\iota \in \mathbf{I} - \mathbf{J}$. The result of the marginalisation

does not depend on the order in which the variables are deleted, but this order can have a deep impact on the runtime performance of the operation.

The deletion of variable X_ι is denoted by $\mathcal{RT}^{\downarrow \mathbf{X}_I \setminus \{X_\iota\}}$ and represents the potential $\phi^{\downarrow \mathbf{X}_I \setminus \{X_\iota\}}$. The procedure to sum out a variable X_ι is a recursive process that traverses the RPT from root to leaves performing different operations according to the nature of the nodes in the RPT, as explained in Algorithm 12.

```

1  sumOut( $X_l, \mathcal{RT}$ )
   Input: An RPT  $\mathcal{RT}$  defined for  $\mathbf{X}_I$  and a variable  $X_l \in \mathbf{X}_I$  (the variable to be summed
       out)
   Output: An RPT for  $\mathcal{RT}^{\downarrow \mathbf{X}_I \setminus \{X_l\}}$ 
2  Let root be the root of  $\mathcal{RT}$ ;
3  if root is a Value node labelled with a real number  $r$  then
4      | Make a new RPT  $\mathcal{RT}'$  with a Value node as root;
5      | Put  $r \cdot |\Omega_{X_l}|$  as the label of the root of  $\mathcal{RT}'$  ;
6  end
7  else if root is a Potential node labelled with a potential  $P$  then
8      | Make a new RPT  $\mathcal{RT}'$  with a Potential node as root;
9      | if  $X_l$  is a variable in the domain of  $p$  then
10         | Put  $P^{\downarrow \mathbf{X}_I \setminus \{X_l\}}$  as the label of the root of  $\mathcal{RT}'$  (this is an external operation that
11           | depends on the data structure that holds the potential) ;
12         end
13         else
14             | Put  $p \cdot |\Omega_{X_l}|$  as the label of the root of  $\mathcal{RT}'$ ;
15         end
16     end
17 else if root is a Split node labelled with  $X_i$  then
18     | if  $X_i = X_l$  then
19         | Let  $\mathcal{RT}'$  be the sum of all the children of root using Algorithm 11;
20     | end
21     | else
22         | Make a new RPT  $\mathcal{RT}'$  with a Split node as root;
23         | Put  $X_i$  as the label of the root of  $\mathcal{RT}'$ ;
24         | foreach child of root,  $ch_i(\mathcal{RT})$  do
25             |  $\mathcal{RT}'_i \leftarrow \text{sumOut}(X_l, ch_i(\mathcal{RT}))$ ;
26             | Set  $\mathcal{RT}'_i$  as the ith child of the root of  $\mathcal{RT}'$ ;
27         | end
28     end
29 else if root is a List node then
30     | Let with and without be the list of children of root containing and not containing
31       |  $X_l$  respectively ;
32     | Let  $\mathcal{RT}_1$  be the multiplication of all the factors in the with list (using Algorithm 10);
33     |  $\mathcal{RT}_2 \leftarrow \text{sumOut}(X_l, \mathcal{RT}_1)$ ;
34     | Make a new RPT  $\mathcal{RT}'$  with a List node as root;
35     | Put  $\mathcal{RT}_2$  and all the factors in without list as children of the root of  $\mathcal{RT}'$ ;
36 end
37 return  $\mathcal{RT}'$  ;

```

Algorithm 12: Algorithm to marginalise out a variable from an RPT.

- If the root is a Value node (lines 3 to 6 of Alg. 12) or a *Potential* node that does not contain X_l in its domain (lines 7, 8 and 13 of Alg. 12), the result of the operation would be the node multiplied by the number of possible states of X_l , denoted as $|\Omega_{X_l}|$. This situation corresponds to the case in which the variable X_l is in the theoretical domain of the potential associated to the root, but it does not appear in an explicit way. The operation is done as if the potential depends of X_l .

If the *Potential* node has X_l in its domain, then the marginalisation operation is performed according to the data structure of the potential enclosed in the node (lines 7 to 11 of Alg. 12).

Example 39 Consider the RPT in Fig. 4.28, composed just of a *Potential* node. The potential that labels it has two variables, and it is represented using a probability tree, as detailed in the figure, within the dashed line. Imagine that we want to marginalise out the variable X_2 from this RPT, using Alg. 12. As X_2 belongs to the domain of the potential, we ask to the internal data structure to perform the marginalisation for us (line 10 of Alg. 12). The resultant RPT will be rooted by the result of the marginalisation, that in this example consists of a *Potential* node with a potential of just the variable X_1 , as displayed in Fig. 4.29.

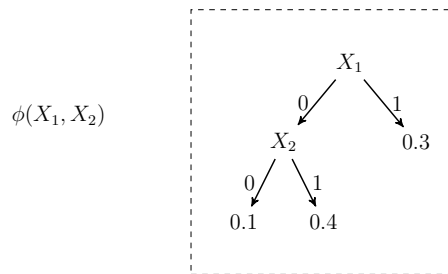


Figure 4.28: Marginalisation of RPTs (I).

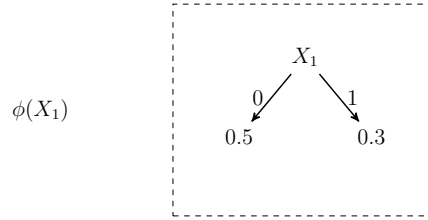


Figure 4.29: Marginalisation of RPTs (II).

- If the root of the RPT is a Split node, then if X_l labels it, a pairwise sum of its children using Alg. 11 is performed (lines 16 to 19 of Alg. 12). Otherwise, if X_l is not the label of the root, the marginalisation operation is recursively applied to every child of the root (lines 20 to 27 of Alg. 12).

Example 40 Consider now the RPT shown in Fig. 4.30, that consists of the same potential as in Fig. 4.28, but represented using Split nodes. If we apply Alg. 12 to this RPT to remove variable X_2 , we will create a new RPT with a Split node rooting it, labelled by the variable X_1 (lines 21 and 22 of Alg. 12). Now we recursively apply Alg. 12 to every of the children of the original Split node, obtaining the RPT shown in Fig. 4.31

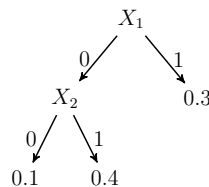


Figure 4.30: Marginalisation of RPTs (III).

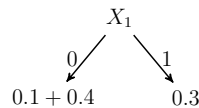


Figure 4.31: Marginalisation of RPTs (IV).

- If the root of the RPT is a List node, then all its children are sorted into two sets, one containing all the factors related to X_l , and the other containing

the remaining (line 30 of Alg. 12). This first set must be combined pairwise using Alg. 10 (line 31 of Alg. 12), and afterwards a recursive call to the algorithm is applied to the result of the multiplication (line 32 of Alg. 12). It is possible that the list of potentials containing X_i is empty. This happens when X_i belongs to the theoretical variables, but not to the list of explicit variables. In this case, the multiplication \mathcal{RT}_1 contains as root a Value node labelled with 1. When X_i is summed out in this potential, the result will have again a Value node as root labelled with $|\Omega_{X_i}|$. The final RPT would contain all the factors that were not related to X_i plus the result of the recursive call (lines 33 and 34 of Alg. 12).

Example 41 Consider for this example the RPT in Fig. 4.32. Again, our mission is to marginalise out X_2 from the RPT using Alg. 12. As the root of the tree is a List node, the first step consists on sorting the factors of the root according to whether they contain variable X_2 or not (line 30 of Alg. 12). In the figure we have enclosed both sets of factors into dashed boxes, with and without.

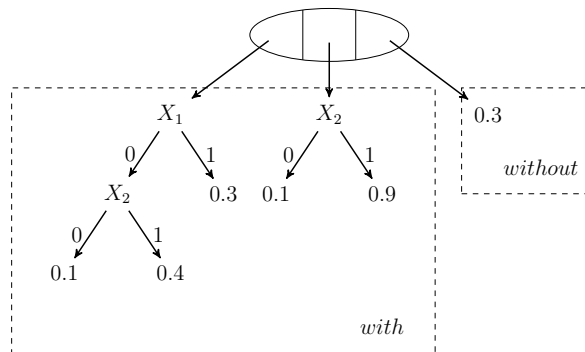


Figure 4.32: Marginalisation of RPTs (V).

The next step would be the combination of all the factors in with, that using Alg. 10 leads to the structure in Fig. 4.33. Note that this is just the with part of the original RPT.

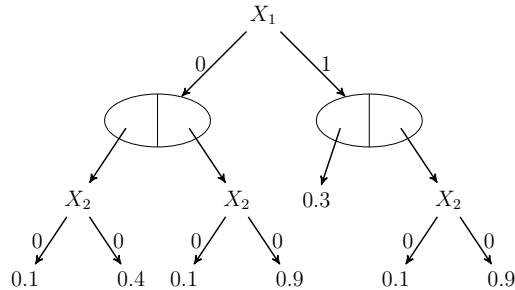


Figure 4.33: Marginalisation of RPTs (VI).

The following step is to recursively apply Alg. 12 to the result of the multiplication, that proceeds by solving the List nodes as shown in Fig. 4.34. This is done because for every child of X_1 (that both are List nodes), both with sets are composed of all the factors, that must be multiplied using Alg. 10.

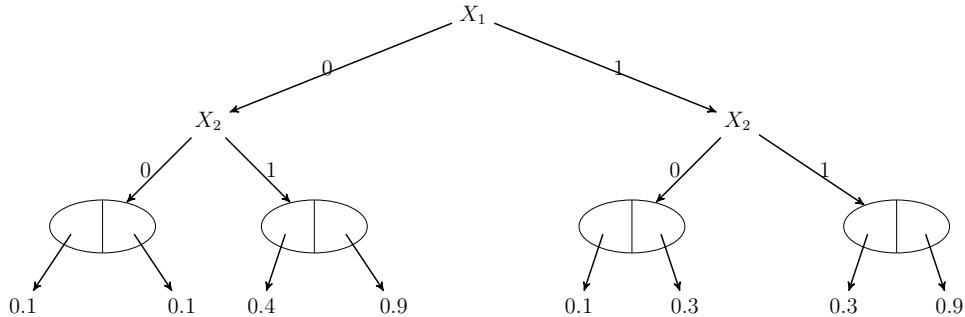


Figure 4.34: Marginalisation of RPTs (VII).

The next step is to marginalise out X_2 in every child (lines 23 to 26 of Alg. 12). As we have now two Split nodes of X_2 , we proceed to sum their children (line 18 of Alg. 12). The result of this last operation is put together with the without part as children of a new List node, root of the solution, obtaining the RPT in Fig. 4.35 as the solution to the marginalisation operation (lines 33 to 36 of Alg. 12).

Example 42 In this example, we consider the RPT in Fig. 4.36 and we want to marginalise out the variable X_2 using Alg. 12.

The first step, see Fig. 4.37, corresponds to line 30 in Algorithm 12, where the original RPT has been divided into two parts: the left part of the tree contains

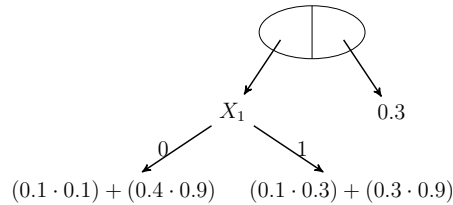


Figure 4.35: Marginalisation of RPTs (VIII).

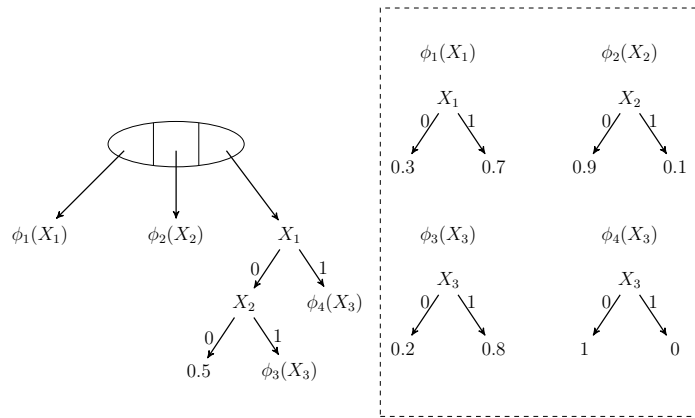


Figure 4.36: Marginalisation of RPTs (IX)

a potential $\phi_1(X_1)$ and the right one contains all the potentials containing the variable to be removed, X_2 , in their domains.

The step represented by Fig. 4.38 has selected the potential $\phi_2(X_2)$ that will be multiplied with the tree under the Split node for X_1 . This corresponds to line 31 of Algorithm 12, that calls Algorithm 10 in order to perform the multiplication.

In Fig. 4.39 the multiplication operation reaches a Split node for X_2 which conveys the restriction of the potential $\phi_2(X_2)$ passed as argument to the values of X_2 . Therefore, the leftmost value in Fig. 4.39 must be multiplied by $\phi_2(X_2 = 0) = 0.9$. The child for $X_2 = 1$ will be given after multiplying $\phi_3(X_3)$ and $\phi_2(X_2 = 1) = 0.1$.

Fig. 4.40 shows the result of removing X_2 from the factor for $X_1 = 0$, by adding the children of the Split node for X_2 according to the procedure described

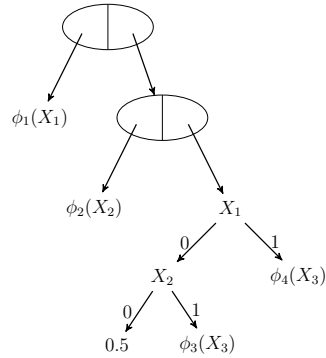


Figure 4.37: Marginalisation of RPTs (X)

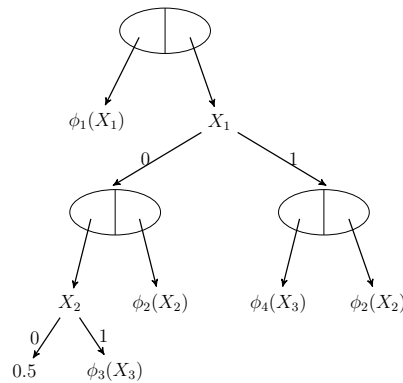


Figure 4.38: Marginalisation of RPTs (XI)

in Algorithm 11. Note that the right child of the Split node labelled with X_1 is a List node that only contains one factor involving X_2 , so the marginalisation operation is directly applied to it. This is shown in Fig. 4.41, where the addition of the children of the factor containing X_2 results in a Value node enclosing a 1. This structure contains the result of the operation.

RPTs can outperform other data structures when marginalising out variables from potentials, because this procedure only uses the parts of the tree where the variable to marginalise out is actually involved. Obviously the performance of the operation depends on the potential and its representation, this will work better

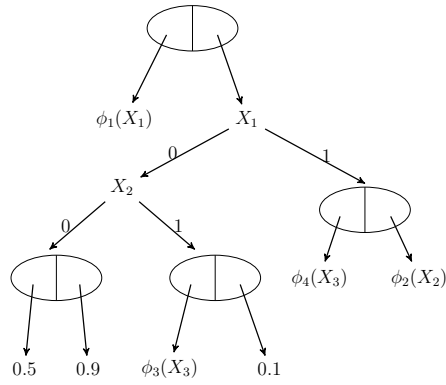


Figure 4.39: Marginalisation of RPTs (XII)

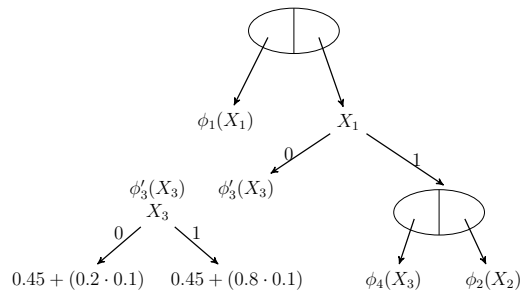


Figure 4.40: Marginalisation of RPTs (XIII)

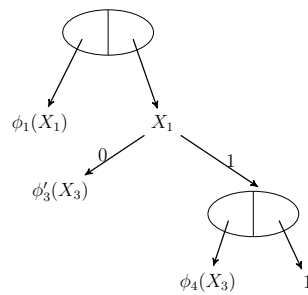


Figure 4.41: Marginalisation of RPTs (XIV)

the more factorised the potential is.

Example 43 For example, imagine that we are performing Variable Elimination over an RPT. Fig. 4.42 shows a step of the algorithm, where two potentials ($\phi_1(X_1, X_2, X_3)$ and $\phi_2(X_1, X_2)$) have to be combined in order to remove variable X_1 .

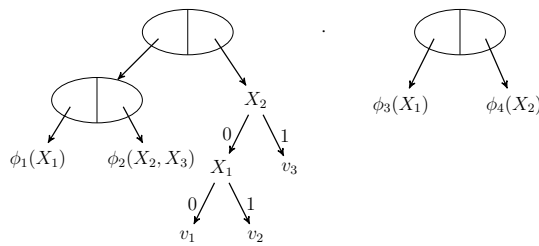


Figure 4.42: Step of Variable Elimination algorithm, X_1 is the variable to be removed.

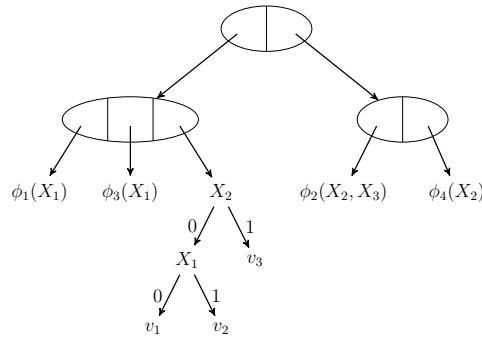


Figure 4.43: Reordered RPT achieved in a step of Alg. 12

The combination of these two RPTs is reduced to make them children of a List node. The next step would be applying Algorithm 12 to the resulting structure. Fig. 4.43 captures a step of the process, and the dotted line encloses the factors that directly affect variable X_1 . In order to remove it, it is necessary to combine the three factors, that is: $P_1(X_1) \cdot P_2(X_1) \cdot P(X_2, X_1)$, and then remove variable A from the resulting RPT. In Fig. 4.44 the result of the combination of the three

factors can be seen. The last step to complete the process is to remove A from the factor enclosed within the dotted line, by summing it out. To sum up, now the removal of X_1 only requires the combination of three simple potentials: $P_1(X_1) \cdot P_2(X_1) \cdot P(X_2, X_1)$.

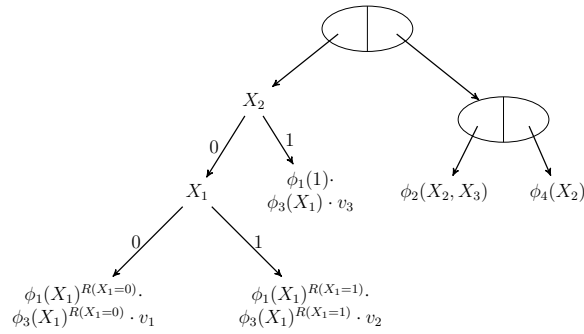


Figure 4.44: Result of applying Alg. 12 and Alg. 10.

4.4.4 Normalisation of RPTs

When using RPTs to perform inference, it is necessary to be able to represent probability distributions with them. For example, when applying the Variable Elimination algorithm to obtain $P(\mathbf{X}|\mathbf{e})$, if the retrieved RPT is not normalised, then we will not be able to obtain the probability value correspondent to $P(\mathbf{X}|\mathbf{e})$ from it, but the value of a potential $\phi(\mathbf{X}|\mathbf{e})$ instead. Any potential represented as an RPT can be turned into a joint probability distribution over the variables for which it is defined through the normalisation operation, that consists of making it add up to 1, as explained in Chapter 2 (Section 3.5.1.2).

The procedure to compute the total sum of a given RPT, sum_{RPT} , is detailed in Algorithm 13. It consists on traversing the tree from root to leaves, applying a different action depending on the kind of node. If *root* is a: *Value* node, just returns the corresponding value (lines 6 to 7 of Alg. 13.); *Potential* node: asks to the internal structure for the total sum of the encapsulated potential (lines 8 to 11 of Alg. 13.); *Split* node: for each configuration of the variables, retrieves the probability value using Alg. 7, and returns the addition of all the computed

values (lines 12 to 17 of Alg. 13.); *List* node: makes a recursive call to Alg. 13 with the result of combining using Alg. 10 every child of the *List* node (lines 18 to 21 of Alg. 13.).

```

1 sum( $\mathcal{RT}$ )
   Input:  $\mathcal{RT}$ : an RPT defined on  $\mathbf{X}_I$ 
   Output: a value that is the sum of all the values in the RPT.
2 begin
3    $root \leftarrow$  root node of  $\mathcal{RT}$ ;
4    $type \leftarrow$  type of  $root$  (Value, Potential, Split or List);
5   switch  $type$  do
6     case Value
7       | return label of  $root$ ;
8     case Potential
9       | Let  $\phi$  be the potential labelling the  $root$ ;
10      | Let  $sum$  be the addition of the values in  $\phi$ , computed by the
11      | data structure holding  $\phi$ ;
12      | return  $sum$ ;
13     case Split
14       | Let  $sum = 0$ ;
15       | foreach configuration  $\mathbf{x}_I$  of  $\mathbf{X}_I$  do
16       |   |  $sum + = \text{probValue}(\mathbf{x}_I)$ , using Alg. 7;
17       | end
18       | return  $sum$ ;
19     case List
20       |  $ch_1(\mathcal{RT}), ch_2(\mathcal{RT}) \dots ch_n(\mathcal{RT})$  children of  $root$ ;
21       | return  $sum(\prod_{i=1}^n ch_i(\mathcal{RT}))$  computing the product with Alg. 10;
22   endsw
23 end

```

Algorithm 13: Algorithm to compute the sum of an RPT.

The normalisation is easily achieved by computing $sum_{\mathcal{RPT}}$ using Algorithm 13, and then including a factor of $(1/sum_{\mathcal{RPT}})$, in the form of a *Value* node that multiplies the whole tree. If the root of the RPT is a *List* node, then the new *Value* node is added as a new child of it. If not, then a new root node is created as a *List* node, and the old RPT will be added as a child, along with the new normalisation factor.

Example 44 This process is illustrated in Fig. 4.45 where we normalise an RPT rooted with a Split node of X_1 . Following Algorithm 13 (lines 14 to 16) we proceed to recover the values for all the possible configurations of the RPT, which using Alg. 7 gives us the values 0.3, 0.5, 0.6 and 0.6, and by adding all of them we obtain the total sum of the RPT, that in this example is 2. The normalising factor, $1/2$ is now included in the RPT as a Value node, child of a new List node that roots the resultant normalised RPT, and sibling of the not normalised RPT.

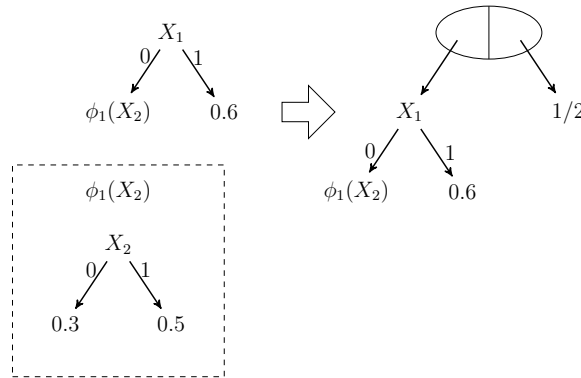


Figure 4.45: Normalisation of RPTs, the RPT represents $\phi(X_1, X_2)$. Enclosed in the dashed line is the potential encapsulated in the Potential node.

A conditional probability distribution $P(\mathbf{X}_J|\mathbf{X}_K)$ can also be represented using an RPT. In this case, for any configuration of the parent variables $\mathbf{X}_K = \mathbf{x}_K$, the restriction of the potential to this configuration should be normalised (i.e. sum up to 1). This conditional normalisation is performed by adding to the RPT a new factor that will be a split chain of all the variables in \mathbf{X}_K , with Value nodes in the leaves. Those Value nodes store normalisation factors, that correspond to $1/\text{sum}_{\mathcal{R}_{\mathcal{J}}^R(\mathbf{x}_K)}$, where the denominator denotes the sum of all the values of the RPT restricted to each configuration of the variables in \mathbf{X}_K .

Example 45 An example of this is represented in Fig. 4.46 where now we have as a normalisation subtree a Split node of the parent variable X_2 fathering the correspondent normalisation factors for each of its states. Note that now Alg. 13

is invoked with the RPT restricted to the correspondent configuration of the set of parents, in this example it is called twice, one for the RPT restricted to $\{X_2 = 0\}$ (obtaining 0.9) and a second call with the RPT restricted to $\{X_2 = 1\}$ (obtaining 1.1).

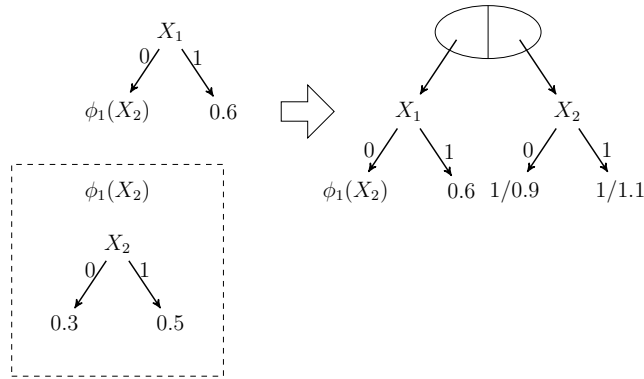


Figure 4.46: Normalisation of RPTs, the RPT represents $P(X_1|X_2)$. Enclosed in the dashed line is the potential encapsulated in the Potential node.

This method must be considered as a first approximation to the problem of the conditional normalisation of an RPT, as the procedure explained above implies increasing the size of the model according to the variables in \mathbf{X}_K and their number of states. Normalising to conditional without adding this factor is a complex problem due to the internal factorisations represented through List nodes within an RPT. This problem will be discussed again in Chapter 6.2, where we propose a methodology to learn normalised RPTs.

In both variations of the operation, the normalisation remains as a factorisation until it is needed to get the actual value during the inference process, where both factors are combined.

4.5 Experimental evaluation

We have carried out a series of experiments in order to check the performance of RPTs with respect to other structures for probability potentials representa-

tion. Our intuition is that RPTs are a more suitable data structure to represent potentials that can be factorised, as RPT's basic operations are optimized to take advantage of this kind of patterns, among others, like context-specific independencies. RPTs should perform quicker than probability trees, that basically benefit from context-specific independencies, and certainly RPTs should outperform a representation based on tables (CPTs) that even though they are usually inefficient, their use is quite widespread.

The idea of these experiments is to do inference over a Bayesian network whose conditional probability distributions present a given degree of factorability and also context-specific independencies. We want to compare the runtime performance, along with the size of the handled structures during the process, when storing the potentials as RPTs, and compare to the performance of probability trees and conditional probability tables.

We chose a moderate-sized well-known Bayesian network structure for our experimentation, the Barley network [66] (details are available in Appendix 7.2.4). Preserving the skeleton of the network, we replaced each conditional probability distribution with an RPT representing a factorised conditional distribution. Afterwards, we run the Variable Elimination algorithm over the Bayesian network and measured the total time of computing the posterior probabilities for all the variables in the network, along with the sizes of the structures stored during the process.

For comparison, we repeated the inference, first transforming the RPTs into PTs and secondly turning them into CPTs. Using this procedure, we could check how PTs and CPTs handle distributions that have a given level of factorisations. With this study we aim at emphasizing the real need of using more efficient data structures when designing probabilistic graphical models.

4.5.1 Generation of a random RPT

The RPTs used in the experiments were generated at random. The procedure to generate a random RPT runs as follows. We considered three parameters: a set of variables \mathbf{X} , a probability p_S for the generation of Split nodes as inner nodes, and a probability p_P for the generation of Potential nodes on the leaves.

The procedure is recursive, generating first the root node and going down to the leaves. If the size of \mathbf{X} is less or equal to two, we will generate a leaf node: if \mathbf{X} is not empty, we generate a Potential node of its variables. If there are no variables in \mathbf{X} , then we incorporate a Value node labelled by a random value to the structure. If the number of remaining variables is between 3 and 5, the probability of generating an inner node (either Split or List node) is 0.2. If the number of variables considered is equal or greater than 5, then we will always generate an inner node.

When generating an inner node, a Split node of a random variable within the set will be created with probability p_S . For List nodes, a maximum of 5 children is allowed in order to bound the size of the factorisation. The choosing of the number of children for a List node follows a Poisson distribution. For each child of the List node, we randomly create a subset of variables to be represented. The intersection between the subsets doesn't have to be empty.

When building a leaf node, we will generate either a Potential node or a Value node according to p_P . To build a Value node, a random number between 0 and 1 is generated and stored in the node. The procedure to create a Potential node runs as follows: randomly obtain a number n between 1 and the size of the set of remaining variables. Randomly retrieve n variables from the set and store them as the variable set for the new potential enclosed within the new Potential node. For each configuration of the new set of variables, store a random value between 0 and 1, as its probability value. Potential nodes are internally represented as probability trees, with a small prune factor of 0.001, to avoid the storage of many similar values within the structure.

At the end of the process, we check if any of the variables of \mathbf{X} was left out. If this is the case, a Potential node with the remaining variables is created, and attached to the previously generated RPT through a List node that becomes the new root of the final RPT.

In this way it is possible to calibrate the RPT to represent a potential with the desired level of factorisations or context-specific independencies. The generated RPTs are normalised afterwards as explained in Sect. 4.4.4 in order to keep it as a conditional probability distribution.

4.5.2 Fixed RPT generation parameters

The aim of this experiment is to check the performance of the inference process over the considered data structures (RPTs, PTs and CPTs) when the distributions in the network can be highly factorised. RPTs should take advantage of this over other data structures like traditional probability trees, that only increase the performance of the algorithm when the distributions have context-specific independencies.

To do so, the procedure explained in Sect. 4.5.1 was applied to generate one RPT corresponding to each conditional probability distribution of the original Bayesian network, with the following parameters: $p_S = 0.001$ and $p_P = 0.8$. This ensures that the inner nodes are more likely to be List nodes, and this means that the distribution would be highly factorised. In the leaves we will have mostly Potential nodes ($p_P = 0.8$), that are internally stored as probability trees.

After the transformation, the Variable Elimination algorithm was executed over the network, in order to compute the posterior distribution for each variable. Then the RPTs of the Bayesian network were transformed into PTs and CPTs, and the Variable Elimination algorithm was executed again over the structures. The procedure to turn an RPT into a PT or a CPT is easy: we just have to build the structures and fill the values by recovering the values for each configuration of the variables from the RPT. In the case of PTs, they can be pruned afterwards if we chose to do so.

Probability trees spend some computing time in pruning the trees. They do so by pruning the branches that have repeated values, or when the variation is smaller than a given threshold. The threshold used for this experiment was 0.0001, that is a very small value [17], so the trees may only be pruned when the values are the same or almost the same. In this experiment we also considered disabling the prune of the probability trees, so no time would be used in other process than the inference itself.

Fig. 4.47 shows the results obtained in 30 repetitions of the experiment. It can be seen how the inference process is slower in unpruned trees and conditional probability tables than using RPTs and probability trees for conditional probability distributions representation. Also, RPTs outperform PTs in all the cases.

4.5 Experimental evaluation

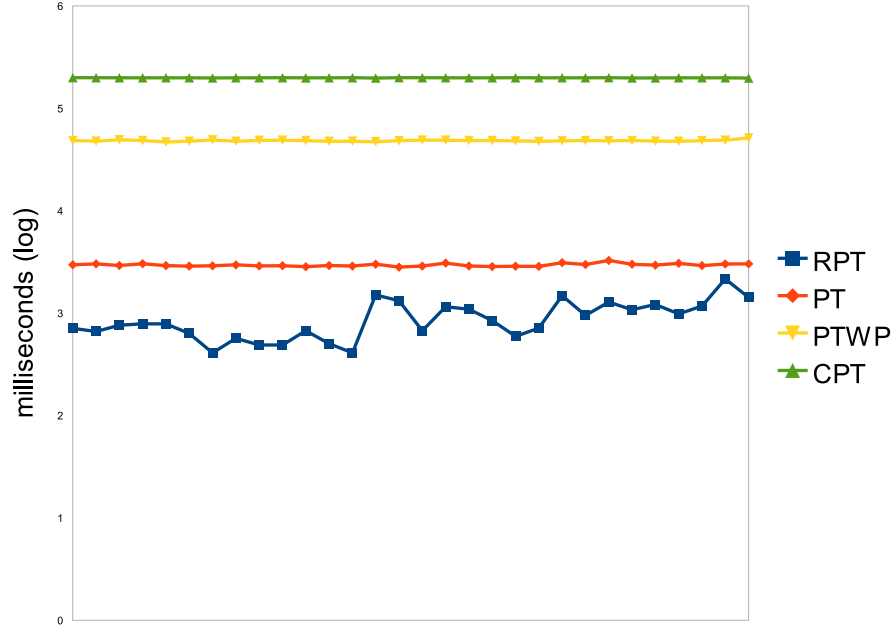


Figure 4.47: Experiment 1: execution time of removing all the variables in the network over RPTs, PTs, unpruned PTs (PTWP) and CPTs.

In terms of size of the generated structures, RPTs are usually more compact. We measured the size of the biggest structure stored during the inference process, understanding by size the number of inner and leaf nodes. In RPTs, we will count Potential nodes as the total number of nodes of the inner structure: as we are representing them using probability trees, the size is the number of inner nodes plus the number of leaves. We also recorded the number of probability values stored in the leaves of the biggest structure. For CPTs, we understand by size the number of stored probability values.

Table 4.1 shows the average and standard deviation of the maximum sizes recorded on each one of the 30 runs of the algorithm, being the RPTs smaller and observing a smaller standard deviation. CPTs and unpruned PTs correspond in this experiment to exact representations of the distributions, so the number of probability values used by both structures are the same. However, unpruned PTs perform better in terms of time, this can be due to the optimization of the basic operations using probability trees.

4.5 Experimental evaluation

	Number of Prob. Values			
	RPT	PT	unpruned PT	CPT
average	6980,72	7504,26	2177280	2177280
standard deviation	1522,12	12633,19	0	0
	Number of Nodes			
	RPT	PT	unpruned PT	CPT
average	7849,35	8463,29	2527747	2177280
standard deviation	1706,98	14695,73	0	0

Table 4.1: Average and standard deviation of the size of the biggest structure stored during the inference

We can conclude from this experiment that performing inference over probability distributions with a high level of factorisations is more efficient, both in terms of time consumption and storage space, when storing the potentials using RPTs instead of PTs.

4.5.3 Varying RPT generation parameters

The aim of this second experiment was to check the performance of the Variable Elimination over different structures of RPTs, that is to say, when the conditional probability distributions present different levels of factorability. This time the RPTs were generated varying the parameter p_S from 0.0 to 1, and the same procedure as explained in Sec. 4.5.2 was performed. Smaller values of p_S are translated into RPTs representing highly factorised distributions, and the higher the p_S value, the less factorised the distribution, and therefore closer to a PT representation. For each value of the parameter, the algorithm was run 30 times, and the average of the times obtained is shown in Fig. 4.48, along with the performance of probability trees, unpruned probability trees, and conditional probability tables.

4.5 Experimental evaluation

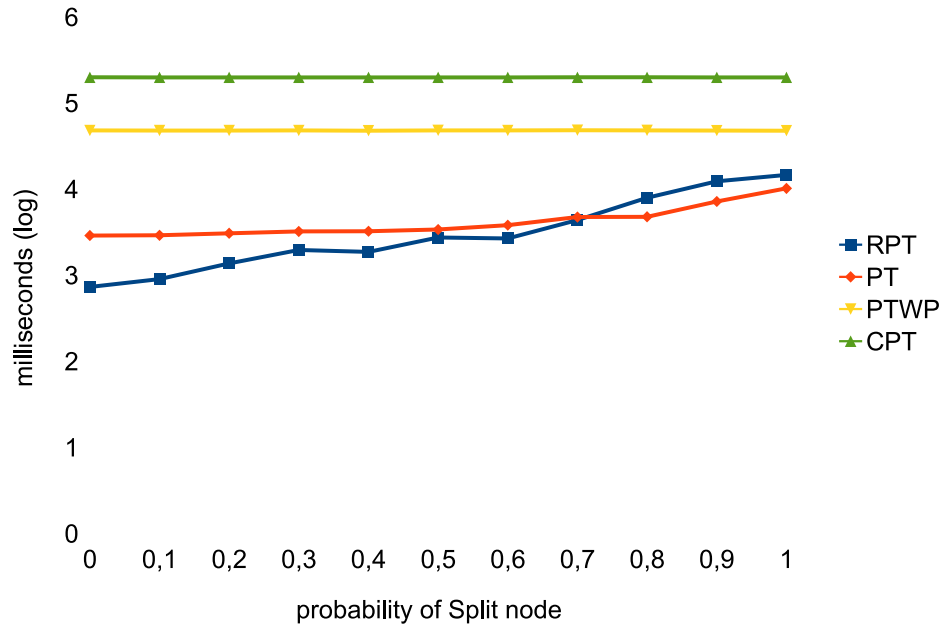


Figure 4.48: Experiment 2: time average of the Variable Elimination algorithm over probability trees, unpruned PTs (PTWP), CPTs and RPTs, considering different factorisation levels.

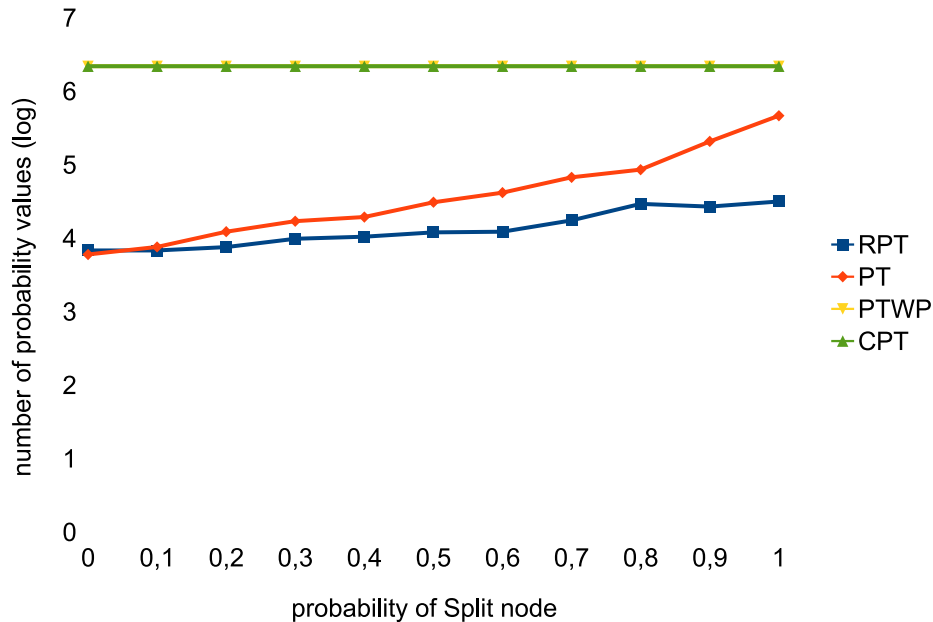


Figure 4.49: Experiment 2: size average, in terms of number of stored probability values, of the biggest structure stored during the inference process over RPTs, CPTs, probability trees and unpruned probability trees (PTWP).

The results suggest that RPTs run faster in average than PTs when the degree of factorisations within the probability distribution is high (low values of p_S). The performance of RPTs decrease when the probability of obtaining a Split node is close to 1, mainly because the RPTs operations are optimized to search for factors within the structure, which consumes time, and also due to the normalisation factor included in the RPTs, that increases their complexity. However, the RPT structures remain smaller: Fig. 4.49 shows the average of the number of probability values stored to represent the biggest structure stored in memory during the inference process for each step of the experiment, in logarithmic scale. Figure 4.50 shows the size, in terms of number of nodes in the structure, as in the previous experiment.

We see that the tendency is the same in both figures, on average the RPT structures are smaller, but increasing in size as the probability of Split node grows.

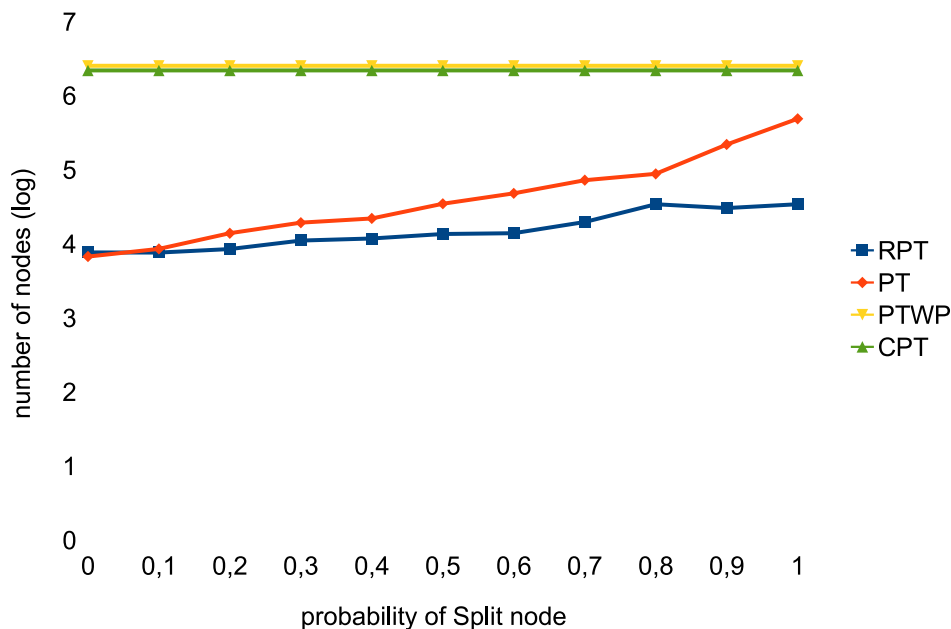


Figure 4.50: Experiment 2: size average, in terms of number of stored nodes, of the biggest structure stored during the inference process over RPTs, CPTs, probability trees and unpruned probability trees (PTWP).

The presence of Potential nodes at the leaves of the RPTs help reducing their size as they compress the information. However, this is not an inherent benefit of RPTs: if we represent the same distribution with RPTs without List nodes and PTs, using the same ordering for the variables in both trees, the size in terms of number of nodes (as we are computing it) should be the same. This equal ordering is not guaranteed in this experimentation, so it might be the case that RPTs are capturing context-specific independencies hidden for the equivalent PT representation.

4.6 Conclusions and Future Work

This chapter gives a detailed explanation of a powerful data structure for representing probabilistic potentials. RPTs present a high expressive ability, and they

can take advantage of context-specific independencies and possible factorisations of the probability potentials in order to obtain more compact and effective representations than other structures like Probability Trees or Conditional Probability Tables. In this chapter we have explained how the basic operations to perform inference (restriction, combination and marginalisation) work on RPTs, along with how to normalise the structure.

Several examples have been shown where it seems that depending on the nature of the distributions, RPTs are able to reduce the computation time of the inference process when comparing to PTs and CPTs, using compact representations. We expect that the ability for capturing context specific independencies and factorising potentials is likely to be more faithful in representing the true underlying probability distributions, avoiding overfitting problems when learning from small data sets.

Part III
Inference

Chapter 5

Inference with factorised representations

The prospective importance of capturing factorisations for more effective representations of probabilistic potentials has been discussed in previous chapters, along with the procedure followed by Recursive Probability Trees to take advantage of these particularities to speed up inference and, when possible, compact the structure. The problem is that most of the time the presence of these factorisations within the distributions is not obvious, or maybe the potentials are not dividable at all, so building a factorised representation like an RPT is not an straightforward task.

In this chapter we describe our proposal for finding multiplicative decompositions of probabilistic potentials, distinguishing between the obtaining of exact and approximate decompositions. The methodology presented here is in general valid for different representations of potentials, like probability tables and probability trees. We will focus on the latest to illustrate the algorithm, as probability trees are the simplest case of RPTs, and the translation from a factorisation with probability trees as factors to an equivalent Recursive Probability Tree is immediate. It is also handy to exemplify our proposal using probability trees as previous factorisation methods operate over them, so an accurate comparison between our proposal and existing algorithms can be directly performed.

5.1 Motivation

Previous approaches try to deal with finding factorisations within probability trees looking for proportional factors in subtrees. The problem of this approach, that will be discussed later in this chapter, is that it is highly dependent on the ordering of the variables in the tree. This means that for a given probability tree, we might have to rebuild the tree for all the possible ordering of its variables before finding an optimal factorisation. This approach can easily become intractable in terms of efficiency, so it would be inconceivable to incorporate such approach, for instance, to the inference.

In this chapter we define a new approach to the factorisation of probabilistic potentials, providing an efficient framework to either find the exact decomposition hidden in the potential or provide an accurate approximation. We illustrate that the new factorisation procedure can be used for controlling the tradeoff between efficiency and accuracy in approximate inference algorithms, through a modified version of the Variable Elimination scheme [24; 25].

5.2 Classical factorisation of probability trees

In this section we discuss a previous approach at factorising potentials [67; 68; 69] using probability trees. This methodology exploits the existence of proportional subtrees, as defined in Chapter 3.7.1.3.

If proportionalities are found in the structure, then it is possible to build an *exact decomposition* of the tree. The algorithm looks for proportional sub-trees located below a given variable. Therefore, the performance is highly dependent on the order of the variables in the tree. The method that determines if we have encounter proportional subtrees is denoted as **isProportional**(\mathcal{PT}, α). This method takes a probability tree \mathcal{PT} and returns true if its subtrees are proportional, storing in α the proportionality factors. The algorithm returns false if the subtrees are not proportional.

Once we notice that there are proportionalities in the tree, and we have defined the location of the pattern, we use algorithm **obtain_Factors**($\mathcal{PT}, Y, \mathbf{w}, \alpha, List_\phi$) (Alg. 14) to compute the factorisation. This algorithm takes a probability tree

5.2 Classical factorisation of probability trees

\mathcal{PT} , where under the context \mathbf{w} exists a subtree rooted by Y that encodes a proportionality defined by the factors in α . This algorithm modifies \mathcal{PT} by replacing the descendent of Y with the correspondent proportionality factors in α (lines 8 to 15 of Alg. 14), and includes a new probability tree in $List_\phi$ that fills with 1 all the subtrees except the one defined by \mathbf{w} (lines 4 to 7 of Alg. 14), that is replaced with the subtree under the configuration $\{\mathbf{w}, Y = y_0\}$. The combination of \mathcal{PT} and the potential in $List_\phi$ gives the original potential as a result.

```

1 obtain Factors( $\mathcal{PT}, Y, \mathbf{w}, \alpha, List_\phi$ )
   Input: A probability tree  $\mathcal{PT}$  on  $\mathbf{X}$ , a variable  $Y \in \mathbf{X}$ , a configuration  $\mathbf{w}$ ,
           a list of proportionality factors  $\alpha$  and an empty list of potentials
            $List_\phi$ .
   Output: A modification of  $\mathcal{PT}$  with respect to  $Y$  and an inclusion of a
             potential in  $List_\phi$ . The multiplication of the modified  $\mathcal{PT}$  and
             the new element of  $List_\phi$  results in the original potential defined
             in  $\mathcal{PT}$ 

2 begin
3   Let  $\mathcal{PT}_F$  be a copy of  $\mathcal{PT}$ 
4   for each configuration  $\mathbf{z}$  not compatible with  $\mathbf{w}$  do
5     | Make  $\mathcal{PT}_F^{R(\mathbf{z})} = 1$ .
6   end
7   Make  $\mathcal{PT}_F^{R(\mathbf{w})} = \mathcal{PT}^{R(\mathbf{w}, Y=y_0)}$ .
8   for each configuration  $\mathbf{y}_i \in \Omega_Y$  do
9     | if  $y_i == y_0$  then
10    | |  $\mathcal{PT}_F^{R(\mathbf{w}, Y=y_0)} = 1$ 
11    | end
12    | else
13    | |  $\mathcal{PT}_F^{R(\mathbf{w}, Y=y_i)} = \alpha_i$ 
14    | end
15  end
16   $List_\phi = List_\phi \cup \mathcal{PT}_F$ 
17 end

```

Algorithm 14: Factorisation of a probability tree with proportionalities under a given context.

Therefore, in order to perform a factorisation of a probability tree it is needed to explore the structure searching for proportionalities, and once detected, build

5.2 Classical factorisation of probability trees

the factorisation with the described methodology. The general framework is described in Alg. 15, where we recursively check the nodes in the tree using a breath-first search looking for the variable Y . Once we find it (lines 12 to 22 of Alg. 15), we check if the subtrees under the node labelled by Y are proportional (line 16 of Alg. 15), and if so we proceed to build the decomposition (line 18 of Alg. 15). Otherwise the tree is left untouched.

```

1 Factorise_classic( $\mathcal{PT}, Y, List_\phi$ )
   Input: A probability tree  $\mathcal{PT}$  on  $\mathbf{X}$ , a variable  $Y \in \mathbf{X}$  and an empty list of
           potentials  $List_\phi$ .
   Output: A decomposition of  $\mathcal{PT}$  with respect to  $Y$  given as a list of
           potentials  $List_\phi$ .
2 begin
3   Let  $L$  be the label of  $\mathcal{PT}$ 's root.
4   if  $L$  is a number then
5     | return False.
6   end
7   else
8     | Let  $X = L$ .
9     | if  $X == Y$  then
10    | | return True.
11    | end
12    | else
13    | | for each  $x_k \in \Omega_X$  do
14    | | | Let  $\mathcal{PT}_k = \mathcal{PT}^{R(x_k)}$ .
15    | | | if Factorise_classic( $\mathcal{PT}_k, Y, List_\phi$ ) then
16    | | | | if isProportional( $\mathcal{PT}_k, \alpha$ ) then
17    | | | | | Let  $C = (X_C = x_C, X = x_k)$ .
18    | | | | | obtain_Factors( $\mathcal{PT}, Y, C, \alpha, List_\phi$ )
19    | | | | end
20    | | | end
21    | | end
22    | end
23  end
24  return False.
25 end

```

Algorithm 15: Classical factorisation of probability trees.

5.2 Classical factorisation of probability trees

Example 46 As an example, consider the probability distribution for variables X_1 , X_2 and X_3 represented as the probability tree shown in Fig. 5.1. It can be seen that the sub-trees below X_2 , for branch $X_1 = 0$, are proportional. The classical factorisation procedure would decompose the tree in Fig. 5.1 as the product of the two trees in Fig. 5.2, i.e., the value of the decomposition for a given configuration of the involved variables, is equal to the result of multiplying the values obtained by evaluating that configuration in both trees. Notice that, in this case, the size of the resulting factorisation (which is the sum of the sizes of individual trees that comprise the factorisation) is higher than the one of the original tree, but applying the classical procedure to the tree in the left side of Fig. 5.2 it could be further decomposed, as the two sub-trees below X_2 are also proportional when $X_1 = 0$.

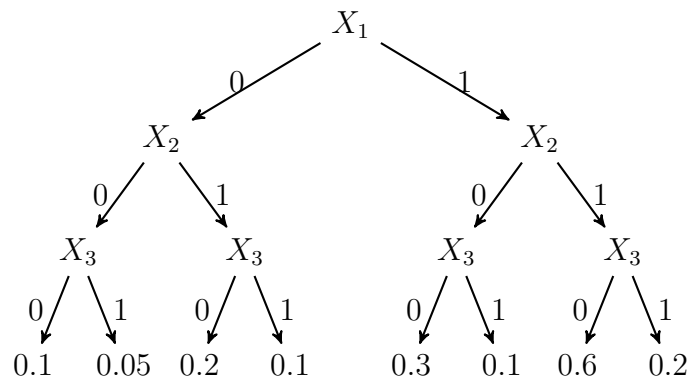


Figure 5.1: A probability tree with proportional sub-trees.

This methodology was extended to cope with cases where an exact factorisation of the probability tree could not be found. The methodology is the same as described above, but this time they relax the algorithm **isProportional**(\mathcal{PT}, α) to find a set of proportionality coefficients that minimizes the divergence between the original tree and the factorisation. This *approximate factorisation* of probability trees used to represent the probabilistic potentials [68; 69] has been proven to be an appropriate way of controlling the tradeoff between complexity and accuracy of propagation algorithms.

But it can even be the case that the classical factorisation algorithm (both exact and approximate approaches) is not able to decompose a tree with respect

5.2 Classical factorisation of probability trees

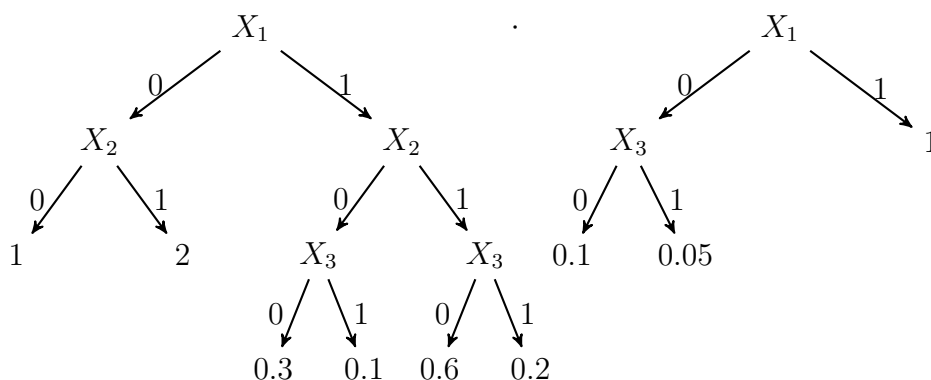


Figure 5.2: Decomposition of the tree in Fig. 5.1 using Alg.15.

to a given variable, if it is located away from the root of the tree. For instance, the tree in Fig. 5.3 can be expressed as the product of the two trees in Fig. 5.4, but such decomposition cannot be obtained using the classical factorisation technique, since variable X_2 is located near to the leaves. In theory, classical factorisation could find the right decomposition, if the variables in the tree are re-arranged until the appropriate order is obtained. However, that would be too costly in terms of time.

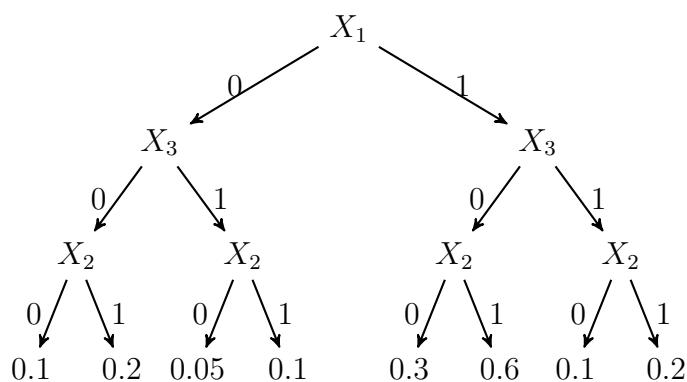


Figure 5.3: Probability tree that cannot be factorised by variable X_2 with classical factorisation.

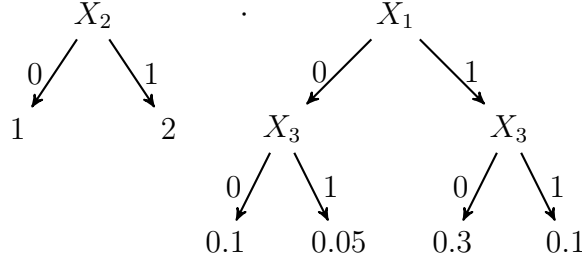


Figure 5.4: Decomposition of the tree in Fig. 5.3, which cannot be obtained using Alg.15.

5.3 Fast factorisation of probability trees

In order to deal with the disadvantages of the classical factorisation procedure described above, we propose a new methodology that quickly performs a decomposition of a given probability tree. In case that an exact decomposition of the probability tree can not be computed, we offer an efficient measure that defines the best variable to guide the decomposition.

5.3.1 Exact Decomposition

In what follows we propose a factorisation technique able to deal with situations like the one described in Figs. 5.3 and 5.4, in a computationally efficient way. The key concept is the decomposability of trees, as stated in the next definition.

Definition 4 *A probability tree \mathcal{PT} defined for a set of variables \mathbf{X} is said to be decomposable with respect to $\mathbf{Y} \subsetneq \mathbf{X}$ if there are two probability trees \mathcal{PT}_1 and \mathcal{PT}_2 , defined for variables \mathbf{Y} and $\mathbf{Z} \subsetneq \mathbf{X}$ respectively, such that*

1. $\mathbf{Y} \cap \mathbf{Z} = \emptyset$,
2. $\mathbf{X} = \mathbf{Y} \cup \mathbf{Z}$ and
3. $\mathcal{PT} = \mathcal{PT}_1 \cdot \mathcal{PT}_2$.

Notice that the tree in Fig. 5.3 is decomposable with respect to X , as the three conditions in Def. 4 are met. A detailed procedure for finding a factorisation of

5.3 Fast factorisation of probability trees

a decomposable tree is described in Alg. 16. It makes use of the *restriction* operation over probability trees as defined in Chapter 3.7.1.1.

The algorithm starts by computing the proportionality factors of the tree (lines 3 to 9 of Alg. 16), as one of the factors of the decomposition will be a probability tree of the variables in \mathbf{Y} as inner nodes, and the computed proportionality factors as leaves (line 10 of Alg. 16). The second factor of the decomposition will be \mathcal{PT} restricted to the first configuration of the variables in \mathbf{Y} (line 11 of Alg. 16). The last step of the algorithm is to normalise both factors (lines 13 and 14 of Alg. 16).

```

1 Factorise( $\mathcal{PT}, \mathbf{Y}$ )
   Input: A probability tree  $\mathcal{PT}$  on  $\mathbf{X}$  and set of variables  $\mathbf{Y} \subset \mathbf{X}$ .
   Output: A decomposition of  $\mathcal{PT}$  with respect to  $\mathbf{Y}$ , given as two
             probability trees  $\{T_1, T_2\}$ .
2 begin
3   Let  $\mathbf{y}_0, \dots, \mathbf{y}_{r-1}$  be the elements of  $\Omega_{\mathbf{Y}}$ .
4    $\mathbf{Z} \leftarrow \mathbf{X} \setminus \mathbf{Y}$ .
5   Let  $(\mathbf{Z} = \mathbf{z})$  be any configuration for all variables in  $\mathbf{Z}$ .
6   Let  $\alpha_0, \dots, \alpha_{r-1}$  be the leaves of tree  $\mathcal{PT}^{R(\mathbf{Z}=\mathbf{z})}$ .
7   for  $i \leftarrow 0$  to  $r - 1$  do
8     | Let  $\beta_i = \frac{\alpha_i}{\alpha_0}$ .
9   end
10  Let  $\mathcal{PT}_1$  be a tree with the variables in  $\mathbf{Y}$  as inner nodes and
      $\beta_0, \dots, \beta_{r-1}$  as leaves.
11   $\mathcal{PT}_2 \leftarrow \mathcal{PT}^{R(\mathbf{Y}=\mathbf{y}_0)}$ .
12  Let  $s_{\mathcal{PT}}, s_{\mathcal{PT}_1}$  and  $s_{\mathcal{PT}_2}$  be the sum of all the values in  $\mathcal{PT}, \mathcal{PT}_1$  and  $\mathcal{PT}_2$ 
     respectively.
13   $\mathcal{PT}_1 \leftarrow \mathcal{PT}_1 \cdot \frac{s_{\mathcal{PT}}}{s_{\mathcal{PT}_1}}$ .
14   $\mathcal{PT}_2 \leftarrow \mathcal{PT}_2 \cdot \frac{1}{s_{\mathcal{PT}_2}}$ .
15  return  $\{\mathcal{PT}_1, \mathcal{PT}_2\}$ 
16 end

```

Algorithm 16: Fast factorisation of probability trees.

The next proposition shows that, if a tree is decomposable, then Alg. 16 actually finds a decomposition consistent with Def. 4.

5.3 Fast factorisation of probability trees

Proposition 1 *Let \mathcal{PT} be a probability tree defined for a set of variables \mathbf{X} . If \mathcal{PT} is decomposable with respect to $\mathbf{Y} \subsetneq \mathbf{X}$, then Alg. 16 returns two probability trees \mathcal{PT}_1 and \mathcal{PT}_2 that factorise \mathcal{PT} according to Def. 4.*

Proof 1 *The sets of variables over which \mathcal{PT}_1 and \mathcal{PT}_2 are defined, are determined in Steps 10 and 11. It is clear that, according to the definition of restriction in Chapter 3.7.1.1, conditions (1) and (2) in Def. 4 hold. Now we will show that condition (3) also holds after applying Alg. 16.*

If \mathcal{PT} is decomposable, then for all \mathbf{y}, \mathbf{z}

$$\mathcal{PT}(\mathbf{y}, \mathbf{z}) = \mathcal{PT}_1(\mathbf{y}) \cdot \mathcal{PT}_2(\mathbf{z}). \quad (5.1)$$

Also, according to the definition of the restriction operation, for all \mathbf{y}, \mathbf{z}

$$\mathcal{PT}(\mathbf{y}, \mathbf{z}) = \mathcal{PT}^{R(\mathbf{Y}=\mathbf{y})}(\mathbf{z}). \quad (5.2)$$

Let \mathbf{y}_0 be the first configuration of \mathbf{Y} . It follows from Steps 8 and 10 that $\mathcal{PT}_1(\mathbf{y}_0) = 1$. Therefore,

$$\mathcal{PT}(\mathbf{y}_0, \mathbf{z}) = \mathcal{PT}_1(\mathbf{y}_0) \cdot \mathcal{PT}_2(\mathbf{z}) = \mathcal{PT}_2(\mathbf{z}), \quad (5.3)$$

and according to Eq. 5.2, it means that

$$\mathcal{PT}^{R(\mathbf{Y}=\mathbf{y}_0)}(\mathbf{z}) = \mathcal{PT}_2(\mathbf{z}). \quad (5.4)$$

For any other configuration $\mathbf{y}_j \in \Omega_{\mathbf{Y}}$, we can write

$$\mathcal{PT}(\mathbf{y}_j, \mathbf{z}) = \mathcal{PT}_1(\mathbf{y}_j) \cdot \mathcal{PT}_2(\mathbf{z}) = \mathcal{PT}^{R(\mathbf{Y}=\mathbf{y}_j)}(\mathbf{z}),$$

and using Eq. 5.4,

$$\mathcal{PT}^{R(\mathbf{Y}=\mathbf{y}_j)}(\mathbf{z}) = \mathcal{PT}_1(\mathbf{y}_j) \cdot \mathcal{PT}^{R(\mathbf{Y}=\mathbf{y}_0)}(\mathbf{z}) \Rightarrow \mathcal{PT}_1(\mathbf{y}_j) = \frac{\mathcal{PT}^{R(\mathbf{Y}=\mathbf{y}_j)}(\mathbf{z})}{\mathcal{PT}^{R(\mathbf{Y}=\mathbf{y}_0)}(\mathbf{z})},$$

which corresponds to the calculation in Step 8. Finally, notice that the re-scaling

5.3 Fast factorisation of probability trees

in Steps 13 and 14 do not affect this result, as

$$\begin{aligned}
 \mathcal{PT}_1 \cdot \frac{s_{\mathcal{PT}}}{s_{\mathcal{PT}_1}} \cdot \mathcal{PT}_2 \cdot \frac{1}{s_{\mathcal{PT}_2}} &= \mathcal{PT}_1 \cdot \mathcal{PT}_2 \cdot \frac{\sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z})}{(\sum_{\mathbf{y}} t_1(\mathbf{y}))(\sum_{\mathbf{z}} t_2(\mathbf{z}))} \\
 &= \mathcal{PT}_1 \cdot \mathcal{PT}_2 \cdot \frac{\sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z})}{\sum_{\mathbf{y}} t_1(\mathbf{y})(\sum_{\mathbf{z}} t_2(\mathbf{z}))} \\
 &= \mathcal{PT}_1 \cdot \mathcal{PT}_2 \cdot \frac{\sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z})}{\sum_{\mathbf{y}} \sum_{\mathbf{z}} t_1(\mathbf{y})t_2(\mathbf{z})} \\
 &= \mathcal{PT}_1 \cdot \mathcal{PT}_2 \cdot \frac{\sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z})}{\sum_{\mathbf{y}} \sum_{\mathbf{z}} t(\mathbf{y}, \mathbf{z})} \\
 &= \mathcal{PT}_1 \cdot \mathcal{PT}_2.
 \end{aligned}$$

Therefore, it follows that $\mathcal{PT} = \mathcal{PT}_1 \cdot \mathcal{PT}_2$ and thus condition (3) in Def. 4 holds.

□

The next example illustrates the way in which Alg. 16 carries out the decomposition.

Example 47 Let \mathcal{PT} be the tree in Fig. 5.3. We will use Alg. 16 to decompose it with respect to variable X_2 . The first action is actually performed in Step 5, where a configuration is selected for those variables in the tree, different from the one with respect to which we are going to decompose. In this case, a configuration for (X_1, X_3) has to be selected. Assume we choose the configuration $(X_1 = 0, X_3 = 0)$. Step 6 requires the computation of tree $\mathcal{PT}^{R(X_1=0, X_3=0)}$, which is a tree that only contains variable X_2 , and whose leaves are $\alpha_0 = 0.1$ for $X_2 = 0$ and $\alpha_1 = 0.2$ for $X_2 = 1$. In Step 8, the β coefficients are computed:

$$\beta_0 = \frac{\alpha_0}{\alpha_0} = \frac{0.1}{0.1} = 1, \quad \beta_1 = \frac{\alpha_1}{\alpha_0} = \frac{0.2}{0.1} = 2.$$

Next, a tree is constructed in Step 10 with X_2 as unique variable and β_0 and β_1 as leaves. It corresponds to the leftmost tree in Fig. 5.4. Step 11 computes the second tree in the decomposition, by restricting \mathcal{PT} to $(X_2 = 0)$. This tree is shown in the right side of Fig. 5.4. Finally, Steps 12, 13 and 14 re-scale the values in the decomposition in order to guarantee that the total mass is the same as in the original tree. The re-scaled decomposition is shown in Fig. 5.5.

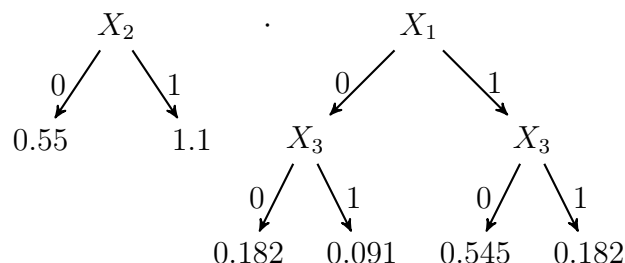


Figure 5.5: Modified decomposition obtained from Fig. 5.4 after making the total mass be equal to the one in the original tree in Fig. 5.3.

An important feature of this decomposition scheme is its low complexity. This is specially interesting because it allows fast factorisation to be included in other algorithms (for instance, inference algorithms) without increasing the complexity order.

Lemma 1 *The complexity of Alg. 16 is linear in the size of the input tree, in the worst case.*

Proof 2 *The complexity is determined by Steps 6, 11 and 12, which compute the restriction of the input tree and the sum of the input and output trees. The restriction operation is, in the worst case, linear in the size of the input tree as it can be obtained just by visiting all the leaves in the tree and keeping those consistent with the restricting configuration. The sum is also linear as it requires to visit all the leaves in the tree. \square*

5.3.2 Approximate Decomposition

As shown in Prop. 1, Alg. 16 finds the correct factorisation of a tree that is actually decomposable. However, it may happen that a tree is not decomposable with respect to a set of variables, but perhaps it is possible to factorise it in such a way that the product of the resulting trees is not far away from the original one. The next example illustrates this fact.

Example 48 *Consider the probability tree in Fig. 5.3, but with the first two leaves equal to 0.11 and 0.19 instead of 0.1 and 0.2 respectively. After such modification,*

5.3 Fast factorisation of probability trees

the tree is no longer decomposable. If we apply Alg. 16 to factorise such tree with respect to variable X , we obtain the decomposition in Fig. 5.6. Notice that this decomposition is actually an approximation. In fact, if we multiply again the two trees, the result is the tree in Fig. 5.7, which is not exactly the same as the original one.

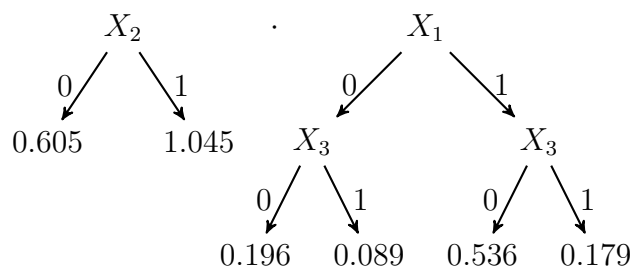


Figure 5.6: Decomposition of the tree in Fig. 5.7, using Alg. 16.

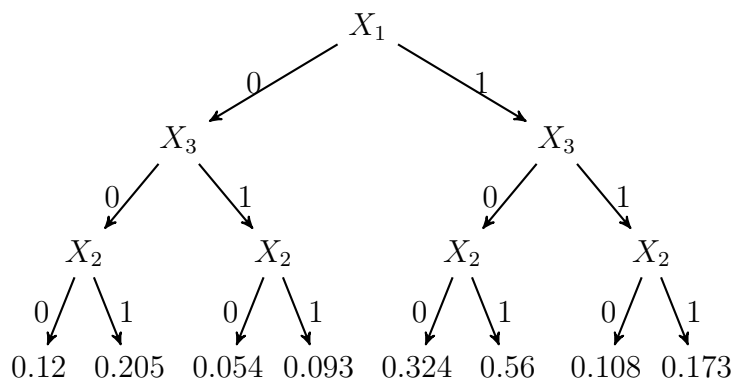


Figure 5.7: Result of multiplying the two trees in Fig. 5.6.

The concept of approximate factorisation has been studied previously [68], but the limitation of that approach is the same as the one described in the case of exact classical factorisation. Here we analyse how to extend the factorisation procedure explained above to the case in which a tree is not exactly decomposable.

If a tree is not exactly decomposable with respect to a given set of variables, we propose to use the *extended Kullback-Leibler divergence* [70] as a basis to

5.3 Fast factorisation of probability trees

determine how far from exact factorisation a given decomposition is. This divergence measure is an extension of Kullback-Leibler divergence [71] to unnormalised potentials. It is an important feature, as in general we deal with unnormalised potentials (i.e., potentials that do not sum up to one), especially after applying Alg. 16. For two unnormalised potentials ϕ_1 and ϕ_2 , it is defined as

$$eKL(\phi_1, \phi_2) = \sum_x \phi_1(x) \log \frac{\phi_1(x)}{\phi_2(x)} + \sum_x (\phi_2(x) - \phi_1(x)). \quad (5.5)$$

From now on, whenever we mention the eKL divergence between probability trees we understand the eKL divergence between the potentials represented by those trees. The key result is given in the next theorem, which provides an upper bound of the eKL divergence for a given decomposition.

Theorem 1 *Let \mathcal{PT} be a probability tree to be decomposed with respect to a set of variables \mathbf{Y} . Let \mathbf{X} be the set of variables for which \mathcal{PT} is defined. Let be $\mathbf{Z} = \mathbf{X} \setminus \mathbf{Y}$. Let \mathcal{PT}_1 and \mathcal{PT}_2 be the output of algorithm **Factorise**(\mathcal{PT}, \mathbf{Y}) (Alg. 16). Then it holds that*

$$eKL(\mathcal{PT}, \mathcal{PT}_1 \otimes \mathcal{PT}_2) \leq \sum_{\mathbf{x}} t(\mathbf{x}) \log t(\mathbf{x}) - \left(\sum_{\mathbf{x}} t(\mathbf{x}) \right) \left(\sum_{\mathbf{y}: t_1(\mathbf{y}) \leq 1} \log t_1(\mathbf{y}) + \sum_{\mathbf{z}} \log t_2(\mathbf{z}) \right), \quad (5.6)$$

where t, t_1 and t_2 are the potentials represented by trees $\mathcal{PT}, \mathcal{PT}_1$ and \mathcal{PT}_2 respectively.

Proof 3

$$\begin{aligned}
 eKL(\mathcal{PT}, \mathcal{PT}_1 \otimes \mathcal{PT}_2) &= \sum_{\mathbf{x}} t(\mathbf{x}) \log \frac{t(\mathbf{x})}{t_1(\mathbf{y})t_2(\mathbf{z})} + \sum_{\mathbf{x}} (t_1(\mathbf{y})t_2(\mathbf{z}) - t(\mathbf{x})) \\
 &= \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \log \frac{t(\mathbf{y}, \mathbf{z})}{t_1(\mathbf{y})t_2(\mathbf{z})} + \sum_{\mathbf{y}, \mathbf{z}} (t_1(\mathbf{y})t_2(\mathbf{z}) - t(\mathbf{y}, \mathbf{z})) \\
 &= \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) (\log t(\mathbf{y}, \mathbf{z}) - \log t_1(\mathbf{y}) - \log t_2(\mathbf{z})) \\
 &\quad + \sum_{\mathbf{y}, \mathbf{z}} (t_1(\mathbf{y})t_2(\mathbf{z}) - t(\mathbf{y}, \mathbf{z})) \\
 &= \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \log t(\mathbf{y}, \mathbf{z}) - \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \log t_1(\mathbf{y}) \\
 &\quad - \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \log t_2(\mathbf{z}) \\
 &\quad + \sum_{\mathbf{y}, \mathbf{z}} t_1(\mathbf{y})t_2(\mathbf{z}) - \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}).
 \end{aligned}$$

Now, let us denote by t_1^* and t_2^* the potentials corresponding to the trees \mathcal{PT}_1 and \mathcal{PT}_2 in Steps 10 and 11 of Alg. 16, i.e., before re-scaling the trees. Then,

$$\begin{aligned}
 \sum_{\mathbf{y}, \mathbf{z}} t_1(\mathbf{y})t_2(\mathbf{z}) &= \sum_{\mathbf{y}, \mathbf{z}} t_1^*(\mathbf{y}) \frac{s_{\mathcal{PT}}}{s_{\mathcal{PT}_1}} t_2^*(\mathbf{z}) \frac{1}{s_{\mathcal{PT}_2}} \\
 &= \frac{s_{\mathcal{PT}}}{s_{\mathcal{PT}_1} s_{\mathcal{PT}_2}} \sum_{\mathbf{y}, \mathbf{z}} t_1^*(\mathbf{y}) t_2^*(\mathbf{z}) \\
 &= \frac{s_{\mathcal{PT}}}{s_{\mathcal{PT}_1} s_{\mathcal{PT}_2}} \left(\sum_{\mathbf{y}} t_1^*(\mathbf{y}) \right) \left(\sum_{\mathbf{z}} t_2^*(\mathbf{z}) \right) \\
 &= \frac{s_{\mathcal{PT}} s_{\mathcal{PT}_1} s_{\mathcal{PT}_2}}{s_{\mathcal{PT}_1} s_{\mathcal{PT}_2}} = s_{\mathcal{PT}} = \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}),
 \end{aligned}$$

where $s_{\mathcal{PT}}$, $s_{\mathcal{PT}_1}$ and $s_{\mathcal{PT}_2}$ are defined in Step 12 of Alg. 16. Hence,

5.3 Fast factorisation of probability trees

$$\begin{aligned}
eKL(\mathcal{PT}, \mathcal{PT}_1 \otimes \mathcal{PT}_2) &= \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \log t(\mathbf{y}, \mathbf{z}) \\
&\quad - \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \log t_1(\mathbf{y}) - \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \log t_2(\mathbf{z}) \\
&= \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \log t(\mathbf{y}, \mathbf{z}) - \sum_{\mathbf{y}} \left((\log t_1(\mathbf{y})) \sum_{\mathbf{z}} t(\mathbf{y}, \mathbf{z}) \right) \\
&\quad - \sum_{\mathbf{z}} \left((\log t_2(\mathbf{z})) \sum_{\mathbf{y}} t(\mathbf{y}, \mathbf{z}) \right).
\end{aligned}$$

Notice that, since the values in t are not negative, it holds that

$$\sum_{\mathbf{z}} t(\mathbf{y}, \mathbf{z}) \leq \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \quad (5.7)$$

and

$$\sum_{\mathbf{y}} t(\mathbf{y}, \mathbf{z}) \leq \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}). \quad (5.8)$$

Furthermore, the values in t_2 are guaranteed to be lower than 1, and therefore their log is a negative number. Thus, we can write

$$\begin{aligned}
eKL(\mathcal{PT}, \mathcal{PT}_1 \otimes \mathcal{PT}_2) &\leq \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \log t(\mathbf{y}, \mathbf{z}) - \sum_{\mathbf{y}: t_1(\mathbf{y}) \leq 1} \left((\log t_1(\mathbf{y})) \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \right) \\
&\quad - \sum_{\mathbf{z}} \left((\log t_2(\mathbf{z})) \sum_{\mathbf{y}, \mathbf{z}} t(\mathbf{y}, \mathbf{z}) \right) \\
&= \sum_{\mathbf{x}} t(\mathbf{x}) \log t(\mathbf{x}) \\
&\quad - \left(\sum_{\mathbf{x}} t(\mathbf{x}) \right) \left(\sum_{\mathbf{y}: t_1(\mathbf{y}) \leq 1} \log t_1(\mathbf{y}) + \sum_{\mathbf{z}} \log t_2(\mathbf{z}) \right),
\end{aligned}$$

which completes the proof. □

Note that, if a decomposition is exact, then $eKL(\mathcal{PT}, \mathcal{PT}_1 \otimes \mathcal{PT}_2)$ is equal to 0, and the further a decomposition is to the exact one, the higher the value of

5.3 Fast factorisation of probability trees

the divergence reaches [70]. Observe also that the upper bound given in Eq. 5.6, actually depends on the specific decomposition through the term

$$S = \sum_{\mathbf{y}:t_1(\mathbf{y})\leq 1} \log t_1(\mathbf{y}) + \sum_{\mathbf{z}} \log t_2(\mathbf{z}),$$

which suggests that S could be used as a measure of the degree of decomposability of a tree \mathcal{PT} with respect to \mathbf{Y} . The computation of S is rather fast, as it requires a time linear on the size of \mathcal{PT}_1 and \mathcal{PT}_2 . Hence, we use the reasoning above to formally define the degree of decomposability of a tree with respect to a given set of variables, called the *factorisation degree*, as follows.

Definition 5 *Let \mathcal{PT} be a probability tree. Let \mathbf{X} be the set of variables for which \mathcal{PT} is defined, and $\mathbf{Y} \subset \mathbf{X}$. Let $\mathbf{Z} = \mathbf{X} \setminus \mathbf{Y}$. Let \mathcal{PT}_1 and \mathcal{PT}_2 be the output of algorithm **Factorise**(\mathcal{PT}, \mathbf{Y}), described in Alg. 16. We define the factorisation degree of \mathcal{PT} with respect to \mathbf{Y} as*

$$\text{fd}(\mathcal{PT}, \mathbf{Y}) = \sum_{\mathbf{y}:t_1(\mathbf{y})\leq 1} \log t_1(\mathbf{y}) + \sum_{\mathbf{z}} \log t_2(\mathbf{z}), \quad (5.9)$$

where t_1 and t_2 are the potentials represented by \mathcal{PT}_1 and \mathcal{PT}_2 respectively.

The factorisation degree defined above provides a heuristic way to choose the variable or set of variables with respect to which a tree can be decomposed, producing the lowest error in terms of eKL divergence. It is only heuristic since what is minimised is not the eKL divergence itself, but an upper bound, as given in Theorem 1.

This heuristic suggests a way to control the tradeoff between accuracy and complexity in approximate inference algorithms for Bayesian networks, namely by establishing a threshold of factorisation degree, and decomposing those trees for which there is a set of variables whose factorisation degree surpasses such threshold. Notice that, according to Definition 5 and Theorem 1, the higher the factorisation degree, the lower the bound above the divergence between the original and decomposed representation of the tree. Therefore, by setting a lower threshold, more trees are potentially decomposed, and therefore the complexity of the inference problem is reduced as it has to deal with smaller trees, in exchange of losing accuracy.

5.3 Fast factorisation of probability trees

The effectiveness of this approach depends on whether or not the kind of regularity characterised in Definition 4 is actually found in real-world problems. In order to investigate this fact, we have analysed four real-world networks commonly used as a benchmark for approximate inference algorithms. These networks are called Munin [72], Andes [73], Barley [66] and Water [74] (details of the networks can be found in Appendix 7.2.4). The analysis consisted of decomposing their conditional distributions according to the variable with highest factorisation degree, and measuring the root mean squared error between the original probability tree and its decomposition. The results are shown in Fig. 5.8, which is a beanplot [75] that displays the empirical distribution of the errors obtained for the four networks.

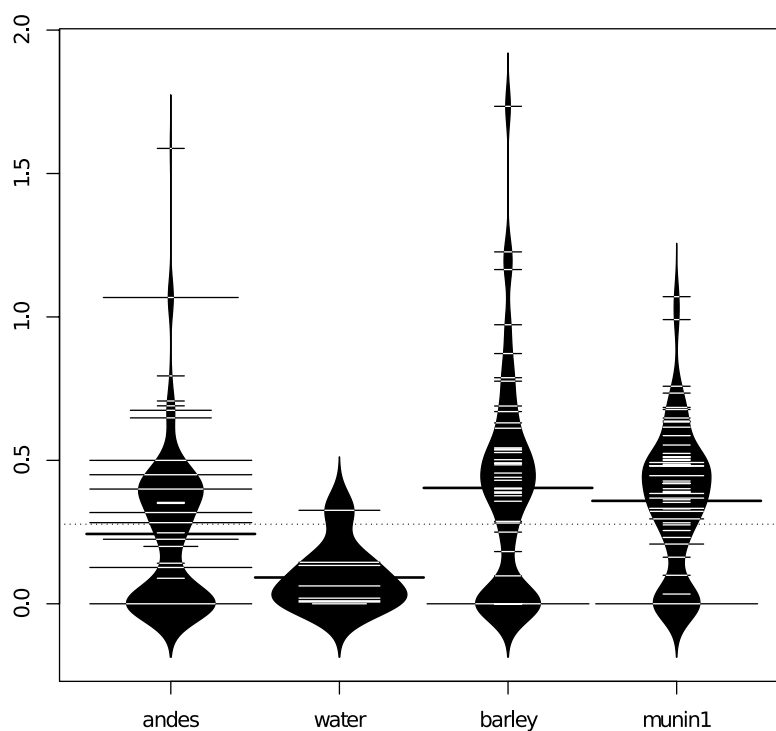


Figure 5.8: Decomposability in four real-world examples. The plot shows the distribution of the potentials that, after decomposing, attained different levels of error.

5.4 Inference with factorised representations

The beanplot consists of a density trace for each batch mirrored to form a polygon shape, and it details the averages of each batch using a slightly bolder line and the overall average using a dotted line. It can be seen how a high amount of distributions can be factorised introducing a very low error, close to zero, specially in the case of the Water network. Of course there are many potentials for which it is not possible to carry out the decomposition without introducing a large error, but this is a common fact in methods for approximating probability trees. For instance, consider the tree in Fig. 5.9. It is clear that it can no longer be approximated using tree pruning, unless a high error in the approximation is admitted.

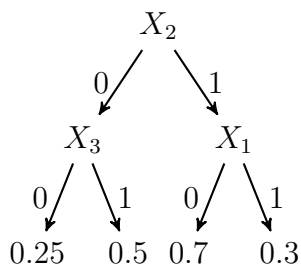


Figure 5.9: Potential $\phi(X_1, X_2, X_3)$ represented as a pruned probability tree.

5.4 Inference with factorised representations

In Section 3.7.1.2 we discussed how a reduction in the size of the initial probability distributions in the network by pruning probability trees can simplify the inference problem, illustrating the workflow of this methodology with the **Prune_VE**(**B**,**E**,**e**, α) algorithm (Alg. 6). The parameter α defines the threshold for the prune of the trees, controlling in this way the accuracy of the approximations. The concepts shown in this chapter about factorisation of probability trees can be applied in a similar fashion to try to ease the computational complexity of the inference.

We define the algorithm **Factorise_VE**(**B**,**E**,**e**, δ) (Alg. 17) that carries out the approximation by factorising the initial distributions in the network for which the best factorisation degree obtained by any of its variables surpasses a given

5.4 Inference with factorised representations

threshold δ . Algorithm 17 can be seen as a wrapper of the specific Variable Elimination algorithm for working with probability trees (Alg. 5) explained in Chapter 3.7.1.2. The reason is that, just like algorithm **Prune_VE**($\mathbf{B}, \mathbf{E}, \mathbf{e}, \alpha$) (Alg. 6), our algorithm needs to preprocess the conditional probability distributions in the network in order to factorise them prior to the inference.

Algorithm 17 takes a Bayesian network whose conditional probability distributions are represented using probability trees (line 3 of Alg. 17). The first step consists on decomposing each and every one of the probability trees (lines 5 to 16 of Alg. 17). For each distribution, we first compute the variable that gives a higher factorisation degree according to Eq. 5.9 (line 8 of Alg. 17). If the higher obtained factorisation degree surpasses our threshold δ , (line 9 of Alg. 17) then we proceed to decompose the distribution (line 10 of Alg. 17). Once all the distributions are preprocessed, we call Alg. 5 with all the variables that are not instantiated (lines 17 to 21 of Alg. 17). Note that the deletion ordering that we follow is established by the position of the variables in vector \mathbf{X} .

```

1 Factorise_VE(B,E,e, $\delta$ )
   Input: A Bayesian network (B) and an observation (E = e). A
           factorisation degree threshold ( $\delta$ ) for decomposing the initial
           distributions.
   Output: The posterior distribution of all the unobserved variables in the
           network, given E = e.
2 Let X be the variables in B.
3 Let PT =  $\{\mathcal{PT}_i, i = 1, \dots, n\}$ , be the probability trees representing the
   conditional distributions in B.
4 PT'  $\leftarrow \emptyset$ .
5 foreach  $\mathcal{PT} \in \mathbf{PT}$  do
6   | Let  $Y_1, \dots, Y_k$  be the variables in  $\mathcal{PT}$ .
7   | Compute  $\text{fd}(\mathcal{PT}, Y_i), i = 1, \dots, k$  according to Eq. 5.9.
8   |  $Y \leftarrow \arg \max_{i=1, \dots, k} \text{fd}(\mathcal{PT}, Y_i)$ .
9   | if  $\text{fd}(\mathcal{PT}, Y) > \delta$  then
10  | | F  $\leftarrow$  Factorise( $\mathcal{PT}, Y$ ).
11  | end
12  | else
13  | | F  $\leftarrow \emptyset$ .
14  | end
15  | PT'  $\leftarrow \mathbf{PT}' \cup \mathbf{F}$ .
16 end
17 R =  $\emptyset$ .
18 foreach  $W \in \mathbf{X} \setminus \mathbf{E}$  do
19  |  $\mathcal{PT} \leftarrow$  Variable_Elimination(X,W,E,e,PT') (using Alg. 5).
20  | R  $\leftarrow \mathbf{R} \cup \{\mathcal{PT}\}$ .
21 end
22 return R.

```

Algorithm 17: Pseudo-code of the variable elimination algorithm with factorisation of the initial distributions.

5.5 Experimental evaluation

The setting we have established is simple, in order to facilitate the evaluation of the real impact of using factorisation of probability trees as a means to control the level of approximation, and also to be able to compare this approach with

the approximation method based on tree pruning, as explained in the previous section.

Using both algorithms discussed above (**Factorise_VE**($\mathbf{B}, \mathbf{E}, \mathbf{e}, \delta$) (Alg. 17) and **Prune_VE**($\mathbf{B}, \mathbf{E}, \mathbf{e}, \alpha$) (Alg. 6)) we carried out a series of tests in order to check that parameter δ is appropriate for controlling the complexity of the inference process, in a similar way as parameter α .

The tests consisted of running both algorithms over the four real-world networks mentioned in Sec. 5.3.2 (Munin, Andes, Barley and Water) with different values of α and δ , in order to compute the posterior distribution for each variable of the networks. For each run, we measured the execution time for the whole process, the error in the estimation of the posterior probabilities and the average and maximum sizes of the probability trees handled during the inference process. The error was measured using Fertig and Mann's divergence [51], as explained in Chapter 1.

The results of the experiments are summarised in Figs. 5.10 to 5.15. In general, it can be said that parameter δ can be used to control the approximation level in a similar way as parameter α (Figs. 5.10 and 5.11). The only anomaly is detected in the case of network Munin (see the bottom part of Fig. 5.10), where for the first point, execution time is higher than for others with lower errors. However, the same behaviour can be observed for the α parameter in this case. It can be seen that for networks Barley and Munin, the convergence to low error values is reached more quickly by algorithm **Factorise_VE**, while the contrary happens for networks Andes and Water.

5.5 Experimental evaluation

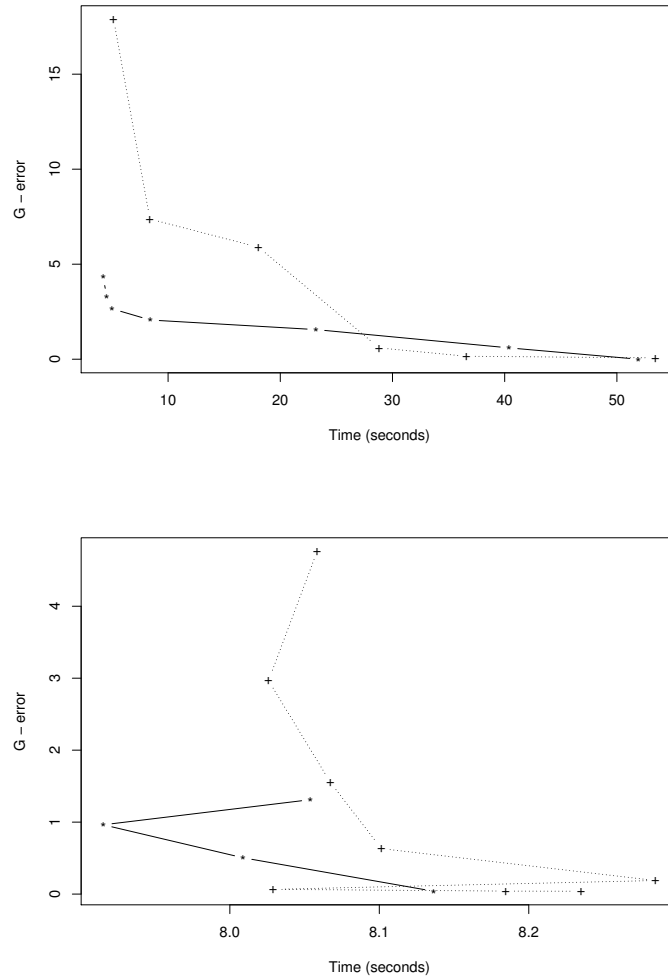


Figure 5.10: Error vs. time for the Barley (top) and Munin (bottom,) networks. The solid line corresponds to method **Factorise_VE** and the dotted one to **Prune_VE**.

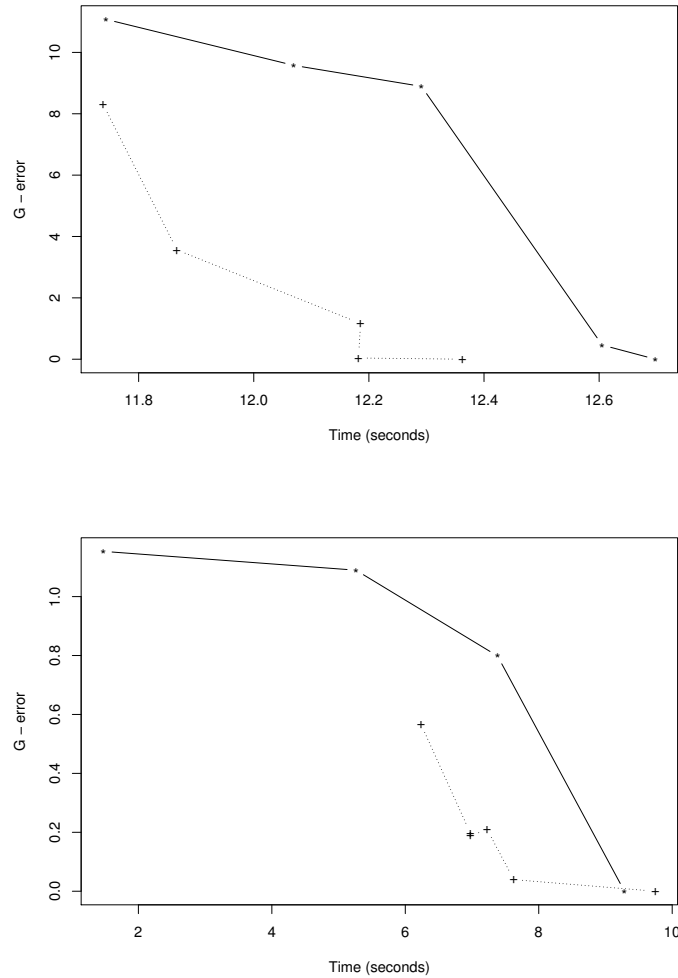


Figure 5.11: Error vs. time for the Andes (top) and Water (bottom) networks. The solid line corresponds to method **Factorise_VE** and the dotted one to **Prune_VE**.

Regarding the size of the potentials involved in the calculations during the inference process, the experiments show how the average size is lower for algorithm **Factorise_VE** (see Figs. 5.12 and 5.14). However, the maximum size of such potentials is lower when using algorithm **Prune_VE** (see Figs. 5.13 and 5.15).

5.5 Experimental evaluation

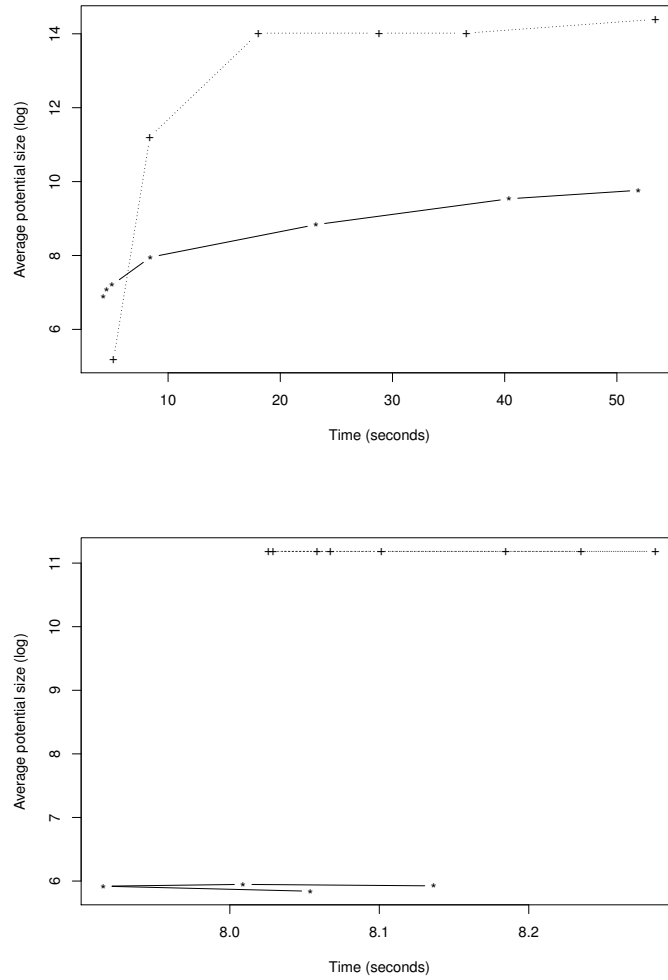


Figure 5.12: Average potential size (in logarithmic scale) vs. time for the Barley (top) and Munin (bottom) networks. The solid line corresponds to method **Factorise_VE** and the dotted one to **Prune_VE**.

5.5 Experimental evaluation

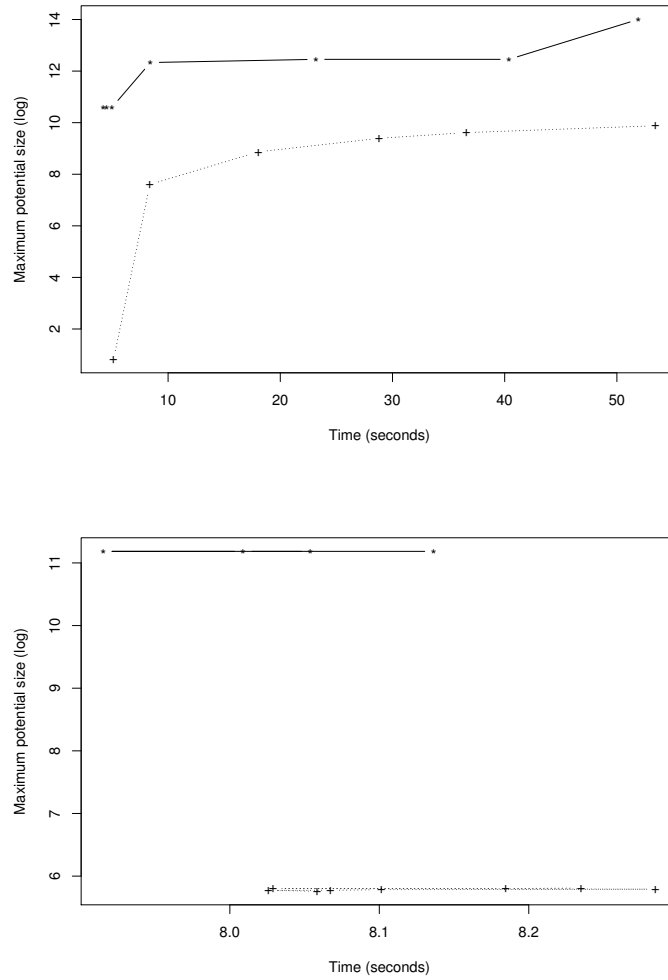


Figure 5.13: Maximum potential size (in logarithmic scale) vs. time for the Barley (top) and Munin (bottom) networks. The solid line corresponds to method **Factorise_VE** and the dotted one to **Prune_VE**.

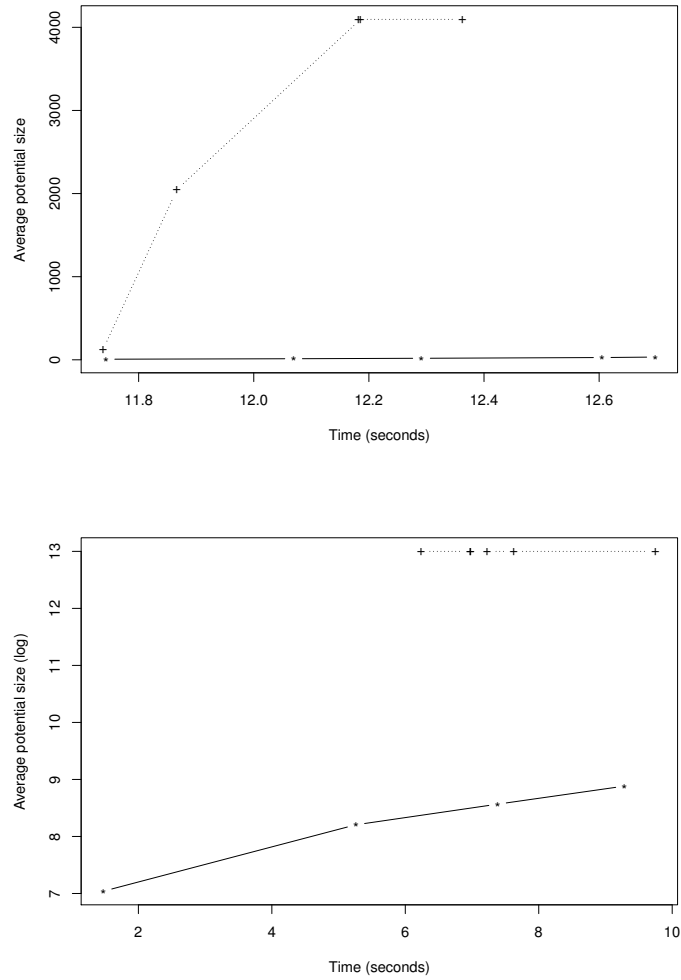


Figure 5.14: Average potential size vs. time for the Andes (top) and Water (bottom) networks. The solid line corresponds to method **Factorise_VE** and the dotted one to **Prune_VE**.

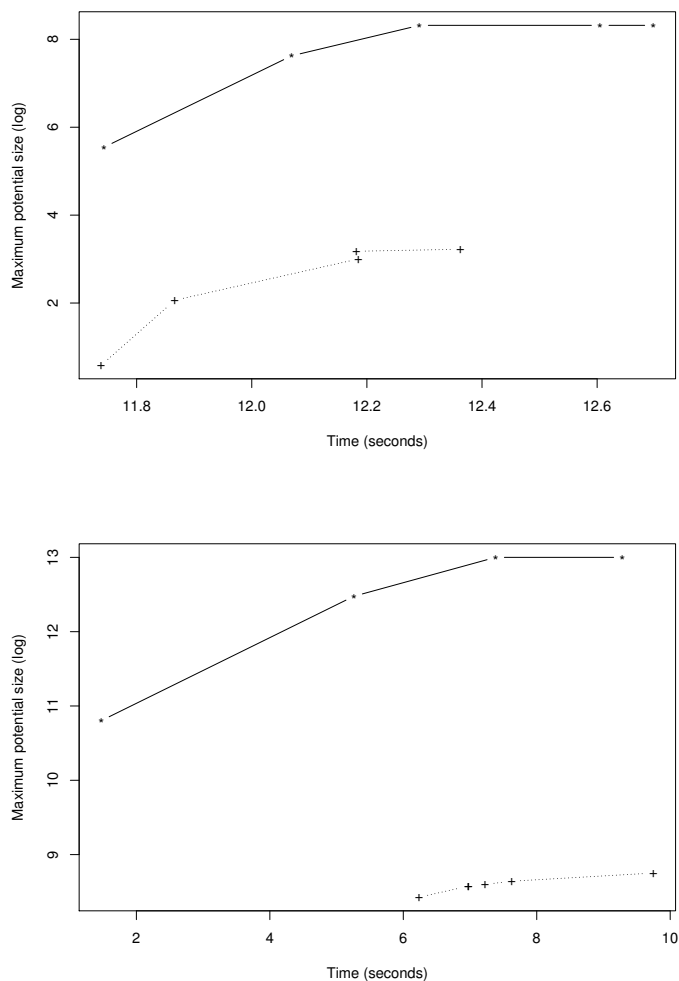


Figure 5.15: Maximum potential size (in logarithmic scale) vs. time for the Andes (top) and Water (bottom) networks. The solid line corresponds to method **Factorise_VE** and the dotted one to **Prune_VE**.

5.6 Conclusions and Future Work

In this chapter we have introduced a new and fast procedure for factorising probability trees. An important feature of the proposed algorithm is related to its capability for obtaining optimal decompositions in case that the tree is actually

decomposable, in the sense that the decomposition is as compact as possible. We have also shown that the decomposition can be carried out even if the tree does not really contain proportional subtrees, in which case the obtained factorisation will be approximate.

In order to deal with the degree of approximation of the possible decompositions of a potential we have introduced a measure called the *factorisation degree*, that provides a heuristic to rank the variables in the domain of a potential according to the accuracy of the decompositions that they induce. The computation of such measure is fast enough as to be included in probabilistic inference algorithms, where computing time is a crucial issue.

We have analysed the behaviour of the fast factorisation algorithm as a means of controlling the tradeoff between accuracy and complexity. In the networks tested in the experiments, the factorisation degree performed in a similar way as tree pruning.

Possible applications of the concept of fast factorisation go beyond probabilistic inference algorithms. We define as a future line of research the application of the concepts developed in this Chapter within algorithms for learning structured representations of probabilistic potentials, such as Recursive Probability Trees and their possible variations.

Part IV

Learning

Chapter 6

Learning Recursive Probability Trees

We discuss in this chapter the process of learning RPTs. First we discuss the possibility of transforming a probabilistic potential into an RPT following a top-down greedy approach. This algorithm looks for context-specific independencies along with factorisations within the original potential to build a small RPT structure. This methodology is later adjusted to cope with the learning process from data, where we follow a very similar approach to learn a small RPT that encodes an approximation of the potential represented by the database.

Also in this chapter we propose an algorithm for learning RPTs from data that follows a score-and-search approach. This algorithm starts from an initial RPT structure and builds a group of neighbours using a set of operators. The best of this neighbours is chosen to continue the search from it, until a stop criterion is reached.

The chapter ends with an experimental evaluation of the three proposed algorithms, along with the main conclusions and thoughts for future related works.

6.1 Learning RPTs from probabilistic potentials

This section starts by describing our proposal for building an RPT from another probabilistic potential (a CPT, for instance). The problem of finding a minimal RPT is not trivial, as we will discuss in Section 6.1.4, where we prove that it

6.1 Learning RPTs from probabilistic potentials

is in fact NP-hard. Hence, we propose to obtain an RPT in a greedy way, following a heuristic designed for selecting Split nodes that are likely to reduce the dependencies among the remaining variables. In this way we intend to increase the possibilities of finding multiplicative factorisations, which constitute the basis for obtaining RPTs of small size. Furthermore, the algorithm proposed in this chapter can be used to obtain approximations of the original potential, when the size of the exact representation through an RPT is too large.

The proposed algorithm is modified in Section 6.1.5, where we learn RPTs from a database. In this case, we measure each considered RPT using a Bayesian score, and we apply a different approach when normalising, using the Laplace smoothing.

6.1.1 Motivation

In general, inference algorithms become less efficient as the number of variables and general complexity of the model grow, as this implies operations with big structures that require large storage space and increase the computational processing time. The ability of generating a Recursive Probability Tree from a probabilistic potential can be incorporated to the inference process, when the management of big amounts of data becomes an issue.

6.1.2 Building an RPT from a probabilistic potential

We have designed an algorithm oriented to the detection of context-specific independencies and also multiplicative factorisations. Context-specific independencies are sought following an approach similar to the one used for constructing probability trees [76]. It is based on selecting variables for *Split* nodes according to their information gain, as it is done by Quinlan when constructing decision trees [77].

Quinlan's ID3 algorithm [77] builds a decision tree from a set of examples. A decision tree represents a sequential procedure for deciding the class membership of a given instance of the attributes of the problem. That is, the leaves of the decision tree give us the class for a given instance of the attributes. ID3 builds a decision tree in a greedy way, by choosing a good test attribute to refine the

6.1 Learning RPTs from probabilistic potentials

actual structure. To determine which attribute should be the test attribute for a leaf node of the tree, the algorithm applies an information-theoretic measure over all the possible attributes. This information measure gives an idea of the gain of information by partitioning a leaf node in the tree with an attribute.

In the case of probability trees, the algorithm is very similar [58], but the information measure is different, because each leaf in a decision tree represents a class, while in a probability tree, each leaf represents a probability value. Then, we need a measure particularly adapted to probabilities.

For our algorithm, the general idea is to look for context-specific independencies following a top-down approach, choosing at each step a variable that maximizes the information gain. This allows an ordering of the variables that places the most informative ones at the upper nodes in the tree.

Regarding multiplicative decompositions, the basic idea is to detect groups of variables according to their *mutual information*. The mutual information is a quantity that measures the mutual dependence of two variables: it measures how much knowing one of the variables reduces uncertainty about the other.

A threshold ε is defined in order to control when a pair of variables are considered independent based on their mutual information. The groups obtained are later used to get the potentials making up the multiplicative decomposition.

6.1.2.1 Description of the algorithm

The starting point is a potential ϕ defined over a set of variables \mathbf{X} . The goal is to find an RPT representing ϕ , denoted from now on as \mathcal{RT} .

We defined sum_{ϕ} in Chapter 3 (Section 3.5.1.2) as the sum of all the values in a potential, so the sum of values consistent with a given configuration $\mathbf{x}_{\mathbf{J}}$ of a subset of the potential's variables can be written as:

$$sum_{\phi R(\mathbf{x}_{\mathbf{J}})} = \sum_{\mathbf{z} \in \Omega_{\mathbf{z}}} \phi(\mathbf{z}, \mathbf{y})$$

The main algorithm used for computing \mathcal{RT} is referred as **potentialFactorisation** (Alg. 18). The simplified workflow of this algorithm is represented in Fig. 6.1, where we can see how it makes use of several auxiliary algorithms in

6.1 Learning RPTs from probabilistic potentials

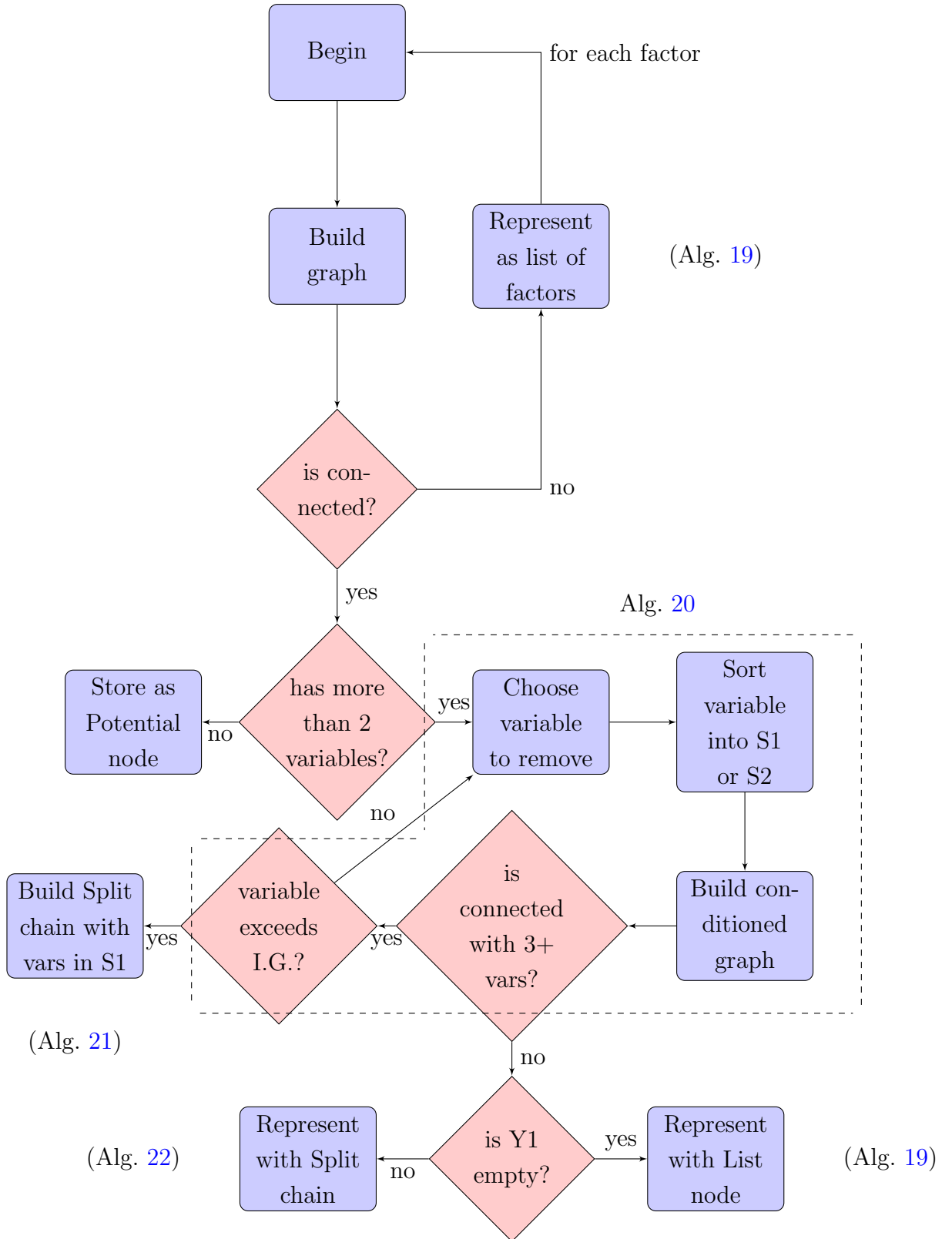


Figure 6.1: Alg. 18 workflow

6.1 Learning RPTs from probabilistic potentials

different parts of the process. In the following we give a detailed explanation of every one of them.

Algorithm **potentialFactorisation** (Alg. 18) operates with an auxiliary graph $G_\phi = (\mathbf{X}, E)$ with vertex set \mathbf{X} and link set E . A link $X_i - X_j$ belongs to E only if the mutual information (used as a measure of dependence) between both variables, denoted as $I(X_i, X_j)$, exceeds a given threshold $\varepsilon > 0$, with:

$$I(X_i, X_j) = \sum_{\substack{x_i \in \Omega_{X_i} \\ x_j \in \Omega_{X_j}}} \phi(x_i, x_j) \log \frac{\phi(x_i, x_j)}{\phi(x_i)\phi(x_j)}, \quad (6.1)$$

$$\phi(x_i, x_j) = \frac{\phi^{\downarrow X_i, X_j}(x_i, x_j)}{sum_\phi}, \quad (6.2)$$

$$\phi(x_i) = \frac{\phi^{\downarrow X_i}(x_i)}{sum_\phi}, \quad \phi(x_j) = \frac{\phi^{\downarrow X_j}(x_j)}{sum_\phi}, \quad (6.3)$$

where $\phi^{\downarrow X_i, X_j}$, $\phi^{\downarrow X_i}$ and $\phi^{\downarrow X_j}$ are the marginals of ϕ over the sets $\{X_i, X_j\}$, $\{X_i\}$ and $\{X_j\}$ respectively. Links $X_i - X_j$ are weighted with $I(X_i, X_j)$.

Algorithm 18 computes G_ϕ in line 4. Line 6 is focused on analysing G_ϕ , searching for connected components. There are two possible scenarios to consider: (i) G_ϕ is decomposed into n components $\mathbf{C} = \{\mathbf{C}_1, \dots, \mathbf{C}_n\}$ and (ii) G_ϕ contains a single connected component. Both of them will be handled, respectively, with auxiliary algorithms **multiplicativeFactorisation** (line 11) and **contextSpecificFactorisation** (line 14) respectively. These algorithms will be described in the next sections.

```

1 potentialFactorisation( $\phi$ )
  Input: A potential  $\phi$ 
  Output:  $\mathcal{RT}$ , an RPT for  $\phi$ 
2 begin
3   // Step 1: compute  $G_\phi$ 
4    $G_\phi \rightarrow$  graph for variables dependencies in  $\phi$ 
5   // Step 2: graph analysis
6    $G_\phi$  analysis looking for connected components
7   // Several scenarios are possible according to  $G_\phi$ 
8   // components
9   if  $G_\phi$  is partitioned into components  $\mathbf{C} = \{\mathbf{C}_1 \dots \mathbf{C}_n\}$  then
10    // Step 3: multiplicative factorisation
11     $\mathcal{RT} \leftarrow$  multiplicativeFactorisation( $\phi, \mathbf{C}$ )
12  else
13    // Only one component: decomposition with Step 4
14     $\mathcal{RT} \leftarrow$  contextSpecificFactorisation( $\phi, G_\phi$ )
15  end
16  return  $\mathcal{RT}$ 
17 end

```

Algorithm 18: Main body of potential factorisation algorithm

6.1.2.2 Computing multiplicative factorisations

When G_ϕ is partitioned as $\mathbf{C} = \{\mathbf{C}_1, \dots, \mathbf{C}_n\}$ the potential to decompose can be expressed as

$$\phi(\mathbf{X}) = f_1(\mathbf{C}_1) \dots f_n(\mathbf{C}_n) S_n, \quad (6.4)$$

where $f_i(\mathbf{C}_i) = \phi^{\perp \mathbf{C}_i}(\mathbf{X})$, $i = 1, \dots, n$, are the resulting factors and S_n is a normalisation constant that guarantees that ϕ and its factored counterpart sum up to the same value:

$$S_n = \frac{\sum_{\mathbf{x} \in \Omega_{\mathbf{X}}} \phi(\mathbf{x})}{\sum_{\mathbf{x} \in \Omega_{\mathbf{X}}} \prod_{i=1}^n f_i(\mathbf{x}^{\perp \mathbf{C}_i})}. \quad (6.5)$$

6.1 Learning RPTs from probabilistic potentials

This decomposition is computed using **multiplicativeFactorisation** (Alg. 19). The result will be a *List* node LN with a child (factor) for every component C_i in G_ϕ .

Algorithm 19 iterates on the set of components C (lines 6 to 23). Every iteration yields a child of LN . Two situations may arise when dealing with a component C_i . The first one is focused on components with one or two variables (lines 8 to 14). In this case a *Potential* node will represent the corresponding factor $f(C_i)$. The second will work with components with more than two variables (lines 15 to 22) and generates new recursive calls to **potentialFactorisation** for the analysis of $\phi^{\downarrow C_i}$ (line 19). The final part of Algorithm 19 computes the normalisation constant (line 25) as shown in Eq. 6.5. This constant will be represented with a *Value* node included as the last child of LN .

Example 49 Consider the network represented in Fig. 6.2 as the dependencies graph computed in the first part of algorithm **potentialFactorisation** (line 3 of Alg. 18). The graph is disconnected, so algorithm **multiplicativeFactorisation** (Alg. 19) is called (line 10 of Alg. 18).

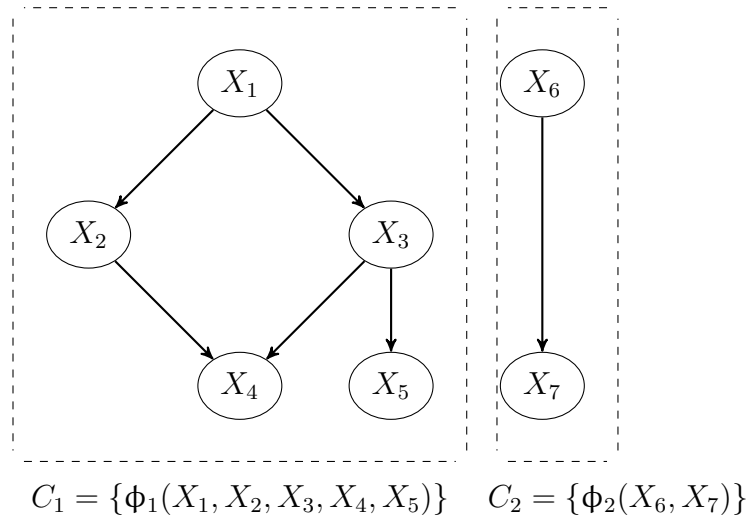


Figure 6.2: Dependencies graph computed during the factorisation process.

Alg. 19 analyses each connected component separately, storing the independent results as children of a *List* node. Finally, a normalisation constant is stored to

6.1 Learning RPTs from probabilistic potentials

ensure the correctness of the representation (lines 24 to 28 of Alg. 19). In the example proposed in Fig. 6.2, the network is composed of two connected components: C_1 and C_2 . The first component, C_1 , is analysed by recursively calling to Alg. 18, as C_1 's number of variables is higher than 2 (lines 15 to 22 of Alg. 19). The second component, C_2 , as it only contains two variables, is directly represented as a Potential node (lines 8 to 14 of Alg. 19). The built RPT at this stage of the process is shown in Fig. 6.3.

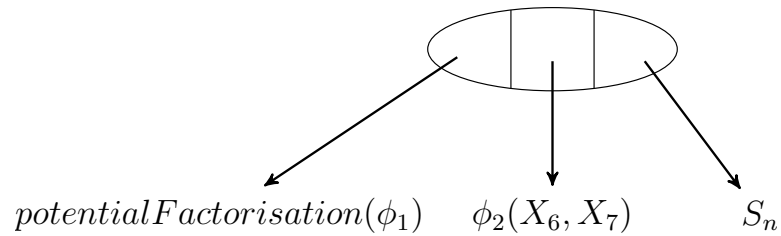


Figure 6.3: RPT representing the situation described in Fig. 6.2.

6.1 Learning RPTs from probabilistic potentials

```

1 multiplicativeFactorisation( $\phi$ ,  $\mathbf{C}$ )
   Input:
       Potential  $\phi$ 
       List  $\mathbf{C}$  of connected components in  $G_\phi$ 
   Output:  $LN$ , a List node
2 begin
3     // Makes  $LN$ , a new and empty List node
4      $LN \leftarrow$  new List Node
5     // Iterates on the list of components  $\mathbf{C}$ 
6     for each  $\mathbf{C}_i$  in  $\mathbf{C}$  do
7         // Considers the number of variables in the component
8         if  $\mathbf{C}_i$  contains 1 or 2 variables then
9             // Makes  $PN$  a new Potential node
10             $PN \leftarrow$  new Potential node;
11            //  $PN$  content: potential for variables in  $\mathbf{C}_i$ 
12             $PN \leftarrow \phi^{\downarrow \mathbf{X}_{\mathbf{C}_i}}$ ;
13            // Adds new node  $PN$  as factor in the List node
14            Add  $PN$  as  $LN$  child
15        else
16            // Factor with more than 2 variables: more analysis
17            // required. A new RPT  $\mathcal{T}_{\mathbf{C}_i}$  will be produced after
18            // analyzing  $\mathbf{C}_i$ . New call to potentialFactorisation
19             $\mathcal{T}_{\mathbf{C}_i} \leftarrow$  potentialFactorisation( $\phi^{\downarrow \mathbf{C}_i}$ )
20            // Adds  $\mathcal{T}_{\mathbf{C}_i}$  as new factor in the List node
21            Add  $\mathcal{T}_{\mathbf{C}_i}$  as  $LN$  child
22        end
23    end
24    // Computes the normalisation constant
25     $S_n \leftarrow$  computed normalisation constant (Eq. 6.5)
26    // Adds a new Value node  $VN$  for the constant
27     $VN \leftarrow$  new Value node for  $S_n$ ;
28    Add  $VN$  to  $LN$ ;
29    return  $LN$ 
30 end

```

Algorithm 19: Algorithm to compute multiplicative factorisations.

6.1.2.3 Detecting context-specific independencies

When G_ϕ is connected, ϕ cannot be decomposed as a product of factors, but still it may be possible to obtain such decompositions under some context, due to context-specific independencies. This condition will be analysed checking the *degree of dependence* between every variable X_i and those other variables belonging to its neighbourhood, $ne(X_i)$. The above mentioned degree of dependence is computed as

$$V_{X_i} = \sum_{X_j \in ne(X_i)} I(X_i, X_j). \quad (6.6)$$

This is the main goal of Algorithm **contextSpecificFactorisation** (see Algorithm 20), which computes and returns \mathcal{RT} , the RPT representing the potential received as argument.

The main block of Algorithm 20 consists of a loop iterating on the variables in $dom(\phi)$ until completing the decomposition (lines 6 to 30). The variables will be selected according to their degrees of dependence. The loop starts off testing if the potential under consideration is defined over one or two variables, which is in fact the stop condition. In such case, the potential will be decomposed and added as a *Potential* node to \mathcal{RT} (lines 8 - 12). The rest of the loop (lines 13 to 29) is devoted to removing variables step by step and checking the corresponding changes in G_ϕ .

Every iteration of the loop selects the variable maximizing the degree of dependence in order to look for the context in which the potential might be factorised. The chosen variable corresponds to:

$$X_{max} = \arg \max_{X_i} V_{X_i}. \quad (6.7)$$

Once X_{max} is selected it will be included in \mathbf{S}_1 or \mathbf{S}_2 . These are auxiliary vectors of variables that represent two types of variables: those that are closely related to all the others (\mathbf{S}_1) and those variables that are only highly dependent of only a subset of the variables in the component (\mathbf{S}_2) (line 3 of Alg. **contextSpecificFactorisation**). Variables in \mathbf{S}_1 will be translated into Split nodes in the RPT representation, whilst the variables in \mathbf{S}_2 will be used along with

6.1 Learning RPTs from probabilistic potentials

the variables in the resultant connected components for further analysis. Note that the ordering in which the variables are selected is very important, as every deletion is dependent on the previous one. The splitting of the tree with the variables of \mathbf{S}_1 will be performed using a first-in-first-out fashion.

Therefore, X_{max} will be included in \mathbf{S}_1 if it is completely connected to the rest of variables in G_ϕ . Otherwise it will be included in \mathbf{S}_2 . In both cases, X_{max} and its links will be removed from G_ϕ producing a new graph $G_\phi^{X_{max}}$ that will be considered in further iterations.

The remaining links are re-weighted with the value of the mutual information conditional on X_{max} , computed as

$$I(X_i, X_j | X_{max}) = \sum_{\substack{x_i \in \Omega_{X_i} \\ x_j \in \Omega_{X_j} \\ x_{max} \in \Omega_{X_{max}}} \phi(x_i, x_j, x_{max}) \log \frac{\phi(x_i, x_j | x_{max})}{\phi(x_i | x_{max}) \phi(x_j | x_{max})} \quad (6.8)$$

where

$$\phi(x_i, x_j, x_{max}) = \frac{\phi^{\downarrow X_i, X_j, X_{max}}(x_i, x_j, x_{max})}{sum_\phi}, \quad (6.9)$$

$$\phi(x_i, x_j | x_{max}) = \frac{\phi^{\downarrow X_i, X_j, X_{max}}(x_i, x_j, x_{max})}{\phi^{\downarrow X_{max}}(x_{max})}, \quad (6.10)$$

$$\phi(x_i | x_{max}) = \frac{\phi^{\downarrow X_i, X_{max}}(x_i, x_{max})}{\phi^{\downarrow X_{max}}(x_{max})}, \quad \phi(x_j | x_{max}) = \frac{\phi^{\downarrow X_j, X_{max}}(x_j, x_{max})}{\phi^{\downarrow X_{max}}(x_{max})}. \quad (6.11)$$

6.1 Learning RPTs from probabilistic potentials

```

1 contextSpecificFactorisation( $\phi, G_\phi$ )
  Input:
    Potential  $\phi$ 
    Dependencies graph  $G_\phi$ 
  Output:  $\mathcal{RT}$ , an RPT for  $\phi$ 
2 begin
3    $\mathbf{S}_1, \mathbf{S}_2 \leftarrow$  empty vectors of variables;
4    $G_\phi^{X_{max}} \leftarrow G_\phi$  // Initially the graph to analyse is  $G_\phi$ 
5   stopCondition  $\leftarrow$  false // Initially stopCondition is false
6   repeat
7     if  $|dom(\phi)| < 3$  // Number of variables in  $\phi < 3$ 
8     then
9        $PN \leftarrow$  new Potential node for  $\phi$  //  $PN$ : new Potential node
10       $\mathcal{RT} \leftarrow PN$  //  $\mathcal{I}_P$ : output of the algorithm
11      stopCondition  $\leftarrow$  true
12    end
13    else
14       $X_{max} \leftarrow$  selected var. for removing // Selects  $X_{max}$  with
15      Eq. 6.23
16      classifies  $X_{max}$  into  $\mathbf{S}_1$  or  $\mathbf{S}_2$ ;
17       $G_\phi^{X_{max}}$  graph after deleting  $X_{max}$  and its links
18       $I_\phi(X_{max}) \leftarrow$  value of information gain
19      if  $G_\phi^{X_{max}}$  is connected and  $|dom(\phi)| > 2$  then
20        if  $I_\phi(X_{max})$  exceeds the threshold (Eq. 6.19) then
21           $\mathcal{RT} \leftarrow$  independentFactorisation( $\phi, \mathbf{S}_1$ );
22          stopCondition  $\leftarrow$  true;
23        end
24      end
25      else
26        stopCondition  $\leftarrow$  true;
27        if  $\mathbf{S}_1 = \emptyset$  then  $\mathcal{RT} \leftarrow$  multiplicativeFactorisation( $\phi, \mathbf{C}$ );
28        else  $\mathcal{RT} \leftarrow$  splitChainFactorisation( $\phi, \mathbf{C}, \mathbf{S}_1, \mathbf{S}_2$ );
29      end
30    end
31  until stopCondition is true;
32  return  $\mathcal{RT}$ 
33 end

```

Algorithm 20: Algorithm to search for context-specific independencies.

6.1 Learning RPTs from probabilistic potentials

According to the structure of $G_\phi^{X_{max}}$, the algorithm proceeds as follows:

- $G_\phi^{X_{max}}$ remains connected after removing X_{max} . Then the decomposition is guided by the content of \mathbf{S}_1 through a call to **independentFactorisation**, (line 20 of Algorithm 20). This call relies on the information gain produced by splitting on X_{max} . The intuitive idea is to decide when the information gain generated by splitting by the current context (determined by the variables in \mathbf{S}_1) is high enough to start analyzing the restricted potentials independently. This process is controlled by a parameter δ , described in detail in Section 6.1.2.6, that sets the amount of information gain that we consider enough to split the tree by a given variable.

Example 50 *Consider the scenario proposed in Fig. 6.4, where the graph in the left represents the whole cluster that is being analysed. Imagine that the first removed variable was X_4 that, as it was connected to all the others, is stored in S_1 . As the cluster remains connected, we check if X_4 surpasses the threshold of information gain. Imagine that this threshold is not surpassed, so we keep removing variables from the cluster. The next variable that maximized the degree of dependence was X_6 , that is only connected to X_1 and X_5 , so it is stored in S_2 . Again, as the graph is still connected, we look for another variable to be removed, that in this case is X_5 . This variable is connected to all the remaining variables in the cluster, so we introduce it in S_1 . Again, we check the information gain generated by splitting by X_4 and X_5 , and we discover that it is higher than the threshold, so we proceed to apply Alg. 21. This algorithm basically splits by all the current variables in S_1 , and applies Alg. 18 to the original potential restricted to the correspondent context, for every possible combination of states of the variables in S_1 .*

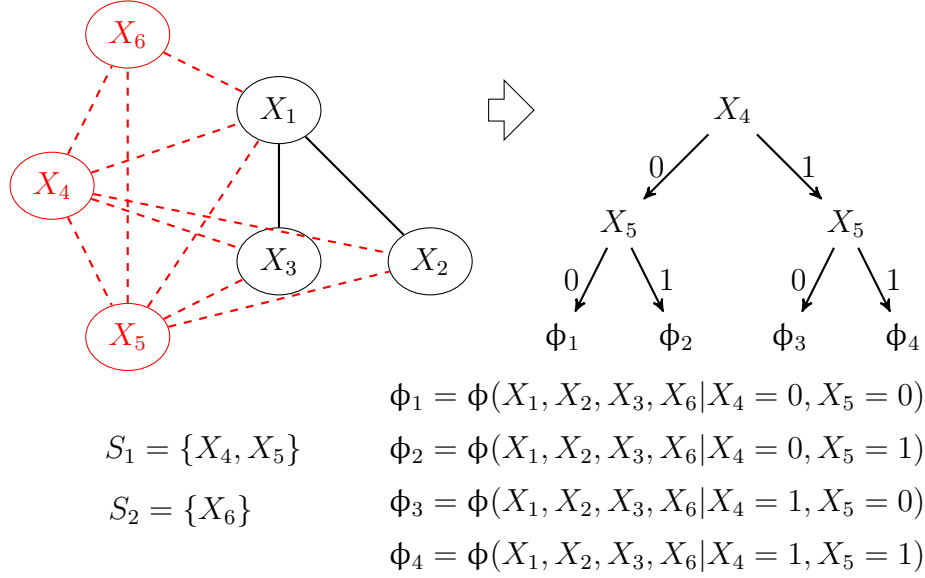


Figure 6.4: Step of the algorithm when the removal of X_5 exceeds the threshold of information gain.

- $G_\phi^{X_{max}}$ is disconnected and S_1 is empty. Then ϕ is decomposed as a multiplicative factorisation, (see the call to **multiplicativeFactorisation** in line 26 of Algorithm 20).

Example 51 *This scenario is represented in Fig. 6.5, where we have disconnected the graph into two clusters after removing variable X_5 . This variable goes into S_2 because it was only directly connected to X_3 and X_2 , but not to X_1 nor X_4 .*

The way of representing this scenario is by creating a List node with a children for each resultant connected component, plus an extra child for the normalisation constant. The potential related to each cluster will include the variables in the cluster plus the variables in S_2 , in this case only X_5 . Algorithm 18 is afterwards applied to each child independently.

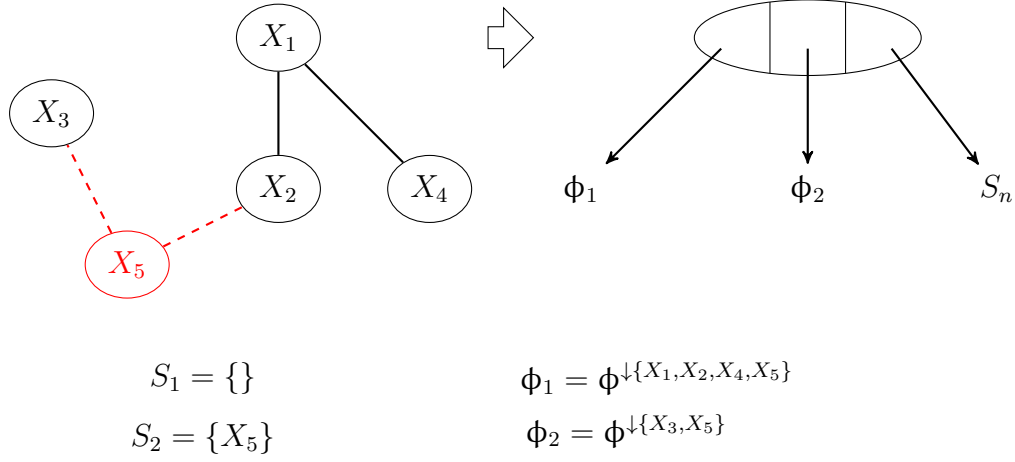
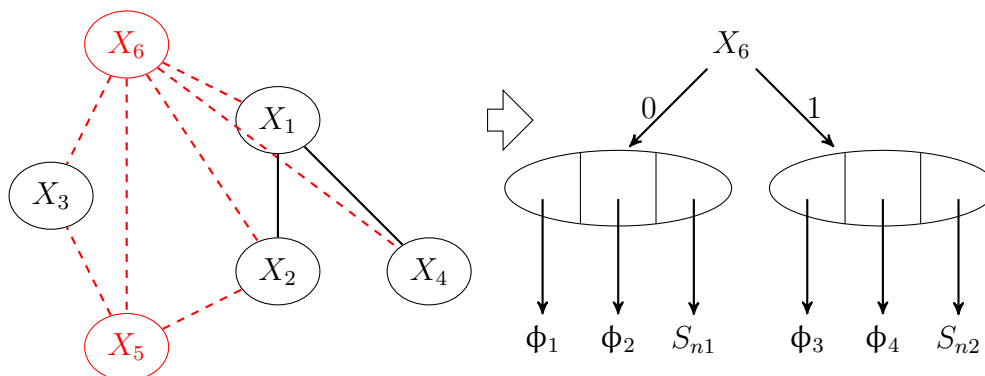


Figure 6.5: Step of the algorithm when we find a division in the graph and S_1 is empty.

- $G_\phi^{X_{max}}$ is disconnected and S_1 is not empty. Then a chain of *Split* nodes will be considered (with a call to **splitChainFactorisation**, Algorithm 20, line 27).

Example 52 Consider the graph in Fig. 6.6. The first variable removed from the graph was X_6 that, as it was connected to all the others, was introduced into S_1 . As the removal of X_6 did not disconnect the graph neither generated enough information gain, then we chose another variable to remove, that in this case was X_5 . As X_5 was not connected to all the variables, we stored it into S_2 . The removal of X_5 produces the disconnection of the graph into two clusters, and so we proceed to represent this situation using an RPT.

Algorithm 22 takes the original potential, the obtained set of clusters and both sets S_1 and S_2 as arguments. The representation will consist of a *Split* chain of the variables in S_1 , that in this case consists only of X_6 , and a factorisation for each context defined by a *List* node with as many children as connected components are plus the normalisation factors. Each subfactor with enough size will be analysed afterwards.



$$S_1 = \{X_6\}$$

$$S_2 = \{X_5\}$$

$$\phi_1 = (\phi^{R(X_6=0)})_{\downarrow(X_3, X_5)}$$

$$\phi_2 = (\phi^{R(X_6=0)})_{\downarrow(X_1, X_2, X_4, X_5)}$$

$$\phi_3 = (\phi^{R(X_6=1)})_{\downarrow(X_3, X_5)}$$

$$\phi_4 = (\phi^{R(X_6=1)})_{\downarrow(X_1, X_2, X_4, X_5)}$$

Figure 6.6: Step of the algorithm when we find a division in the graph and we have variables in S_1 .

6.1.2.4 independentFactorisation algorithm

This auxiliary algorithm is called within Algorithm 20 when the information gain due to splitting on X_{max} exceeds the threshold described in the previous section. This algorithm receives as arguments the potential to decompose, ϕ , and the set S_1 , that contains the variables connected to the rest of variables in $G_\phi^{X_{max}}$ when removed. The structure of the procedure is described in Algorithm 21.

```

1 independentFactorisation( $\phi, \mathbf{S}_1$ ); Input:
    Potential  $\phi$ 
    Vector of variables  $\mathbf{S}_1$ 
Output:  $\mathcal{RT}$ , an RPT with a Split node as root
2 begin
3   // Makes a chain of Split nodes for  $\mathbf{S}_1$  variables
4    $\mathcal{RT} \leftarrow$  root node in the split chain
5   // Iterates on  $\mathbf{S}_1$  configurations
6   for each possible value  $s_1$  of  $\mathbf{S}_1$  do
7     // Tries a decomposition for the potential restricted to
8     //  $s_1$  configuration:  $\phi^{R(\mathbf{S}_1=s_1)}$ 
9      $\mathcal{RT}_{s_1} \leftarrow$  potentialFactorisation( $\phi^{R(\mathbf{S}_1=s_1)}$ );
10    // Adds the resulting RPT to  $\mathcal{RT}$ 
11    Add  $\mathcal{RT}_{s_1}$  to leaf node for  $s_1$  configuration in  $\mathcal{RT}$ ;
12  end
13  return  $\mathcal{RT}$ 
14 end
    
```

Algorithm 21: Algorithm to independently factorise the branches of a Split chain.

independentFactorisation creates a *Split* chain (a set of *Split* nodes, one per variable in \mathbf{S}_1) This Split chain will follow the order in which the variables were introduced into \mathbf{S}_1 , that is to say, the first variable introduced will correspond to the root of the chain, the next variable will split the tree next, and so on. A loop (lines 6 to 12) iterates on the configurations in $\Omega_{\mathbf{S}_1}$, making new calls to **potentialFactorisation**. The potential passed as argument is $\phi^{R(\mathbf{S}_1=s_1)}$ (line 9) (ϕ restricted to the current configuration). The output of these calls is included as children in \mathcal{RT} (line 11).

6.1.2.5 splitChainFactorisation algorithm

This algorithm is called from Algorithm 20 when either $G_\phi^{X_{max}}$ is disconnected in \mathbf{C} components or the remaining variables in the connected component is less or equal than 2, and \mathbf{S}_1 is not empty. The RPT to be computed by such algorithm

6.1 Learning RPTs from probabilistic potentials

should reflect the conditional dependence on the variables in \mathbf{S}_1 , including a chain of *Split* nodes (a complete tree defined over this set of variables). Each leaf of this tree corresponds to a complete configuration for the variables in \mathbf{S}_1 .

The **splitChainFactorisation** algorithm distinguishes two different scenarios:

- **C** contains a single component. In this case the factor assigned to the leaf node in the split chain for each configuration \mathbf{s}_1 is the output of a new call to **potentialFactorisation**, passing as argument the potential restricted to configuration \mathbf{s}_1 (lines 24 to 28).

Example 53 *This scenario is represented in Fig. 6.7, where after removing the variables X_2, X_3 and X_4 the set of remaining variables in the connected component only contains two variables. The algorithm builds a Split chain with the variables stored in S_1 , and for each branch, we apply Alg. 18 to the correspondent potential.*

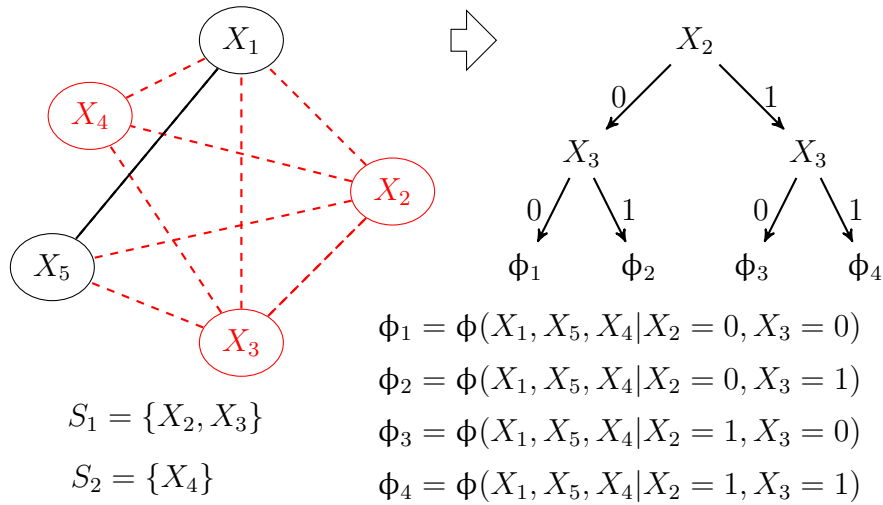


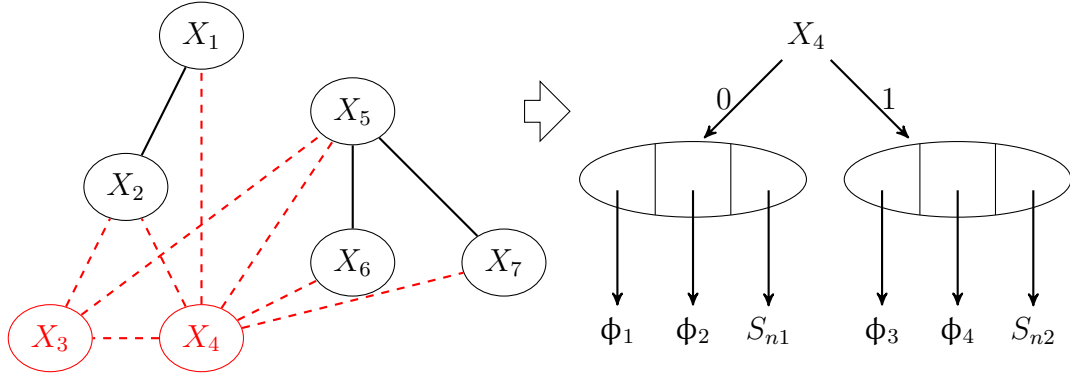
Figure 6.7: Creation of a Split chain.

- **C** contains several components (lines 7 to 23). Then the algorithm computes a multiplicative factorisation producing a *List* node LN whose factors are the result of decomposing each component. Components will be considered one by one (lines 11-17). Focusing on the iteration for the i -th component,

6.1 Learning RPTs from probabilistic potentials

a new call to **potentialFactorisation** is made, but using as argument ϕ restricted to the configuration \mathbf{s}_1 and marginalised to keep the variables in $\mathbf{C}_i \cup \mathbf{S}_2$ (line 15). The result is stored as the i -th child in LN (line 16). The last child in this node is the normalisation constant. Once computed the multiplicative factorisation, LN is stored in the leaf node of the split chain corresponding to \mathbf{s}_1 (line 22).

Example 54 Consider the situation represented in Fig. 6.8. After removing X_4 and X_3 we find a decomposition of the graph in two connected components, so we proceed to build an RPT with a Split chain of the variables in S_1 , in this example just X_4 , and in the leaves we build the factorisation, that has a List node as a root, and one child per resultant connected component plus a normalisation factor. Again, we apply Alg. 18 to every correspondent subfactor.



$$\begin{aligned}
 S_1 &= \{X_4\} & \phi_1 &= (\phi^{R(X_4=0)}) \downarrow (X_1, X_2, X_3, X_4) \\
 S_2 &= \{X_3\} & \phi_2 &= (\phi^{R(X_4=0)}) \downarrow (X_5, X_6, X_7, X_3, X_4) \\
 & & \phi_3 &= (\phi^{R(X_4=1)}) \downarrow (X_1, X_2, X_3, X_4) \\
 & & \phi_4 &= (\phi^{R(X_4=1)}) \downarrow (X_5, X_6, X_7, X_3, X_4)
 \end{aligned}$$

Figure 6.8: Creation of a Split chain with a decomposition of the auxiliary graph.

The normalisation constant is computed as follows. Assume that leaf h is reached by configuration \mathbf{s}_h . The potential assigned to the leaf is $\phi^h = \phi^{R(\mathbf{S}_1=\mathbf{s}_h)}$

6.1 Learning RPTs from probabilistic potentials

(denoting the potential restricted to configuration \mathbf{s}_h). The complete decomposition of ϕ^h (represented in the RPT with a *List* node) is given by the components $\mathbf{C} = \{\mathbf{C}_1, \dots, \mathbf{C}_n\}$:

$$\phi^h(\mathbf{c}_1 \dots \mathbf{c}_n, \mathbf{s}_2) = S^h \prod_{i=1}^n (\phi^h)^{\downarrow \mathbf{C}_i \cup \mathbf{S}_2}(\mathbf{c}_i, \mathbf{s}_2). \quad (6.12)$$

S^h is computed according to the nature of the factorisation performed, in order to minimize the possible normalisation error produced when factoring. When the factors do not share variables (i.e. \mathbf{S}_2 is an empty set) then S^h is a normalisation constant that is computed as

$$S^h = \frac{\sum_{\substack{\mathbf{c} \in \Omega_{\mathbf{C}} \\ \mathbf{s}_2 \in \Omega_{\mathbf{S}_2}}} \phi^h(\mathbf{c}_1 \dots \mathbf{c}_n, \mathbf{s}_2)}{\sum_{\substack{\mathbf{c} \in \Omega_{\mathbf{C}} \\ \mathbf{s}_2 \in \Omega_{\mathbf{S}_2}}} \prod_{i=1}^n (\phi^h)^{\downarrow \mathbf{C}_i \cup \mathbf{S}_2}(\mathbf{c}_i, \mathbf{s}_2)}. \quad (6.13)$$

Proposition 2 *Otherwise, S^h is a potential that depends on the variables in \mathbf{S}_2 (common variables of all the factors), $S^h(\mathbf{s}_2)$ can be computed as:*

$$S^h(\mathbf{s}_2) = ((\phi^h)^{\downarrow \mathbf{S}_2}(\mathbf{s}_2))^{1-n}. \quad (6.14)$$

Proof 4 *As $S^h(\mathbf{s}_2)$ depends on the variables in \mathbf{S}_2 , equation (6.13) can be rewritten as:*

$$S^h(\mathbf{s}_2) = \frac{\sum_{\mathbf{c} \in \Omega_{\mathbf{C}}} \phi^h(\mathbf{c}_1 \dots \mathbf{c}_n, \mathbf{s}_2)}{\sum_{\mathbf{c} \in \Omega_{\mathbf{C}}} \prod_{i=1}^n (\phi^h)^{\downarrow \mathbf{C}_i \cup \mathbf{S}_2}(\mathbf{c}_i, \mathbf{s}_2)}.$$

In the denominator, the potentials in the product do not share any variable but those included in \mathbf{S}_2 , so the previous formula can be written as:

$$S^h(\mathbf{s}_2) = \frac{\sum_{\mathbf{c} \in \Omega_{\mathbf{C}}} \phi^h(\mathbf{c}_1 \dots \mathbf{c}_n, \mathbf{s}_2)}{\prod_{i=1}^n \sum_{\mathbf{c} \in \Omega_{\mathbf{C}}} (\phi^h)^{\downarrow \mathbf{C}_i \cup \mathbf{S}_2}(\mathbf{c}_i, \mathbf{s}_2)} = \frac{(\phi^h)^{\downarrow \mathbf{S}_2}(\mathbf{s}_2)}{\prod_{i=1}^n (\phi^h)^{\downarrow \mathbf{S}_2}(\mathbf{s}_2)}.$$

which completes the proof. □

6.1 Learning RPTs from probabilistic potentials

```

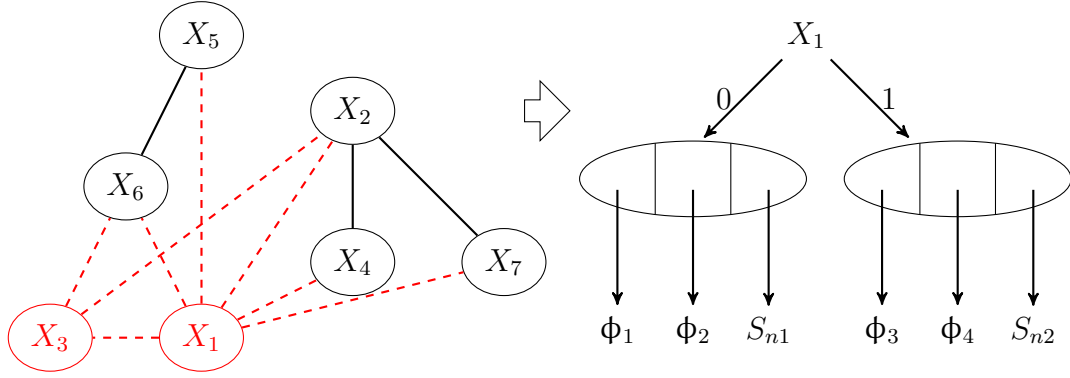
1 splitChainFactorisation( $\phi, \mathbf{C}, \mathbf{S}_1, \mathbf{S}_2$ )
   Input:
       Potential  $\phi$ 
       List  $\mathbf{C}$  of connected components in  $G_\phi$ 
       Vectors of variables  $\mathbf{S}_1$  and  $\mathbf{S}_2$ 
   Output:  $\mathcal{RT}$ , an RPT with a Split node as root
2 begin
3   // Makes a chain of Split nodes for  $\mathbf{S}_1$  variables
4    $\mathcal{RT} \leftarrow$  root node in the split chain ;
5   for each possible value  $s_1$  of  $\mathbf{S}_1$  do
6     // Checks the number of components in  $\mathbf{C}$ 
7     if  $\mathbf{C}$  has more than one component then
8       // There will be a factor per component
9       // The factors will be stored in a List node
10       $LN \leftarrow$  new List Node ;
11      for each element  $\mathbf{C}_i$  in  $\mathbf{C}$  do
12        // Tries a decomposition for the potential according
13        // to configuration  $s_1$ :  $\phi^{R(\mathbf{S}_1=s_1)}$ 
14        // but marginalised to variables in  $\mathbf{C}_i \cup \mathbf{S}_2$ 
15         $\mathcal{RT}_{s_1} \leftarrow$  potentialFactorisation( $(\phi^{R(\mathbf{S}_1=s_1)}) \downarrow_{\mathbf{C}_i \cup \mathbf{S}_2}$ );
16        Add  $\mathcal{RT}_{s_1}$  to  $LN$  ;
17      end
18      // Computes the normalisation constant
19       $S^h \leftarrow$  computed normalisation potential (Eq. (6.14));
20       $PN \leftarrow$  new Potential node for  $S^h$ ;
21      Add  $PN$  to  $LN$ ;
22      Add  $LN$  to leaf node for  $s_1$  configuration in  $\mathcal{RT}$  ;
23    end
24    else
25       $\mathcal{RT}_{s_1} \leftarrow$  potentialFactorisation ( $\phi^{R(\mathbf{S}_1=s_1)}$ )
26      // Adds the resulting RPT to  $\mathcal{RT}$ 
27      Add  $\mathcal{RT}_{s_1}$  to leaf node for  $s_1$  configuration in  $\mathcal{RT}$  ;
28    end
29  end
30  return  $\mathcal{RT}$ 
31 end

```

Algorithm 22: Algorithm to represent the RPT once a factorisation is found.

Example 55 We can find another example of a decomposition of the graph in Fig. 6.9. The components conforming the decomposition are $\mathbf{C}_1 = \{X_5, X_6\}$ and $\mathbf{C}_2 = \{X_2, X_4, X_7\}$. Assume that $\mathbf{S}_1 = \{X_1\}$ and $\mathbf{S}_2 = \{X_3\}$. Then the tree will contain a *Split* node for X_1 . Factors f_3 and f_6 represent normalising constants. New decompositions should be analysed for f_1 , f_2 , f_4 and f_5 with recursive calls to the algorithm.

Sometimes, to simplify the notation and when the marginal variables are explicitly given in the potential arguments, we will write $(\phi^h)^{\downarrow \mathbf{C}_i \cup \mathbf{S}_2}(\mathbf{c}_i, \mathbf{s}_2)$ as $\phi^h(\mathbf{c}_i, \mathbf{s}_2)$ and $(\phi^h)^{\downarrow \mathbf{S}_2}(\mathbf{s}_2)$ as $\phi^h(\mathbf{s}_2)$.



$$\mathbf{S}_1 = \{X_1\}$$

$$\mathbf{S}_2 = \{X_3\}$$

$$\phi_1 = (\phi^{R(X_4=0)})^{\downarrow (X_5, X_6, X_3)}$$

$$\phi_2 = (\phi^{R(X_4=0)})^{\downarrow (X_2, X_4, X_7, X_3)}$$

$$\phi_3 = (\phi^{R(X_4=1)})^{\downarrow (X_5, X_6, X_3)}$$

$$\phi_4 = (\phi^{R(X_4=1)})^{\downarrow (X_2, X_4, X_7, X_3)}$$

Figure 6.9: Example of G_ϕ complete division. New recursive calls required for factors ϕ_1 , ϕ_2 , ϕ_3 and ϕ_4 .

6.1.2.6 Setting the sensitivity of context-specific independencies detection

The introduction of a new *Split* node through a call to Alg. 21 in Algorithm 20 (lines 19 to 22) depends on the information gain produced by splitting ϕ on X_{max} (the variable selected for splitting). The information gain is computed

6.1 Learning RPTs from probabilistic potentials

as the difference between the Kullback-Leibler divergence between the potential obtained after splitting by X_{max} and the original potential. This difference can be computed as follows [76]:

$$I_\phi(X_{max}) = \text{sum}_\phi(\log|\Omega_{X_{max}}| - \log \text{sum}_\phi) + \sum_{x_i \in \Omega_{X_{max}}} \text{sum}_\phi^{R(X_{max}=x_i)} \log \text{sum}_\phi^{R(X_{max}=x_i)} \quad (6.15)$$

Proposition 3 *The maximum value for $I_\phi(X_{max})$ can be obtained using the properties of Shannon's entropy, as:*

$$I_\phi(X_{max}) \leq \text{sum}_\phi(\log|\Omega_{X_{max}}| - \log \text{sum}_\phi) + N_{X_{max}} \log N_{X_{max}}. \quad (6.16)$$

Proof 5 *Define $N_{X_{max}} = \sum_{x_i \in \Omega_{X_{max}}} \text{sum}_\phi^{R(X_{max}=x_i)}$ (i.e. the sum of the values of the potential corresponding to the configurations of X_{max}). Then, it holds that*

$$-\frac{1}{N_{X_{max}}} \sum_{x_i \in \Omega_{X_{max}}} \text{sum}_\phi^{R(X_{max}=x_i)} \log \frac{\text{sum}_\phi^{R(X_{max}=x_i)}}{N_{X_{max}}} \geq 0, \quad (6.17)$$

and therefore

$$\begin{aligned} & -\frac{1}{N_{X_{max}}} \sum_{x_i \in \Omega_{X_{max}}} \text{sum}_\phi^{R(X_{max}=x_i)} (\log \text{sum}_\phi^{R(X_{max}=x_i)} - \log N_{X_{max}}) \geq 0 \Rightarrow \\ & -\frac{1}{N_{X_{max}}} \left\{ \sum_{x_i \in \Omega_{X_{max}}} \text{sum}_\phi^{R(X_{max}=x_i)} \log \text{sum}_\phi^{R(X_{max}=x_i)} - \sum_{x_i \in \Omega_{X_{max}}} \text{sum}_\phi^{R(X_{max}=x_i)} \log N_{X_{max}} \right\} \geq 0 \Rightarrow \\ & \sum_{x_i \in \Omega_{X_{max}}} \text{sum}_\phi^{R(X_{max}=x_i)} \log \text{sum}_\phi^{R(X_{max}=x_i)} \leq \log N_{X_{max}} \sum_{x_i \in \Omega_{X_{max}}} \text{sum}_\phi^{R(X_{max}=x_i)} = N_{X_{max}} \log N_{X_{max}}. \end{aligned} \quad (6.18)$$

Hence, replacing in (6.15) the upper bound obtained in (6.18), we obtain (6.16), which completes the proof. \square

6.1 Learning RPTs from probabilistic potentials

Using this result, the threshold for detecting context-specific independencies will be controlled by the parameter $0 \leq \delta \leq 1$, so that the variable X_{max} will be used for introducing a new *Split* node if

$$I_\phi(X_{max}) \geq \delta (\text{sum}_\phi(\log |X_{max}| - \log \text{sum}_\phi) + N_{X_{max}} \log N_{X_{max}}). \quad (6.19)$$

Thus, the value of δ controls the behaviour of the algorithm regulating the degree of context-specific independencies detection. Note that values of δ close to 1 indicate that only when the information gain is close to its upper bound, the split will be carried out.

6.1.3 Examples

In this section we illustrate the algorithm with two examples of learning from potentials with different features. Although the examples are not exhaustive they offer some insights about its application and results.

Example 56 Consider a potential ϕ defined over $\mathbf{X} = \{X_1, X_2, X_3, X_4\}$ represented as a probability tree (Fig. 6.10). ϕ can be obtained combining the two potentials included in Fig. 6.11, that is: $\phi(X_1, X_2, X_3, X_4) = \phi_1(X_1, X_2) \cdot \phi_2(X_3, X_4)$.

6.1 Learning RPTs from probabilistic potentials

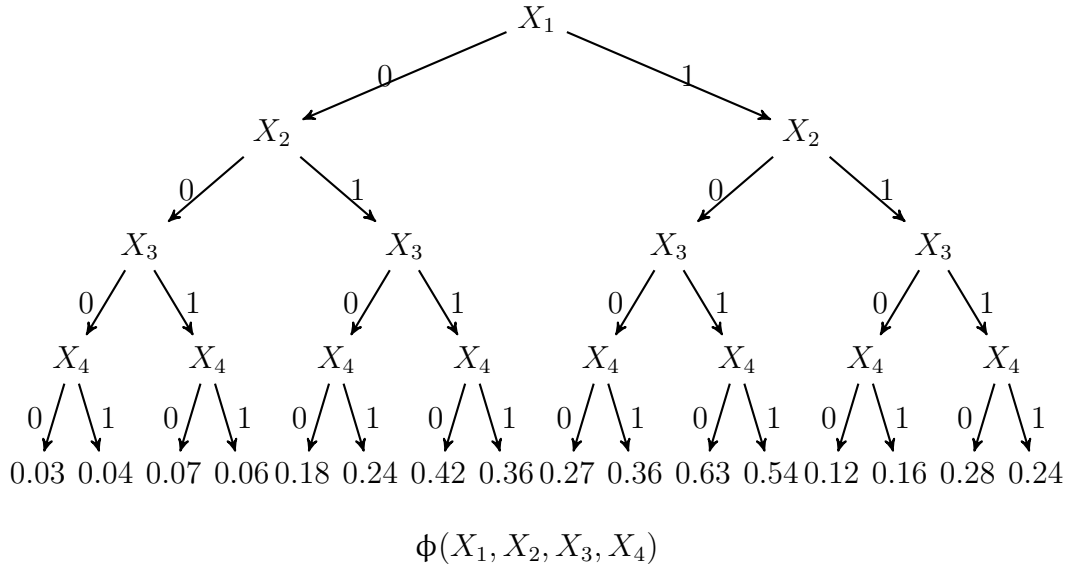


Figure 6.10: Potential ϕ to be decomposed in Example 1.

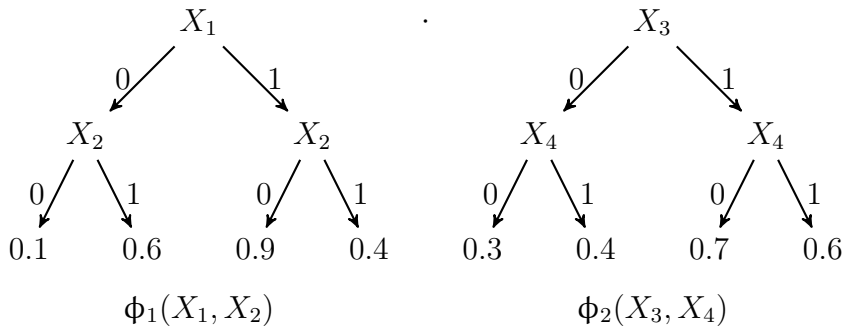


Figure 6.11: Factors generating ϕ in Example 1.

When ϕ is passed as an argument to Algorithm **potentialFactorisation**, the first step is to construct G_ϕ . This involves the computation of the mutual information between every pair of variables. In this example the value selected for the threshold is $\varepsilon = 1E-6$. The selection of this value is justified in order to dismiss small positive values of mutual information for independent variables due to round-off errors. The values of mutual information for each pair of variables are:

6.1 Learning RPTs from probabilistic potentials

- $I(X_1, X_2) = 0.148 > \varepsilon$,
- $I(X_1, X_3) = 1.28E - 16 < \varepsilon$,
- $I(X_1, X_4) = 2.20E - 16 < \varepsilon$,
- $I(X_2, X_3) = 2.22E - 16 < \varepsilon$,
- $I(X_2, X_4) = 2.22E - 16 < \varepsilon$,
- $I(X_3, X_4) = 0.005 > \varepsilon$.

Therefore G_ϕ contains only two links, $X_1 - X_2$ and $X_3 - X_4$, and is partitioned into two components: $\mathbf{C}_1 = \{X_1, X_2\}$ and $\mathbf{C}_2 = \{X_3, X_4\}$. A call to **multiplicativeFactorisation** will decompose ϕ according to the components (see line 11, Algorithm 18). This algorithm iterates over each component (lines 6 to 23 in Algorithm 19). As both factors are defined over 2 variables, a direct decomposition is computed through marginalisation. That is, the factors obtained are $\phi_1(X_1, X_2)$ and $\phi_2(X_3, X_4)$ (see Fig. 6.12).

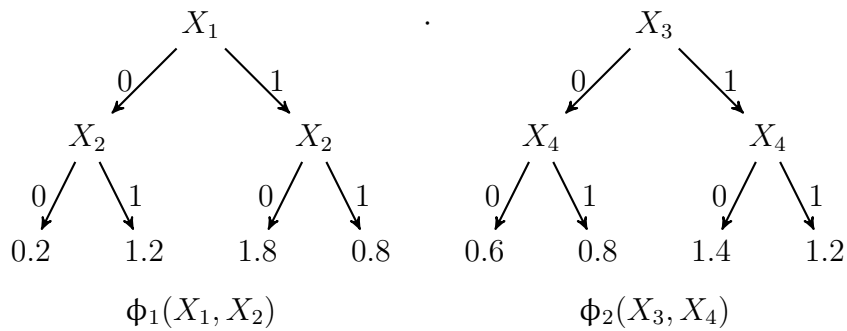


Figure 6.12: Factors obtained from the decomposition in Example 1.

The third factor is the normalisation constant computed as stated in Eq. 6.5. The learned RPT is the one displayed in Fig. 6.13.

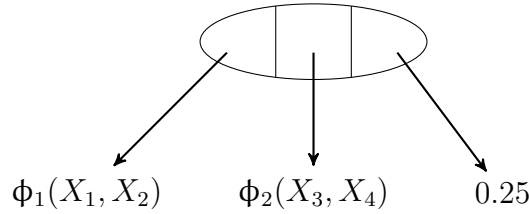


Figure 6.13: RPT learned in Example 1.

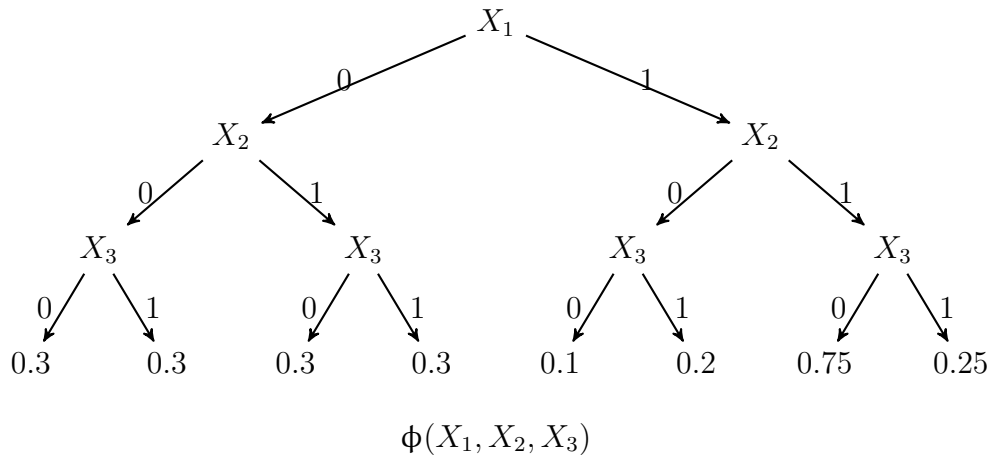


Figure 6.14: Potential ϕ considered in Example 2.

Example 57 Consider a potential ϕ defined over $\mathbf{X} = \{X_1, X_2, X_3\}$ as shown in Fig. 6.14, and a threshold value $\varepsilon = 0.001$. Again, the first operation carried out by Algorithm **potentialFactorisation** is the computation of the mutual information values in order to build G_ϕ . These values are:

- $I(X_1, X_2) = 0.039 > \varepsilon$,
- $I(X_1, X_3) = 0.012 > \varepsilon$,
- $I(X_2, X_3) = 0.021 > \varepsilon$.

6.1 Learning RPTs from probabilistic potentials

Hence, G_ϕ is a complete graph with a single component and Algorithm **contextSpecificFactorisation** is invoked from **potentialFactorisation** (line 14 in Algorithm 18). Then **contextSpecificFactorisation** receives both ϕ and G_ϕ as argument and the block in line 13 is executed as $|\Omega_{\mathbf{X}}| = 3$. The next step selects a candidate variable for deletion, computing the connectivity values as defined in Eq. (6.22). These values are: $V(X_1) = 0.051$, $V(X_2) = 0.06$ and $V(X_3) = 0.033$.

Therefore, X_2 is the selected variable (X_{max}) and is inserted in \mathbf{S}_1 (because it was connected to X_2 and X_3 in G_ϕ). Now $G_\phi^{X_2}$ is obtained by computing the new weight for the remaining link: $0.025 > \varepsilon$ for $X_1 - X_3$. The information gain for X_2 is 0.099. As the maximum information gain is 1.73. Assuming a threshold $\delta = 0.05$, then Eq. (6.19) holds and Algorithm **independentFactorisation** is invoked.

As $\mathbf{S}_1 \neq \emptyset$, a Split chain is built with the variables in \mathbf{S}_1 (in this case only X_2). The loop in **independentFactorisation**, lines 6-12, considers each configuration in $\Omega_{\mathbf{S}_1}$, producing new calls to **potentialFactorisation** with arguments $\phi^{R(X_2=x_{21})}$ and $\phi^{R(X_2=x_{22})}$ respectively. These calls finally produce new invocations of **contextSpecificFactorisation** (because $G_\phi^{X_2}$ contains a single component). These calls produce new Potential nodes for $\phi^{R(X_2=x_{21})}$ and $\phi^{R(X_2=x_{22})}$. The learned RPT is shown in Fig. 6.15, where the potential nodes are represented as PTs whose branches containing repeated values have been pruned.

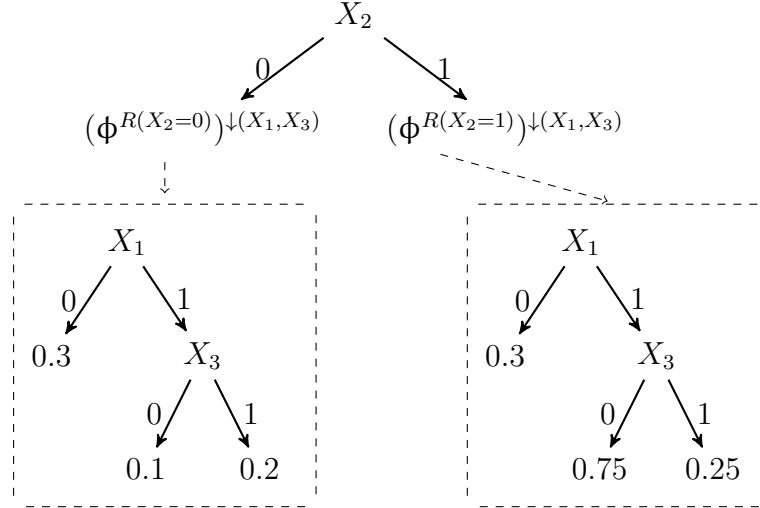


Figure 6.15: RPT learned in Example 2.

6.1.4 Problem Complexity and Properties of the Algorithm

6.1.4.1 Problem Complexity

Obtaining minimum size representations of a probabilistic potential by means of an RPT is not a trivial problem, as we establish in the next proposition. By the size of an RPT \mathcal{RT} , we mean the number of Value nodes plus the sizes of the potential nodes (bounded above by the product of the cardinalities of the domains Ω_{X_i} of all the variables X_i in \mathcal{RT}).

Proposition 4 *Let ϕ be a probabilistic potential represented by an RPT. Then, the problem of finding an RPT of minimum size representing ϕ is NP-hard.*

Proof 6 *By reduction from 3-SAT. Let \mathcal{J} be an instance of 3-SAT with a set of variables $U = \{u_1, \dots, u_m\}$ and a set of clauses $C = \{C_1, \dots, C_k\}$. Let \mathcal{RT} be an RPT with a List root node. Let U denote the variables of \mathcal{RT} and assume the set of possible values for each variable is $\{0, 1\}$.*

The List node will have one element T_i for each one of the clauses C_i , where T_i is a potential defined for the variables $\{u_{i_1}, u_{i_2}, u_{i_3}\}$ in clause C_i . The value of

6.1 Learning RPTs from probabilistic potentials

the potential for a combination of values $(u_{i_1} = r_1, u_{i_2} = r_2, u_{i_3} = r_3)$ will be 1 if the clause is true for this combination of true values of the variables (identifying 0 with false and 1 with true) and 0 otherwise.

It is clear that the RPT \mathcal{RT} represents a potential which has a value of 1 for a configuration of values $(u_1 = r_1, \dots, u_m = r_m)$ if all the clauses are true. The 3-SAT problem is equivalent to the fact that the potential represented by the RPT \mathcal{RT} is identically 0. If the set of non trivial clauses (i.e. those clauses non containing u_i and $\neg u_i$) is different from the empty set, then the potential will have some 0 values, and the fact that the set of clauses can be satisfied is equivalent to the fact that the minimum size of an RPT representing this potential is greater or equal to 2: if the clauses can be satisfied there is a 0 value and we need, at least, another leaf to represent the 1 corresponding to the satisfying configuration; if the clauses cannot be satisfied, then we can represent the potential by means of an RPT with only one node which is a Value node containing a 0. \square

6.1.4.2 Properties of the algorithm

Union property

If ϕ is a non-null potential defined for variables $dom(\phi)$, then we can always regard it as a probability distribution obtained by the normalisation, i.e. ϕ/sum_ϕ . This probability distribution determines a family of conditional independence relationships. In what follows, $I(\mathbf{X}_1, \mathbf{X}_2 | \mathbf{S} = \mathbf{s})$ means that the set of variables \mathbf{X}_1 is independent of the set of variables \mathbf{X}_2 given the configuration $(\mathbf{S} = \mathbf{s})$. We say that *the union property with respect to configurations holds* in ϕ if for the probability distribution obtained by normalising ϕ , it holds that for any configuration $(\mathbf{S} = \mathbf{s})$,

$$I(\mathbf{X}_1, \mathbf{X}_2 | \mathbf{S} = \mathbf{s}) \text{ and } I(\mathbf{X}_1, \mathbf{X}_3 | \mathbf{S} = \mathbf{s}) \Rightarrow I(\mathbf{X}_1, \mathbf{X}_2 \cup \mathbf{X}_3 | \mathbf{S} = \mathbf{s})$$

A consequence of this property is that if there is a dependence between two sets of variables \mathbf{X} and \mathbf{Y} given $\mathbf{S} = \mathbf{s}$, then it is always possible to find two variables $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$ such that X and Y are dependent given $\mathbf{S} = \mathbf{s}$. The importance of this consequence, from an algorithmic point of view, is remarkable, as it guarantees that in order to detect dependencies between sets of variables,

6.1 Learning RPTs from probabilistic potentials

it is enough to check dependencies between individual variables. Therefore, if we have n variables it is enough to check dependencies between $n(n-1)/2$ pairs of variables, instead of $\sum_{i=1}^{n-1} \binom{n}{i} (2^{n-i} - 2)$ pairs of non trivial disjoint subsets of n variables. In general, we can say that it is sensible to assume that any tractable algorithm to detect dependencies rely on this property to guarantee correctness. For example, when learning Bayesian networks from data, faithfulness is usually assumed, which is a much stronger assumption than union [22].

Even if $\epsilon = 0$, there is no guarantee that the RPT produced by Algorithm 18 will represent ϕ exactly. The reason is that it checks pairwise independence relationships between variables and it does not guarantee the independence between sets of variables. However, if the union property holds in the probability distribution associated with ϕ (the probability distribution given by the potential ϕ after normalisation) then it returns an exact representation.

Proposition 5 *Let ϕ be a potential for which the union property with respect to configurations holds. Then, algorithm potentialFactorisation (Alg. 18), with $\epsilon = 0$, returns an RPT which is an exact representation of ϕ .*

Proof 7 *The proof can be given with a recursive argument. On the leaves (lines 7-12 of Alg. 20, lines 8-14 of Alg. 19, line 11 of Alg. 21, and lines 24-28 of Alg. 22) the representation of the potential is exact.*

When Split nodes are included (Alg. 21 and Alg. 22), then we select a child for each configuration $\mathbf{S}_1 = \mathbf{s}_1$ of split variables. Each one of these children represents the potential $\phi^{R(\mathbf{S}_1=\mathbf{s}_1)}$, so that the node will represent the entire potential ϕ .

When a List node is created (lines 7-23 of Alg. 22 and Alg. 19), the exact decomposition is a result of the following facts:

1. *When $\epsilon = 0$, there is a link between two variables X_i and X_j in the graph associated with the potential ϕ only if the mutual information between X_i and X_j is > 0 (computed from the normalised potential), i.e. when the variables are not independent. When a variable X_{max} is removed from the graph and added to sets \mathbf{S}_1 and \mathbf{S}_2 , the mutual information is computed conditional on X_{max} , and thus, when a link between two variables disappears it means that the variables are conditionally independent given X_{max} . From*

6.1 Learning RPTs from probabilistic potentials

this point onwards, all the computations related to the graph are carried out by previously conditioning on X_{max} .

2. *As a consequence of this, when $\mathbf{X} \setminus \{X_{max_1}, \dots, X_{max_n}\}$ is decomposed into n components $\{\mathbf{C}_1, \dots, \mathbf{C}_n\}$, then any pair of variables $X \in \mathbf{C}_i, Y \in \mathbf{C}_j$ ($i \neq j$) are conditionally independent given $\{X_{max_1}, \dots, X_{max_n}\} = \mathbf{S}_1 \cup \mathbf{S}_2$.*
3. *As a consequence of the union property, it follows that all the components $\{\mathbf{C}_1, \dots, \mathbf{C}_n\}$ are conditionally independent among them given the variables in $\mathbf{S}_1 \cup \mathbf{S}_2$.*
4. *When factoring with $\mathbf{S}_1 \neq \emptyset$, it means that Split nodes have been previously considered for the variables in this set, and that the potential has been restricted to these values, so that the factor is conditional on \mathbf{S}_1 .*
5. *When the Split node is created, the variables in the components are conditionally independent given the variables in \mathbf{S}_2 (Alg. 21 is a special case where $\mathbf{S}_2 = \emptyset$).*
6. *If the variables in the components are conditionally independent given \mathbf{S}_2 , according to the normalised potential $f = \phi / \text{sum}_\phi$, then f can be decomposed as (basic result for conditional independence):*

$$f(\mathbf{c}_1, \dots, \mathbf{c}_n, \mathbf{s}_2) = \frac{\prod_{i=1}^n f(\mathbf{c}_i, \mathbf{s}_2)}{f(\mathbf{s}_2)^{n-1}},$$

where $f(\mathbf{c}_i, \mathbf{s}_2)$ is the marginalisation of f to variables $\mathbf{C}_i \cup \mathbf{S}_2$.

Therefore,

$$\phi(\mathbf{c}_1, \dots, \mathbf{c}_n, \mathbf{s}_2) / \text{sum}_\phi = \frac{\prod_{i=1}^n \phi(\mathbf{c}_i, \mathbf{s}_2) / \text{sum}_\phi}{(\phi(\mathbf{s}_2) / \text{sum}_\phi)^{n-1}}$$

and cancelling sum_ϕ ,

$$\phi(\mathbf{c}_1, \dots, \mathbf{c}_n, \mathbf{s}_2) = \frac{\prod_{i=1}^n \phi(\mathbf{c}_i, \mathbf{s}_2)}{\phi(\mathbf{s}_2)^{n-1}}.$$

This is precisely the factorisation applied in our procedure (the denominator is added as the normalisation potential).

□

Minimal RPT

An RPT \mathcal{RT} is said to be *minimal* if the following conditions are verified:

1. There are no potential leaf nodes with more than two variables, and if a potential in a leaf node has exactly two variables, it cannot be factored as a product of two potentials, each one of them defined for a single variable.
2. If N is a *Split* node and ϕ is the potential associated with this node, with $\text{dom}(\phi) = \mathbf{C}$, then it is not possible to decompose ϕ as product of two potentials ϕ_1 and ϕ_2 defined on \mathbf{C}_1 and \mathbf{C}_2 respectively, where \mathbf{C}_1 and \mathbf{C}_2 are non-empty sets that constitute a partition of \mathbf{C} .

The intuition behind this definition is what guides the algorithm presented in this chapter. The algorithm seeks for factorisations until reaching potentials defined for 2 variables, and also Split nodes are only considered when multiplicative decompositions are not possible, with the aim of obtaining factorisations based on conditional independence relationships.

The following result shows that our algorithm finds minimal representations except at most in what concerns the nodes representing normalisation factors, that still can remain as Potential leaf nodes not verifying condition 1 above.

Proposition 6 *Let ϕ be a potential for which the union property with respect to configurations holds. Then, algorithm `potentialFactorisation` (Alg. 18) with $\epsilon = 0$ obtains minimal exact decompositions, except possibly for the normalisation factors $\phi^h(\mathbf{s}_2)^{1-n}$ of Eq. (6.14).*

Proof 8 *We already proved in Prop. 5 that the decompositions obtained are exact. Minimality can be derived from the description of the algorithms as we show below.*

The first property of minimal decompositions is a consequence of the fact that the algorithms only add Potential leaf nodes in the following situations:

- *Lines 8-14 of Alg. 19. In this case potentials only have 1 or 2 variables, and in the case of a cluster \mathbf{C}_i the potential cannot be decomposed as a product of potentials of 1 variable, because it would mean that the variables are independent, in which case they would not constitute a connected component of G_ϕ .*

6.1 Learning RPTs from probabilistic potentials

- *Lines 7-12 of Alg. 20. In this case, potentials are defined for less than 3 variables. If a potential is defined just for 2 variables, then it cannot be decomposed as a product of potentials of 1 variable, because its associated dependence graph has only one component, which means that the variables cannot be independent according to ϕ .*
- *Lines 19-21 of Alg. 22. This case corresponds to potentials representing normalisation factors.*

The second property of minimal representations is a consequence of the fact that the Split nodes added (loops starting in line 6 of Alg. 21 and line 5 of Alg. 22) are labelled with variables from \mathbf{S}_1 . Whenever a new variable is added to \mathbf{S}_1 , the dependence graph is connected, and therefore the set of variables cannot be decomposed into two independent non-trivial subsets. Hence, a decomposition of the potential into two factors with non-empty disjoint sets of variables is not possible. This line of reasoning requires that the variables taken from \mathbf{S}_1 to conform chains of Split nodes are selected in the same order that they are introduced in \mathbf{S}_1 , since it is the order in which the connectivity of the dependence graph conditional on the variables was tested. □

Thus, our algorithm can be considered as a greedy algorithm to find RPTs of small size, which under certain conditions are minimal representations of a probabilistic potential. The intuitive idea behind the procedure for selecting nodes for conditioning (Split nodes) is to reach degrees of dependence among the remaining variables as low as possible. In other words, we try to condition on variables that make the remaining ones become independent or weakly dependent, in order to be able to represent the potential over them as a List node. The proof of Prop. 6 also requires that the variables chosen for *Split* nodes from \mathbf{S}_1 are selected in the same order in which they were inserted in \mathbf{S}_1 . This is the most natural way of performing the splitting, as this means to follow the heuristic that we used for selecting variables.

The algorithm could be modified to obtain minimal representations without exceptions, by trying to decompose the normalisation factors, $\phi^h(\mathbf{s}_2)^{1-n}$. The problem is that the exponent $(1-n)$ conveys that the independence relationships in ϕ^h are not necessarily the same as in $\phi^{h^{1-n}}$. More precisely, it holds that any

6.1 Learning RPTs from probabilistic potentials

factorisation of ϕ^h as a product of potentials can be translated into a product decomposition of $\phi^h(\mathbf{s}_2)^{1-n}$ and vice versa (for example if $\phi^h(\mathbf{s}_2) = f_1^h(\mathbf{r}_1)f_2^h(\mathbf{r}_2)$, then it is immediate to show that $\phi^h(\mathbf{s}_2)^{1-n} = f_1^h(\mathbf{r}_1)^{1-n}f_2^h(\mathbf{r}_2)^{1-n}$). However, the marginal independence relationships associated with $(\phi^h(\mathbf{s}_2))^{1-n}$ can be different of those of $\phi^h(\mathbf{s}_2)$, and therefore the union property conditional on configurations might not hold in the new potential. The consequence is that we can reach approximate decompositions even when the conditions of Prop. 5 are verified. The solution of this problem¹ consists of calling Algorithm 1 to decompose potential $\phi^h(\mathbf{s}_2)$ without the exponentiation, obtaining an RPT that can be transformed afterwards into an RPT for $\phi^h(\mathbf{s}_2)^{1-n}$, by raising all the values (x) or potentials (f) in the leaves to the power $1 - n$ (i.e. x^{1-n} or f^{1-n}).

Time complexity of the algorithm

With respect to the time complexity of our greedy algorithm, if we assume that $dom(\phi)$ contains n variables, that ϕ is represented by a probability table with direct access to a value giving a configuration, and that c is the maximum number of cases of each variable, then in the worst case our algorithm does not find any List node (these nodes reduce the complexity of posterior steps), and a full PT is built with depth $n - 1$ (we stop at potentials of size 2). Each level i ($i = 0, \dots, n - 2$) makes $(n - i)(n - i - 1)/2$ independence tests using mutual information. Each independence test needs to compute the marginal for two variables in a potential depending of $n - i$ variables which is of size c^{n-i} . The other operations in each node (computing the graph, its connected components and the potentials associated with the children) are of lower order. Since there are of order c^i nodes per level of the RPT, this results in an order of time complexity $O((n - i)(n - i + 1)c^{n-i})$, and summing over the different levels, we obtain $O(\sum_{i=0}^{n-2}((n - i)(n - i + 1)c^{n-i}))$, and this is of order $O(n^2c^n)$. It must be pointed out that the size of potential ϕ is exponential in the number of variables $m = c^n$. So, the algorithm time complexity is a polynomial function of the input size. If the algorithm finds List nodes, then the complexity is lower, as for a potential with n variables,

¹We have not considered it in our implementation because it is a minor issue in practice, as the normalisation factors will depend on few variables in general.

6.1 Learning RPTs from probabilistic potentials

a multiplicative factorisation decomposes a potential into two potentials with $n - 1$ variables (in the worst case), and by splitting on a variable X_i we obtain a number of children equal to the number of values of X_i , each one of them with $n - 1$ variables, and the number of possible values of a variable is, at least, 2.

6.1.5 Learning from data

The previously described methodology can be adapted to learn an RPT from a database. In general, we follow the same approach as we did when learning from probabilistic potentials, but now we take into consideration the particularities of the new problem. The main differences with the previous approach are:

- The way of computing the degree of dependence between variables is now performed using a Bayesian score.
- We do not consider the information gain as a way of increasing the flexibility of the algorithm, as this might include a higher level of error when learning from data.
- We do not need to compute normalisation factors in the same way as before, as normalised potentials are retrieved from the database.

The starting point is a database *cases* defined over a set of variables \mathbf{X} , and the aim of the algorithm is therefore to find a representation of the probability distribution encoded in *cases* as an RPT.

The proposed algorithm is defined in Alg. 23, and as it can be seen in the pseudocode, it is divided into four steps, iterating over them until finding a suitable representation for the encoded distribution of the data. The first step consists of building an auxiliary graph structure G_c with vertex set \mathbf{X} where a pair of variables $X_i, X_j \in \mathbf{X}$ will be linked if there is probabilistic dependence between them. More precisely, a link $X_i - X_j$ is present in G_c if the weight of the link between X_i and X_j is bigger than 0. We use the Bayesian Dirichlet equivalent metric [6] to measure the relation between pairs of variables, and hence, a link $X_i - X_j$ will be included only if

$$W(X_i, X_j) = BDe(X_i|X_j) - BDe(X_i) > 0. \quad (6.20)$$

6.1 Learning RPTs from probabilistic potentials

```
1 learn(database cases)
   Input: A database cases
   Output:  $\mathcal{RT}$ , an RPT for an approximation of the distribution in cases
2 begin
3   // Step 1: compute  $G_c$ 
4    $G_c \rightarrow$  graph for variables dependencies in cases
5   // Step 2: graph analysis
6    $G_c$  analysis looking for connected components
7   // Several scenarios are possible according to  $G_c$ 
8   if  $G_c$  is partitioned into components  $\mathbf{C} = \{\mathbf{C}_1 \dots \mathbf{C}_n\}$  then
9     // Step 3: multiplicative factorisation
10     $\mathcal{RT} \leftarrow \text{multiplicativeFactorisation}(\text{cases}, \mathbf{C})$ 
11  else
12    // Only one component: decomposition with Step 4
13     $\mathcal{RT} \leftarrow \text{contextSpecificFactorisation}(\text{cases}, G_c)$ 
14  end
15  return  $\mathcal{RT}$ 
16 end
```

Algorithm 23: Main body of the learning algorithm

The second step of the algorithm consists of analyzing the resultant graph G_c . A disconnected representation of G_c can be directly translated as a factorisation, jumping into the third step of the algorithm, because the separation between the clusters mean that the dependence between their variables is weak. The fourth step of the algorithm corresponds to the case when the graph G_c remains as a single connected component, which means that the potential is not decomposable as a list of factors with disjoint variables. However, conditional decompositions are possible, so the algorithm looks for either factorisations that share variables, context-specific independencies, or a combination of both.

Representing a factorisation

If G_c is disconnected in n connected components $\mathbf{X}_1 \cup \dots \cup \mathbf{X}_n = \mathbf{X}$. The factorisation is given by:

$$f(\mathbf{x}) = f_1(\mathbf{x}_1) \cdots f_n(\mathbf{x}_n), \quad (6.21)$$

where $f_i = f^{\downarrow \mathbf{X}_i}$, $i = 1, \dots, n$. Each f_i is a potential with the absolute frequencies for all the variables in the correspondent connected component. Therefore, f corresponds to the joint probability distribution of all the variables in the G_c .

If the clusters do not share any variables (as it happens, for instance, when G_c is disconnected the first time that we compute it in Alg. 23), we can normalise each potential independently using the Laplacian correction to avoid dealing with zero probability values. If the clusters share variables, then we apply the Laplace correction taking into account all the variables only when normalising the first factor. The other factors are normalised using the Laplace correction but conditioned to the common variables, in order to avoid the introduction of normalising errors. This set of *conditioning variables* is associated to the cluster until the end of the algorithm, so further factorisations will be correctly normalised.

If any of the clusters contains more than 2 variables, we recursively apply the algorithm to try to learn a factorised substructure from the database for the correspondent subset of variables. Hence, the distribution for the current cluster can be represented as an RPT where the root node would be a List node containing the factors in Eq. 6.21.

Analysing connected components

This part of the algorithm will work with a subset of the variables of the database and will iteratively perform a series of steps: first, locate the variable within the cluster that present the highest *degree of dependence* with respect to the others, and remove it from the graph; second, recompute the links in the reduced graph by weighting the relations between every pair of the remaining variables; third, analyse the graph: if it becomes disconnected, we can represent the cluster as a factorisation, if it becomes too small (2 variables) it means that a factorisation is not possible, hence we represent the cluster as a Potential node, retrieving

6.1 Learning RPTs from probabilistic potentials

the parameters from the database. The third possibility is that after removing a variable, the graph continues being connected, in which case we iterate selecting a new variable to be removed, and so on.

The above mentioned *degree of dependence* between every variable X_i and those other variables belonging to its neighborhood, $ne(X_i)$, is computed as:

$$V_{X_i} = \sum_{X_j \in ne(X_i)} W(X_i, X_j). \quad (6.22)$$

Every iteration of the loop selects the variable maximizing the degree of dependence in order to look for the context in which the underlying potential might be factorised:

$$X_{max} = \arg \max_{X_i} V_{X_i}. \quad (6.23)$$

Once X_{max} is selected it will be included in \mathbf{S}_1 or \mathbf{S}_2 . These are auxiliary vectors that represent two types of variables: those that are closely related to all the others (\mathbf{S}_1) and those variables that are highly dependent of only a subset of the variables in the component (\mathbf{S}_2). Note that the ordering in which the variables are selected is very important, as every deletion is dependent on the previous one. The splitting of the tree with the variables of \mathbf{S}_1 will be performed using a first-in-first-out fashion.

Therefore, X_{max} will be included in \mathbf{S}_1 if it is completely connected to the rest of variables in G_c . Otherwise it will be included in \mathbf{S}_2 . In both cases, X_{max} and its links will be removed from G_c producing a new graph $G_c^{X_{max}}$ that will be considered in further iterations.

Each remaining link $X_i - X_j$ is re-weighted according to the previously removed variables $\mathbf{X}_s = \mathbf{S}_1 \cup \mathbf{S}_2$, being only included the links that obtain a positive score:

$$W(X_i, X_j | X_s) = BDe(X_i | X_j, X_s) - BDe(X_i | X_s) > 0. \quad (6.24)$$

Obtaining a factorisation of a cluster

If we disconnect the graph during the procedure explained above, we build the correspondent RPT in the following way:

If \mathbf{S}_1 has variables, we take the first variable X_i in S_1 and build a List node with two children: one will be a Split node of X_i and the second a Potential node with the marginal probability distribution of X_i computed from the database, and normalised using Laplace but checking if X_i belongs to the conditioning set. Then, for every possible child of the Split node, we add the same structure for the next variable in \mathbf{S}_1 , and so on until we represent all the variables in the set. We have to take into account when learning the parameters both the Split nodes above and the possible list of conditioning variables in the Laplace normalisation.

Now, we follow the branches of the Split nodes consistent with every possible configuration of the variables in \mathbf{S}_1 , and at the leaves we store the factorisation according to the variables in \mathbf{S}_2 and the resultant clusters $\mathbf{C} = \{c_1, \dots, c_n\}$. The representation will be a factorisation computed as explained in Sec. 16, but taking into account that we include the variables in \mathbf{S}_2 to the set of variables of every subcluster:

$$f(\mathbf{c}_1, \dots, \mathbf{c}_n, \mathbf{S}_2) := \prod_{i=1}^n f^h(\mathbf{c}_i, \mathbf{S}_2),$$

Once a decomposition is performed, the algorithm is recursively applied to each and every potential obtained successively, until no further decomposition can be computed.

6.2 Score and Search approach

This section presents a search-and-score approach for determining an RPT structure that approximates the model given by a database. The basic operators to explore the search space are defined in Section 6.2.2, followed by the search strategy, explained in Section 6.2.3. The introduced search algorithm is divided into two stages: the first consists of determining the structure of the network, adding and removing arcs from the underlying model. The second stage consists of trying

to factorise every resultant CPT, tying parameters to keep the model normalised during the whole process. Results related to the log likelihood and size of the learned models are compared to PC and K2 algorithms, and detailed in Section 6.3. Experiments on different standardized data sets show that the presented algorithm achieves more accurate approximations than the baseline methods in all the considered databases, and usually more compact models than at least one of the compared methods. In Section 5.6 we present the general conclusions and give a list of possible modifications to the explained methodology, as a line of future research.

6.2.1 Motivation

Learning the structure of the model from a database is a fundamental issue when working with Probabilistic Graphical Models, as it is not always possible to have an expert at our disposal to choose the model that best fits the relationships between the variables. In the case of approximating distributions with Recursive Probability Trees, the search space is huge, so we need to design algorithms that shorten this space while exploring accurate candidates. In the context of a search-and-score method we propose to address this problem in two ways: first by defining a set of operators that consider a specific set of possible neighbours and secondly by setting a search heuristic that controls the dimension of the search space, allowing only a controlled set of neighbours to be considered at each stage.

An issue that must be taken into account when learning RPTs is the normalisation of the structure. In Chapter 4.4.4 we explained how the normalisation of a Conditional Probability Distribution (CPD) is translated into the addition of a new factor to the RPT containing all the parents of the node, increasing in this way the complexity of the model. In the algorithm proposed here, as we are learning a conditional probability distribution for each of the nodes in the network, we have to handle this problem in an efficient way to be able to recover compact structures as well as accurate. We propose to solve this issue by controlling the shape of the factorisations and introducing tied parameters every time we incorporate a factorisation in the model, computing the parameters as a maximum likelihood estimation restricted to the normalisation conditions.

6.2.2 Search and Score approach

In this Chapter we define the framework for learning RPTs using a Search and Score approach. This methodology has been widely used for structural learning of Bayesian networks, and consist of exploring the space of possible RPTs, evaluating every candidate and finally returning the one that gives a better score. In order to explore the search space, we define a set of operators that locally transform the current model. Afterwards, the model is evaluated with a score metric and if its performance is better than the current candidate, the first takes its place and the search continues. When there is no variation in the score achieved by the set of neighbours, the search stops and the better candidate is given as result.

Local operations

In order to search the space of possible RPTs during the learning process, some local operations must be defined. These operations will make local changes to the current learned RPT structure, in order to explore the neighbourhood and locate better candidates.

Add parents

This operator correspond to the addition of arcs in traditional structural learning methodologies. In the context of this work, the addition of a new parent of a node means the inclusion of Split nodes of the parent into the CPD of the son. The new parent is incorporated to the list of parents of the node and the later is deleted from the list of possible parents of the parent, as to avoid a conflict in the resultant model. The parameters are computed using a maximum likelihood estimation for each configuration defined by the branches of the RPT defining the CPD.

We will consider two variations of this operator, one that introduces a Split node of the new parent for every leaf of the current modelled CPD, and other operator that only introduces the Split nodes for concrete configurations of the rest of the parents, considering in this way context-specific independencies within

the distribution. If X_i is the new parent to be introduced in the CPD of X_j , we define:

- $AddSN(X_i, X_j)$ sets X_i as a parent of X_j , that means the inclusion of a Split node of X_i on all the leaves of the RPT representing the CPD of X_j given its parents.
- $AddSNConf(X_i, X_j)$ sets X_i as a parent of X_j but only for a concrete context, that means the inclusion of a Split node of X_i on the leaves of the RPT representing the CPD of X_j given its parents that are consistent with a specific configuration.

Remove parents

$RemoveSN(X_i, X_j)$ removes X_i as a parent of X_j , that means the erasure of the Split nodes related to X_i inside the factor related to X_j , as well as the removal of X_i from the list of parents of X_j , and the addition of X_j as a possible parent of X_i . This operation is conceptually equivalent to the removal of arcs when learning Bayesian networks.

Factorise CPD

The operator $Factorise(X_i)$ performs a factorisation of the RPT related to the CPD representing X_i conditioned to its parents. This operator has been designed to perform a specific factorisation of the RPT, where the parameters are tied and computed so as the model remains normalised. The motivation of this methodology is on one hand not to increase the complexity of the model, and secondly to restrict the search space, as the number of possible factorisations to be performed in a certain model can be intractable.

We can approximate the decompositions by tying the proportional parameters and modifying them along with the proportionality factor (maximizing the log-likelihood of the data) so the factor remains as a CPD. The process is illustrated in Fig. 6.16 and detailed in Algorithm 24. Note that we have only considered binary variables in this work for the sake of simplicity, but this methodology can be extended to cope with variables with more states.

6.2 Score and Search approach

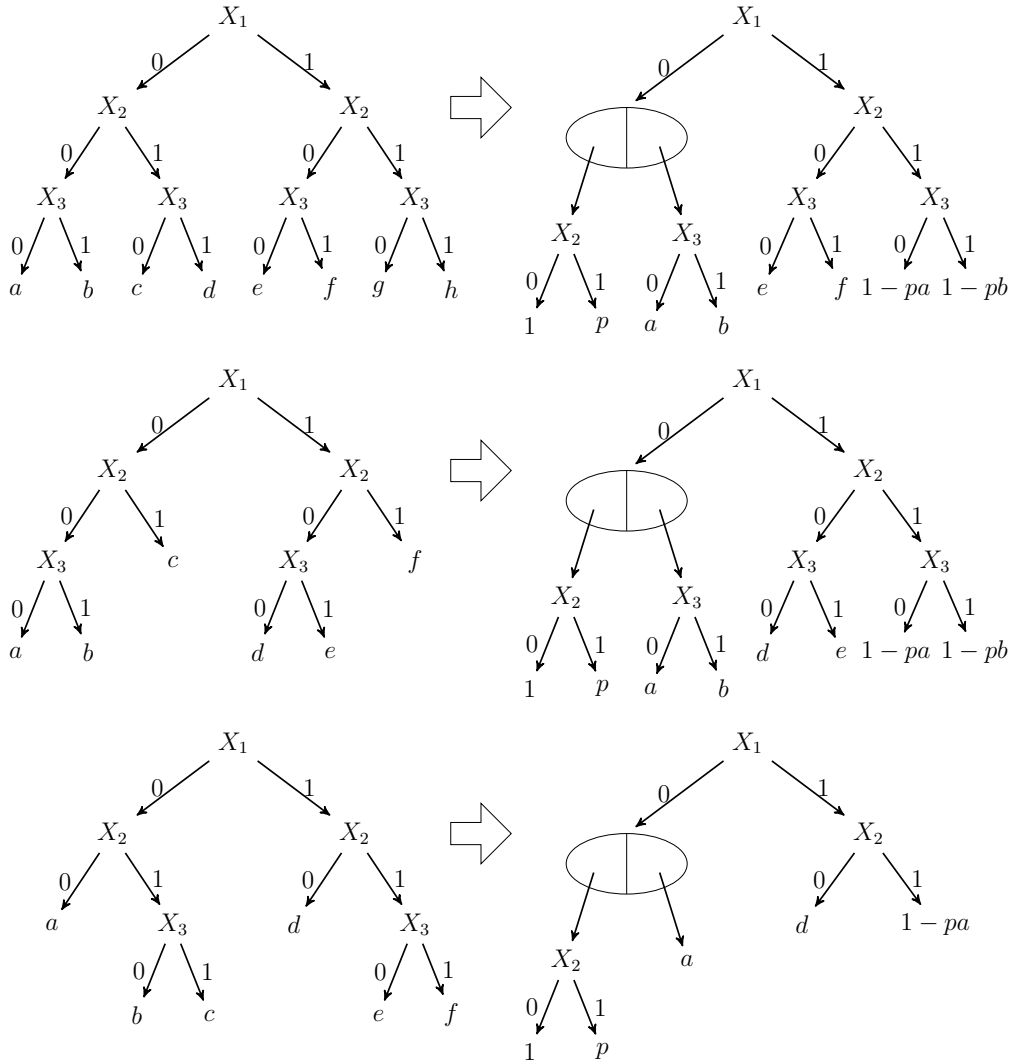


Figure 6.16: Tying parameters to factorise a CPT

```

1 factorise_cpd(RPT cpd)
  Input: An RPT cpd composed of at least 3 variables, and starting with a
           Split chain of at least two of them
  Output: An RPT rpdfact enclosing the performed factorisation
2 begin
3   Let root1 be the root of cpd (a Split node);
4   Let root2 be a Split node of the same variable as root1;
5   //Generate left son;
6   Generate a List Node list;
7   Let sonl be the left son of root1 (a Split node);
8   Let sonll be the left son of sonl;
9   Let sonl2 be a Split node of the same variable as sonl;
10  Let a Value node set to 1 be the left son of sonl2;
11  Compute p as the proportionality factor between the right and left sons
     of sonl by dividing the value corresponding to their first configuration;
12  Let a Value node set to p be the right son of sonl2;
13  Set sonl2 as son of list;
14  Set sonll as son of list;
15  //Generate right son;
16  Let sonr be the right son of root1 (a Split node);
17  Let sonrl be the left son of sonr;
18  Let sonr2 be a Split node of the same variable as sonr;
19  Set sonrl as left son of sonr2;
20  Let sonrr be a copy of sonll;
21  Recompute the parameters on the leaves as (1 - p*current value);
22  Set sonrr as right son of sonr2;
23  Set list as left son of root2;
24  Set sonr2 as right son of root2;
25  Let root2 be the root of an RPT rpdfact;
26  Return rpdfact;
27 end

```

Algorithm 24: Algorithm for factorising an RPT keeping it normalised as a CPD

6.2.3 Search strategy

The methodology that we propose here is composed of three stages. First, we define the first candidate to be evaluated, and the structure that will be the

starting point of the search. Afterwards, we analyse the relations between the parents, finding a promising set of parents for each node. The final stage of the algorithm consists on trying to factorise each conditional probability distribution of the model. Algorithm 25 details the followed methodology.

```

1 learn_RPT(cases)
  Input: A database cases
  Output: A Recursive Probability Tree rpt that approximates the
           distribution in cases
2 begin
3   Create initial model with Alg. 26;
4   for each CPD do
5     while there are possible parents for the node do
6       Insert a parent;
7       Evaluate the new model;
8       if the new model is better then
9         | Replace current model with factorised;
10      end
11      else
12        | Reverse the arc;
13        | Evaluate the new model;
14        | if the new model is better then
15          | | Replace current model with factorised;
16        | end
17      end
18    end
19  end
20  for each CDP do
21    | Apply Factorise operator (Alg. 24);
22    | Evaluate the new model;
23    | if the factorised model is better then
24      | | Replace current model with factorised;
25    | end
26  end
27  Return rpt that corresponds to the current model;
28 end

```

Algorithm 25: Algorithm for learning RPTs from data

In order to measure the quality of the considered candidates, we compute a

Bayesian metric that contains a penalty term. It is defined in Eq. 6.25. The penalty term takes into account the complexity of the model, in terms of number of probability values used to represent the distribution, to benefit more compact structures over larger ones.

$$\begin{aligned}
 BIC(RPT') &= \log(P(D|RPT')) - \frac{A}{2}\log(N) \\
 &= \sum_{i=1}^N \log P(d_i|RPT') - \frac{A}{2}\log(N) \\
 &= \sum_{j=1}^K \sum_{i=1}^N \log P(d_i^j|RPT') - \frac{A}{2}\log(N),
 \end{aligned} \tag{6.25}$$

Setting up the first candidate

The algorithm begins by building a first structure to serve as the starting point for the search. This first structure consists of an RPT which root is a *List* node with as many children as variables in the model. Each child would be then a *Split* node containing the marginal for each one of the variables in the model. The graph equivalent to this situation would be a fully disconnected one. The root List node contains an initial decomposition $\prod_{i=1}^N P(X_i)$, where N is the number of variables in the network. The parameters are learned from the database, maximizing the likelihood. Algorithm 26 details the construction of this initial candidate. After

the set up of this first candidate (line 3 of Alg. 25), the search procedure starts.

```

1 initial_setup(cases)
  Input: A database cases
  Output: A Recursive Tree Node root representing  $\prod_{i=1}^N P(X_i|cases)$ 
2 begin
3   Let root be a List Node;
4   Let N be the number of variables in cases;
5   for i ← 1 to N do
6     Let childi be a Split Node;
7     Let Xi be the ith variable in cases;
8     Set childi's parameters to fit  $P(X_i|cases)$ ;
9     Include childi in the children set of root;
10  end
11  Return root;
12 end

```

Algorithm 26: Initial setup for Learning RPTs from data

First stage: defining the set of parents

The first stage of the algorithm consists on adding and removing parents to and from every CPD in order to find a model that best fits the data among those explored (lines 4-19 of Alg. 25). We keep a list of possible parents to be introduced in every CPD. At the beginning of the algorithm, for each CPD $P(X_i|Pa(X_i))$, the *Candidates* list will be the full set of variables but X_i , and these lists are modified every time we apply the operators that add a parent and the operator that removes a parent to avoid possible conflicts in the final model, as bidirectional arcs or cycles, that are not allowed in the underlying Bayesian network that we are building.

Second stage: factorising CPTs

The second stage of the algorithm consists on visiting all the CPDs and try to apply the Factorise operator. Once a CPD is factorised, the model is evaluated. If the factorisation makes the model more accurate, the operation is fixed and the process continues trying to factorise the next CPD. This procedure corresponds to

lines 20 to 28 of Alg. 25. At this stage we have only considered a factorisation from the top of each tree, but this can be expanded to search for smaller factorisations down the RPTs.

6.3 Experimental evaluation

6.3.1 Learning RPTs from probabilistic potentials

This section presents a set of experiments conducted in order to test the behavior of the algorithm. Some of the experiments analyse the relation between the accuracy of the representation (in terms of the Kullback-Leibler divergence between the original and the learned distributions) and the size of the learned RPTs, and shed light on the problem of controlling the tradeoff between complexity and accuracy. We also introduce the possible benefits of applying this algorithm during the inference process.

The value of δ is not thoroughly studied in this work, remaining as a future line of research. Instead, δ has been fixed for every experiment so that the results are focused in the differences for the values of ϵ .

Learning from CPTs

The first experiment consists of analysing 30 randomly generated CPTs defined over 6 binary variables ($size(P) = 2^6$). The values for each CPT are generated at random, so no context specific independencies or any factorisations are guaranteed to be present on them. For each CPT the learning process is repeated with different threshold values (ϵ varying from 0.0 to 0.01 with an increment of 0.001). The value of δ (see Eq. (6.19)) is set to 0.5. After each execution, the Kullback-Leibler (KL) divergence is computed between the distribution represented in the resulting RPT and the original CPT.

The results are presented using boxplots, representing the full range of data obtained, and with whiskers spanning up to the most extreme data point that is no more than 1.5 times the interquartile range from the box. Fig. 6.17 (left part) shows the KL values (computed for all the RPTs) obtained for each value of ϵ . Higher values of ϵ yield worse approximations.

6.3 Experimental evaluation

There is a certain value of ϵ acting as a limit point: once surpassed this value, the error is highly increased. This fact can be used as a selection criterion for the admissible threshold. It is also observed that higher values of ϵ are linked to solutions almost completely factored (a *List* node with a factor per variable).

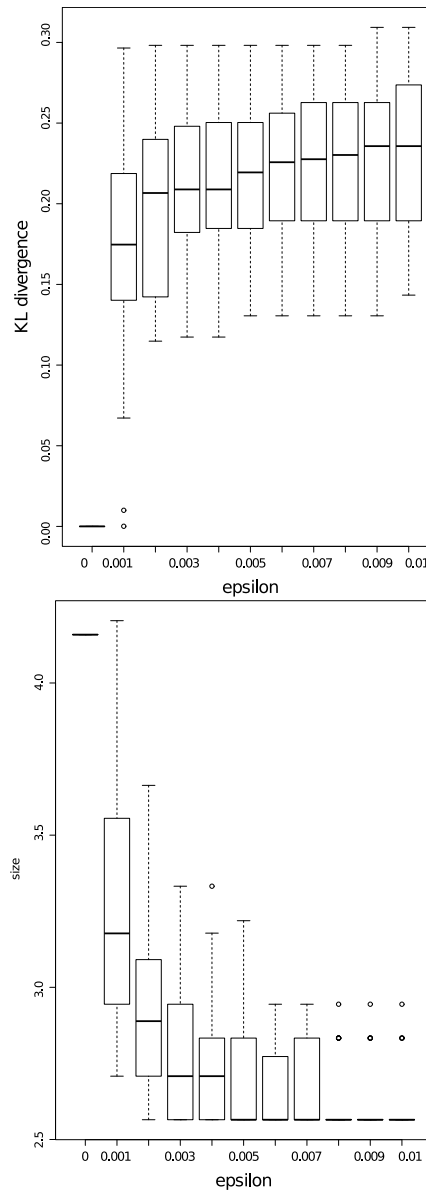


Figure 6.17: KL divergence and size (logarithm of number of probability values stored) of the representation for models learned with different thresholds.

The size of the learned model (in terms of the number of values stored to represent the distribution, in logarithmic scale) for each value of ϵ , is shown in the right part of Fig. 6.17. The size of the representation decreases as ϵ increases because more factorisations are introduced in the model (where the extreme solution would be an almost completely factored representation, i.e. a List node with a factor per variable). Fig. 6.18 depicts the quality of the representations obtained for small values of the threshold by applying the same experimental methodology to values of ϵ between 0 and 0.002. It shows how the error rate grows as the size of the obtained model decreases. This suggests the need to establish a selection criterion for an admissible threshold that tradeoffs accuracy and representation size, that could be a certain variation in the error rates obtained.

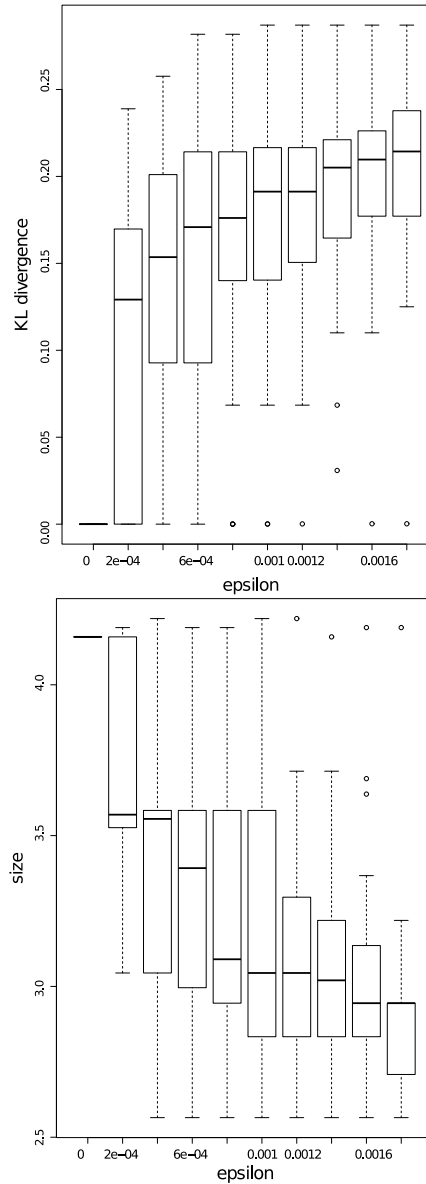


Figure 6.18: KL divergence and size (logarithm of number of probability values stored) of the representation for models learned with smaller thresholds.

Capturing repeated values

The second experiment aims at testing the ability of the algorithm for detecting context specific independencies. For that purpose, 30 randomly generated PTs were used. As in the previous experiment, no context specific independencies

or any factorisations are guaranteed to hold on these trees in this point. The generation of repeated values is attained by using the pruning operation: some sub-trees are replaced by a single value (its average value). Two limit values for the pruning operation are considered: 0.001 (soft pruning) and 0.01 (severe pruning, i.e. more sub-trees are replaced and more repeated values will be produced as well). For each PT the algorithm is executed with different values of ϵ : from 0.0 to 0.01 with an increment of 0.001. The parameter δ is set to 0.01. The right part of Fig. 6.19 shows KL divergence values for every ϵ value, along with a measure of the size of the RPTs obtained, for the case where soft pruning is applied. Fig. 6.20 gathers the results for severe pruning.

The error rate is higher when soft pruning is applied, which means that the algorithm is able to detect these context-specific independencies, providing better approximations when the distribution analyzed contains them. The left part of both Fig. 6.19 and Fig. 6.20 show the variation of the sizes of the representations obtained when increasing the value of the threshold. The structures obtained by Algorithm 18 when learning from severely pruned trees provide a better balance between accuracy and size of the representation.

6.3 Experimental evaluation

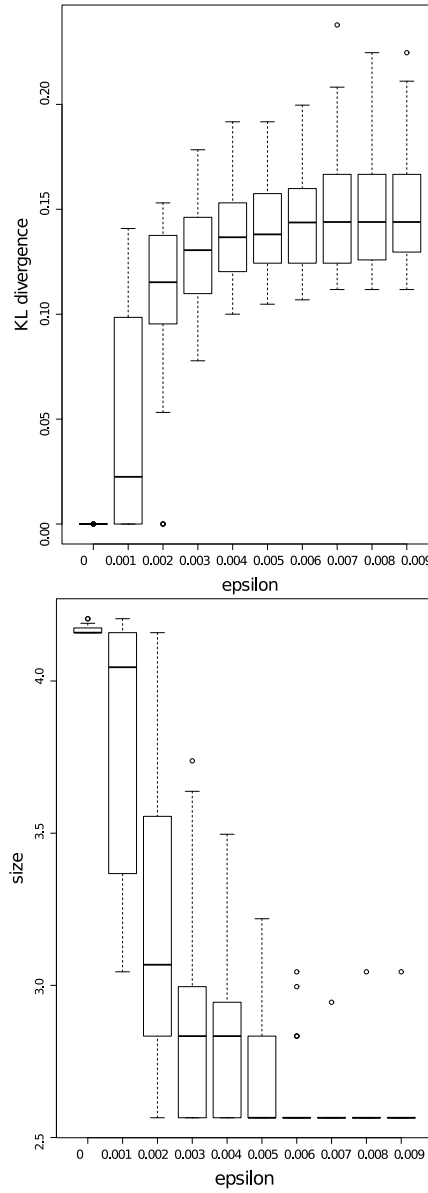


Figure 6.19: KL divergence and size (logarithm of number of probability values stored) of the representation learning from the same tree (slightly pruned) for different values of ϵ

6.3 Experimental evaluation

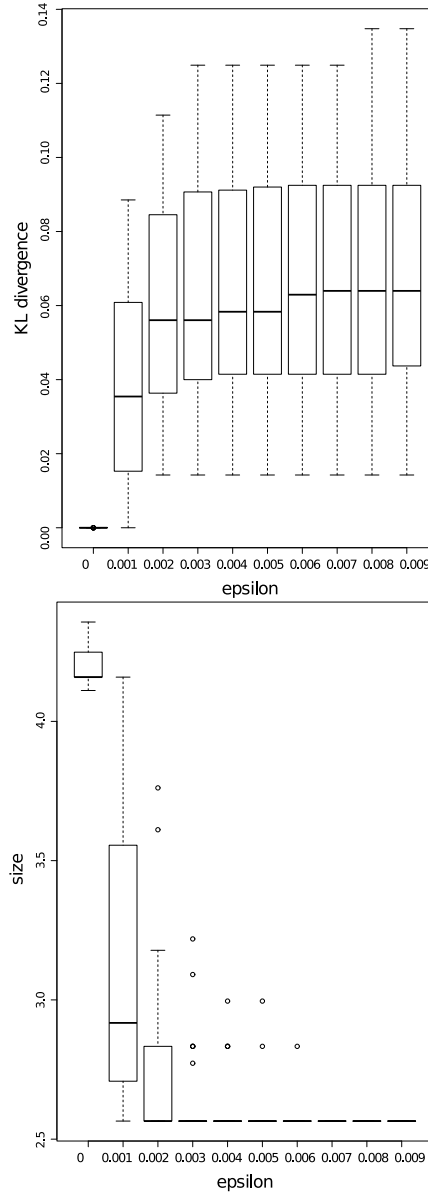


Figure 6.20: KL divergence and size (logarithm of number of probability values stored) of the representation learning from the same tree (severely pruned) for different values of ϵ

Other experiments have been performed to check the relation between accuracy and size of the learned model. The results show that models with reduced sizes are less accurate. This seems to point out that smaller representations are mainly *List* nodes containing simple factors (one or two variables). This factori-

sation is usually a poor representation of the potentials under analysis.

Learning from the same model

Due to the nature of the algorithm and the RPT structure itself, it is possible to get different RPTs representing the same distribution. The relation between different but alternative RPTs is considered in this experiment. Several PTs (30) are studied with the following strategy:

- 30 PT of 6 binary variables each, are generated at random;
- an RPT is learned for every PT, with $\epsilon = 0.002$ as threshold value, and $\delta = 0.001$ (this stage produces $RPT_{1i}, i = 1, \dots, 30$);
- RPT_{1i} is converted into a CPT and used for learning another RPT with $\epsilon = 0.0005$. These RPTs are denoted as $RPT_{2i}, i = 1, \dots, 30$.

It is likely that RPT_{1i} and RPT_{2i} be similar because they represent the same original distribution. The pairs $RPT_{1i} - RPT_{2i}$ are analyzed computing and storing the differences between the sizes of both RPTs (in terms of number of values stored to represent the distribution) and between the Kullback-Leibler divergence respect to the original PT. Regarding the logarithm of the tree sizes, the mean is 2.2 and the standard deviation 6.47. For Kullback-Leibler divergence the mean is 0.0015 and the standard deviation 0.004. The differences in the sizes between RPT_{1i} and RPT_{2i} suggest that the structures learned are different, whilst the low and stable KL rates confirm that both RPTs accurately approximate the original distribution.

Recovering a factorisation of a network from its joint probability distribution

The aim of this experiment is to test the accuracy of the factorisation performed by the algorithm, using as input the joint probability distribution of a Bayesian network. For the experiment, the Cancer network [22] (more details can be checked in Appendix 7.2.4) has been chosen due to its reduced dimension. The

6.3 Experimental evaluation

input of the algorithm is potential obtained by combining all the CPTs specified in the network, and the learning process was repeated varying the threshold values from $\epsilon = 0.0$ to 0.02 with an increment of 0.001. The Kullback-Leibler divergence between the distribution represented by each RPT generated and the original joint probability distribution was computed. The size of the representation, in terms of number of probability values stored, was measured as well. Fig. 6.21 shows that by increasing the ϵ value, the learned structures attain higher error rates, but at the same time, their size is lower. It is interesting to see how for values of ϵ between 0.005 and 0.009 the algorithm retrieves almost the same structure, obtaining the same value for the KL divergence and for the size of the structure. The same happens for the interval between 0.01 and 0.014. Fig. 6.22 shows the structure of the RPT learned for an ϵ value of 0.01.

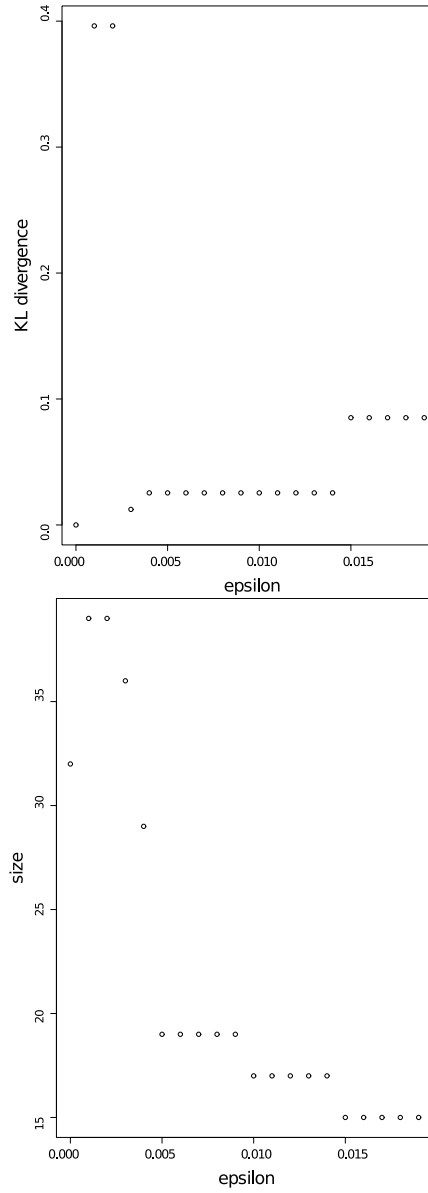


Figure 6.21: KL divergence variation between the joint probability distribution of Cancer network and the model learned, and size of the learned model, for different values of the threshold.

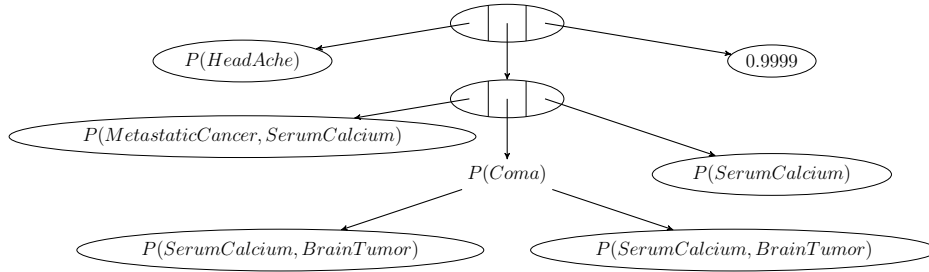


Figure 6.22: RPT learned from the Cancer network

Measuring the size of the RPTs used during the inference process.

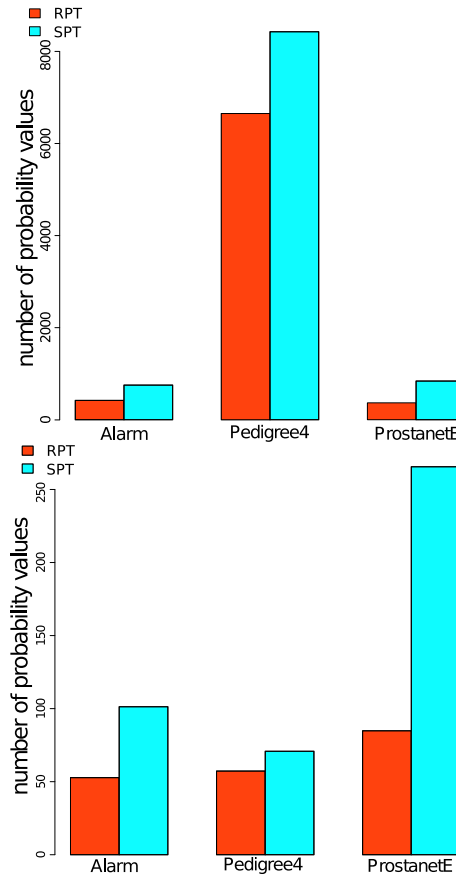


Figure 6.23: Largest and average structure sizes used during the inference process over different Bayesian networks.

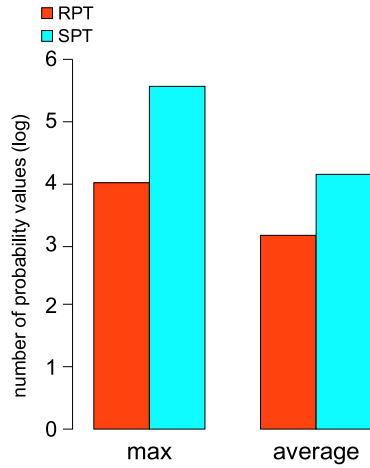


Figure 6.24: Largest (left) and average (right) structure sizes used during the inference process over Barley Bayesian network.

RPTs intend to be a compact structure to represent the probabilistic information in a Bayesian network. In this experiment we perform the variable elimination algorithm, once for each variable in the network, and with no observed variables, over different Bayesian networks whose CPTs are stored first as RPTs and then as PTs, in order to compare the size of the data structures used during the process, in terms of number of values that are stored for representing the distribution. The original probability tables are transformed into RPTs with an ϵ value of 0.05. The variable elimination algorithm was run over networks Alarm [78], Barley [66], Pedigree [44] and prostanetE [79] (details about the networks can be found in Appendix 7.2.4). Fig. 6.23 shows how the maximum and average size of the RPTs used are always lower than those of PTs. The results for Barley network are shown in Fig. 6.24, where the size of the representations is specified in logarithmic scale, due to the significant difference between the results obtained.

In order to check the accuracy of the representation, Table 6.1 presents the average and standard deviation of the KL divergence between the posterior distributions of each node of the network obtained using RPTs and that obtained with PTs. The PTs used are not pruned, so the solution they present is exact. The low values show that the approximation performed with RPTs is close to the original

in all the networks, even without applying an independent parametrisation to optimize the results for each network.

Table 6.1: Average and standard deviation for the KL divergence values obtained between the posterior distribution obtained for every node of each network, using RPTs and using PTs.

	Alarm	Pedigree	prostanetE	Barley
average	0.058	2.29E-5	0.057	0.029
s.d.	0.119	1.4E-4	0.065	0.066

6.3.2 Learning RPTs from data

In this section we present the experimental evaluation carried out in order to analyze the performance of the proposed algorithm. We learned from different kind of databases, both handcrafted and real, and examined the results both in terms of accuracy and size of the obtained representation.

Detecting factorisations and context-specific independencies in the data

The aim of this experiment was to check the accuracy and size of the learned RPTs. We sampled several RPTs with different degrees of factorisations and context-specific independencies within them, and then learned a new structure with the proposed algorithm. Afterwards we compared the learned structure to the original one, both measuring the Kullback-Leibler divergence between them, and counting the number of probability values needed to represent the distribution.

To do so, we used the random RPT generator explained above to build RPTs of 10 binary variables, and varied the probability of generating a Split node (p_S) between 0 and 1, with intervals of 0.1. The probability of generating a Potential node (p_P) in the leaves was set to 0.8. For each combination of the parameters,

the experiment was repeated 30 times. For each generated RPT we sampled a database of 100, 500, 1000, 2000 and 5000 entries.

Figure 6.25 shows the average of the 30 Kullback-Leibler divergence values obtained for each value of p_S . The Kullback-Leibler divergence shows generally reasonable accurate results, getting worse as the level of factorisations decrease in the original distributions. This means that the algorithm performs well at detecting clusters of highly dependent variables, and in general obtains good approximations of the distributions by detecting the patterns hidden within them. As for the number of samples in the database, we can see in Fig. 6.25 that small databases lead to poor representations, whilst too much data leads to overfitting. The best average results are obtained for the database of 2000 entries.

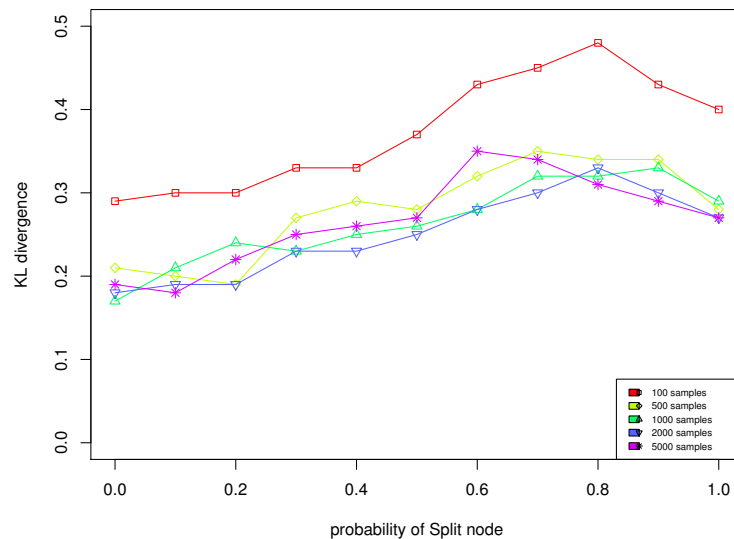


Figure 6.25: Average Kullback-Leibler divergence between the original distribution and the learned, for different RPTs and for different database sizes.

We also measured the sizes of both the original model and the learned structures for all the considered cases. In general, the learned RPTs are much more compact in all the cases. For instance, in Fig. 6.26 we see the averages of the 30 structures generated for each value of p_S , with the database of 2000 entries, where

we can check how the approximations are much smaller and, as seen in Fig. 6.25, still reasonably accurate. We do not show the figures for all the database sizes due to lack of space, but the observed behavior is constant for all the database sizes tested.

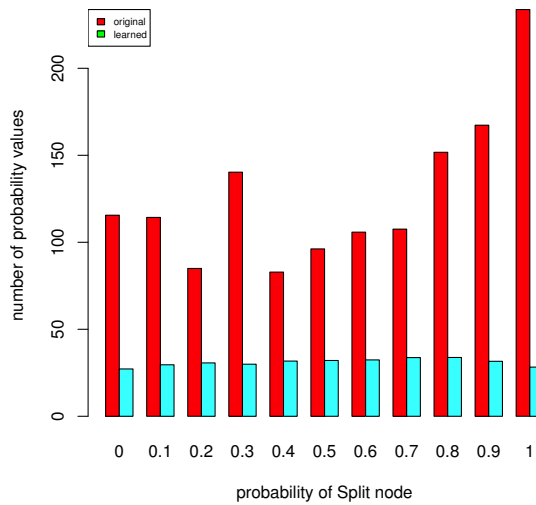


Figure 6.26: Size of the learned RPTs for the database of 2000 instances.

In general we can conclude from this experiment that the proposed algorithm returns compact structures representing good approximations of the original distributions.

Learning Bayesian networks

In this experiment we learned the RPT models from 6 databases extracted from the UCI Machine Learning Repository¹ (more details can be found in Appendix 7.2.4). For each database, we learned the RPT with 80% of the data, and then computed the loglikelihood with the remaining 20%. We compared the resultant accuracy with the models obtained by the PC and K2 algorithms.

¹<http://archive.ics.uci.edu/ml/>

6.3 Experimental evaluation

This procedure was repeated 30 times, every time varying the partition of the database. With the average of the loglikelihood and size (in terms of number of probability values stored) of the models we computed the Bayesian Information Criteria (BIC), that for small sample sizes penalizes more complex models. The results are shown in Fig.6.27, where we can see that RPTs get a better score than at least one of the other learned models with all the databases with the exception of Heart Disease, where all three algorithms obtain a very similar score. We have included the average BIC for all the considered networks, and we can see how RPTs tie with the PC algorithm (with an average of -869,57 (s.d. 283,54) for RPTs and -866,51 (s.d. 452,09) for the PC), whilst K2 results fall far behind (with an average of -1090,98 and standard deviation of 389,64)). A Friedman test revealed that there are no statistically significant differences between the algorithms ($\chi^2 = 11.95, p < 0.05$).

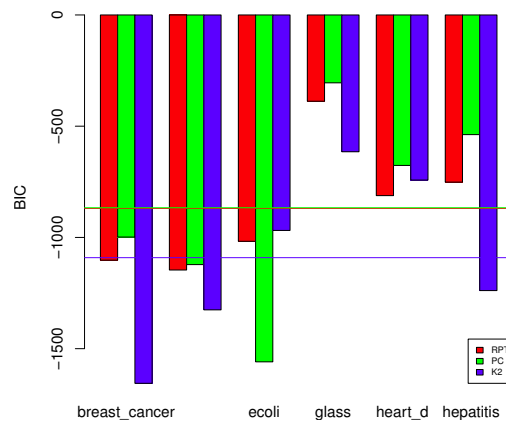


Figure 6.27: Bayesian Information Criteria for the learned models.

6.3.3 Score and Search approach

Learning Bayesian networks

In this experiment we learned the models from four databases extracted from the UCI Machine Learning Repository [80]. Appendix 7.2.4 gives some details on the chosen databases (Hepatitis, Glass Identification, Diabetes and Asia). All the continuous variables were discretized, so in the end all the variables had two states. For each database, we learned the RPT with 70% of the data, and then computed the loglikelihood with the remaining 30%. We compared the resultant accuracy with the PC algorithm and K2. This procedure was repeated ten times, and the average of the log likelihood obtained in each case is shown in Fig.6.28. In general the obtained log likelihood results for the RPTs are better than the other approaches, and in Fig. 6.29 we can see how the RPTs are generally more compact representations than at least one of the other approaches taken into account.

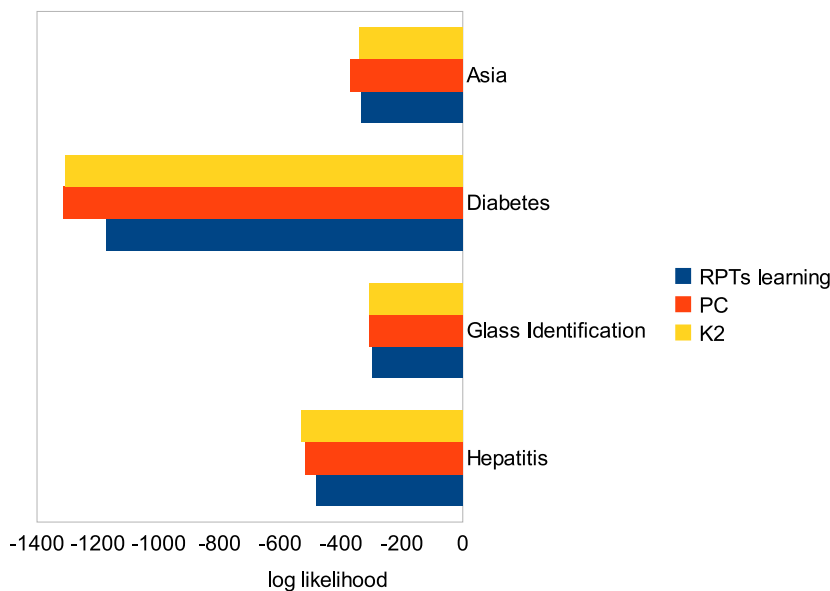


Figure 6.28: Log likelihood of the learned models.

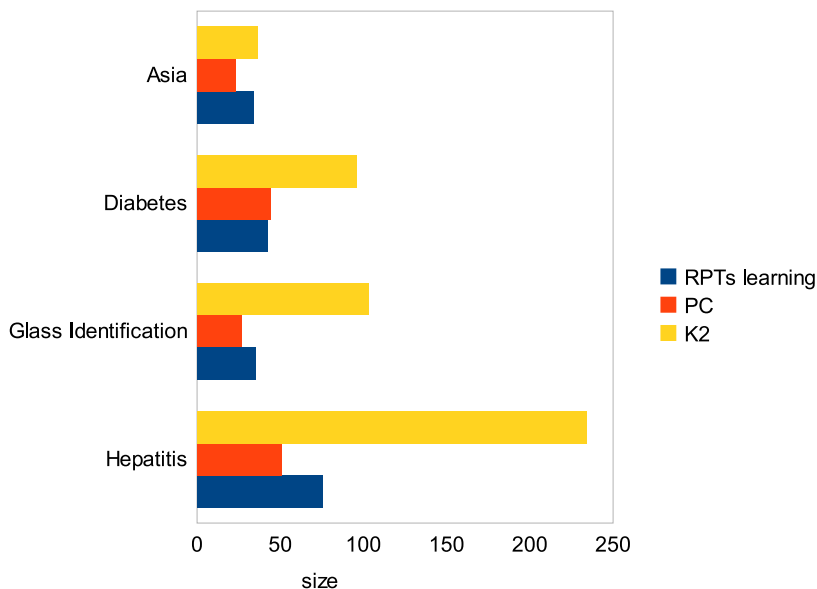


Figure 6.29: Size of the learned models, in terms of number of probability values used to represent them.

6.4 Conclusions and Future Work

In this chapter we propose an algorithm for learning an RPT from a probabilistic potential. The complexity of the problem of learning minimum size RPTs as well as the time complexity and some properties of the algorithm have been theoretically analyzed. The experiments conducted suggest that the algorithm is able to capture most of the details of the original distribution. This proposal can be used as the basis for designing new approximate algorithms for inference in probabilistic graphical models using RPTs during the inference process, instead of CPTs or PTs.

Notice that the algorithm is limited in practice by the size of the distribution to learn from: the distributions used for computing G_P are obtained by marginalizing the original potential P . Therefore, a representation of P allowing efficient computations of marginals certainly improves the performance of the algorithm.

The second contribution of this chapter is the extension of the previous algorithm to learn an RPTs from a database, looking for factorisations and context-

specific independencies within the data. The experiments suggest that the algorithm retrieves accurate and compact representations for the underlying distributions, being competitive against algorithms like PC and K2. In this chapter we have only considered the BDe metric for weighting the relations between variables, but other measures, like the mutual information, can be considered and tested against each other in future works.

Finally, we introduce the idea of designing a traditional score-and-search algorithm for learning RPTs from data. We have proposed many modifications to the different parts of the algorithm, that remains as a future line of research. We have also presented a preliminary experimentation where we show how the obtained models are good and compact representations of the underlying distributions represented by well known data bases present in the literature. We have compared the results of our algorithm to two well known algorithms, obtaining comparable results. As a first approximation to addressing the problem, the results are promising.

Part V

Conclusions

Chapter 7

Conclusions and Future Work

This chapter brings together all the conclusions that have been presented separately on each chapter of this work, lists all the publications where the ideas were presented and concludes with an enumeration of the future research to be performed on the topic.

We have presented throughout this work a data structure for representing probabilistic information. Recursive Probability Trees present an interesting framework to work with Probabilistic Graphical Models, allowing the modelling of different patterns usually found within probability distributions, as context-specific independencies, proportionalities and other types of factorisations. Through the experimental evaluation, RPTs have been proven to be a good alternative to traditional data structures like CPTs or PTs.

RPTs also compact the information and keep it factorised during the inference, allowing the design of algorithms that take this into account in order to speed up the process. We have designed an algorithm that efficiently computes optimal factorisations within probability potentials, as a first step towards the incorporation of RPTs to inference algorithms. Also, we have provided a fast-to-compute measure called the *factorisation degree*, that provides a heuristic to rank the variables in the domain of a potential according to the accuracy of the decompositions that they induce. This measure can be included within inference algorithms, for instance, to build RPTs when necessary.

We have also presented different approaches at addressing the problem of learning RPTs. The first developed algorithm aims at transforming a probabilis-

tic potential stored in any data structure into an RPT by looking for patterns within it. This algorithm uses the mutual information as a measure of the dependency between the variables. Also we have addressed the problem of learning RPTs from data from two different perspectives. Firstly we have adapted the methodology explained before to cope with this problem, using a Bayesian score as a dependency measure. Besides we propose a learning algorithm based in a Search and Score methodology, that gives reasonably good results, and brings up a variety of possible lines of related research.

7.1 List of Publications

The majority of the work presented in this dissertation has been published in the following references (some of them still in revision process):

- [1] A. Cano, M. Gómez-Olmedo, S. Moral, and C. Pérez-Ariza, “Recursive probability trees for Bayesian networks” in: *Lecture Notes in Artificial Intelligence (CAEPIA 2009)*, vol. 5988, pp. 242–251, 2009.
- [2] A. Cano, M. Gómez-Olmedo, S. Moral, C. Pérez-Ariza, and Antonio Salmerón “Inference in Bayesian Networks with Recursive Probability Trees: data structure definition and operations” in: *International Journal of Intelligent Systems*, Wiley, vol. 28, Issue 7, pp.623–647, 2012.
- [3] A. Cano, M. Gómez-Olmedo, S. Moral, C. Pérez-Ariza, and A. Salmerón, “Learning recursive probability trees from probabilistic potentials” in: *Proceedings of the Fifth European Workshop on Probabilistic Graphical Models (PGM-2010)* (P. Myllymäki, T. Roos, and T. Jaakkola, eds.), pp. 49–57, 2010.
- [4] A. Cano, M. Gómez-Olmedo, S. Moral, C. Pérez-Ariza, and A. Salmerón, “Learning Recursive Probability Trees from Probabilistic Potentials” in: *International Journal of Approximate Reasoning*, Elsevier, vol. 53, pp. 1367–1387, 2012.

- [5] A. Cano, M. Gómez-Olmedo, S. Moral, C. Pérez-Ariza, and A. Salmerón, “Learning recursive probability trees from data” accepted for publication in: *Lecture Notes in Artificial Intelligence (CAEPIA-2013)*, 2013.
- [6] A. Cano, M. Gómez-Olmedo, C. Pérez-Ariza, and A. Salmerón, “Fast factorization of probability trees and its application to recursive trees learning” in: *Combining Soft Computing and Statistical Methods in Data Analysis* (C. Borgelt, G. González-Rodríguez, M. Lubiano, M. Gil, P. Grzegorzewski, O. Hryniewicz, eds.), pp. 6572, 2010.
- [7] A. Cano, M. Gómez-Olmedo, C. Pérez-Ariza, and A. Salmerón, “Fast factorization of probability trees and its application to approximate inference in Bayesian networks” in: *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*, vol. 20, Issue 2, pp. 223-243, 2012.

Other publications

These publications are related to topics not discussed in this dissertation, however my contribution on them has enforced my training in learning Bayesian networks and managing databases with a high degree of context-specific independencies, bringing about the possibility of incorporating in the future Recursive Probability Trees to some of the issues being addressed.

- [8] C. B. Pérez-Ariza, A. E. Nicholson, K. B. Korb, S. Mascaro, and C. H. Hu, “Learning dynamic Bayesian networks using causal discovery” in: *Proceedings of the 25th Australasian Joint Conference on Artificial Intelligence*, 2012.
- [9] C. B. Pérez-Ariza, A. E. Nicholson and M. J. Flores, “Prediction of Coffee Rust Disease Using Bayesian Networks” in: *Proceedings of the Sixth European Workshop on Probabilistic Graphical Models (PGM-2012)* (A. Cano, M. Gómez-Olmedo, and T. D. Nielsen, eds.), pp. 259–266, 2012.

7.2 Future Work

In this section we gather together the future lines of research that were commented on each of the chapters of this dissertation. Basically this work provides some answers to three main questions, that can be defined as three research lines flowing in parallel: definition of the data structure and its capabilities, inference over the structure and finally, learning of RPTs. As discussed in every related chapter, the three lines present a huge variety of possibilities for further research. In the following we summarise some of those ideas.

7.2.1 Structure definition

So far, the only way of expressing decompositions where the terms are added instead of multiplied is by including auxiliar variables in the RPT that are marginalised out at some point to compute the result of the decomposition. In order to increase the flexibility of RPTs to freely express this addition of factors, and handle them efficiently, we plan to extend the data structure with the addition of a new type of node, a sum node, in a similar way as the approach followed with NIN-AND trees[65].

Also, in this work we have so far covered only models with discrete variables, but we plan to incorporate continuous variables to the structure, making RPTs able to deal with more realistic models.

7.2.2 Inference

Related to inference, we plan to further investigate the concepts presented in Chapter 5, with the idea of incorporating the fast factorization method and the *factorisation degree* measure to existing inference algorithms, in order to factorise potentials that become very big during the inference. With this we aim at reducing the time complexity of inference algorithms. We also plan to design new inference algorithms that take into account the factorised nature of RPTs and benefit from it.

7.2.3 Learning

Designing new algorithms for learning RPTs is a future line of research, specially after modifying the data structure to deal with sums of complex factors and continuous variables. Focusing in the current state of the structure, we want to modify the Search and Score procedure presented in Chapter 6 by adding new operators and modifying the existents, specially the Factorize operator, as it can be designed in many different ways to explore different types of factorisations. The search startegy is also extensible. So far we have followed a greedy approach, but we plan to apply more sofisticated and efficient startegies.

7.2.4 Applications

Finally, a potential line of future research is the practical application of RPTs. As RPTs stand as a compact and efficient data structure to work with probabilistic potentials, they can be applied to problems where the storage and computational resources are limited, as it is the case of mobile devices. The idea is to build an adaptation of RPTs that optimize the computation time and the size of the structures, to hold the probabilistic information handled in mobile applications that use Probabilistic Graphical Models.

Appendix

Coding in *Elvira*

Recursive Probability Trees and all the related algorithms described in this dissertation have been developed in Java under Elvira [81], an open-source software package for working with probabilistic graphical models.

The Elvira project started in 1997 as a joint collaboration of several Spanish universities, being the University of Granada among them. The exchange of ideas and the joint work led to the development of a wide tool that contains several algorithms for inference and learning of probabilistic graphical models, being specialized in Bayesian networks and influence diagrams. Elvira, through its graphical user interface, makes probabilistic graphical models available to everyone. Although some researchers still contribute to expand Elvira, a new system is under development with the aim of fixing some efficiency and organization problems present in Elvira: ProGraMo [82].

Data structure

The class hierarchy to build Recursive Probability Trees as explained in Chapter 4 is represented through an UML diagram in Fig. 1. It has been designed so the incorporation of new types of nodes becomes a relatively easy task.

The data structure is composed of the following classes:

- RecursivePotentialTree.java
 - This is the main class that defines the structure and its basic functionality.

- `TreeNode.java`
 - This abstract class generalizes the types of nodes that a Recursive Probability Tree can manage.
- `SplitNode.java`
 - This class contains a Node that labels the Split. So far we have only considered *FiniteStates* nodes, that in Elvira correspond to discrete variables, but we plan to make this kind of node able to work with continuous variables as well.
- `ListNode.java`
 - This type of node contains an array of Recursive Probability Trees.
- `ValueNode.java`
 - This node contains a numerical value.
- `PotentialNode.java`
 - This kind of node contains a generic Potential. This allows the incorporation of any data structure to RPTs in Elvira, as long as it works with discrete variables.

Inference with Recursive Probability Trees

The Variable Elimination algorithm is implemented in Elvira as a general framework that is specified for each available data structure. The setting was extended so we could run it on Recursive Probability Trees by adding a new class to the hierarchy, as shown in Fig. 2.

The functionality explained in Chapter 5 was implemented in Class `PotentialTree`. The location of this class with respect to RPTs in the code is shown in the UML diagram in Fig. 3. The main methods that develop the functionality for fast factorisation of probability trees are the following:

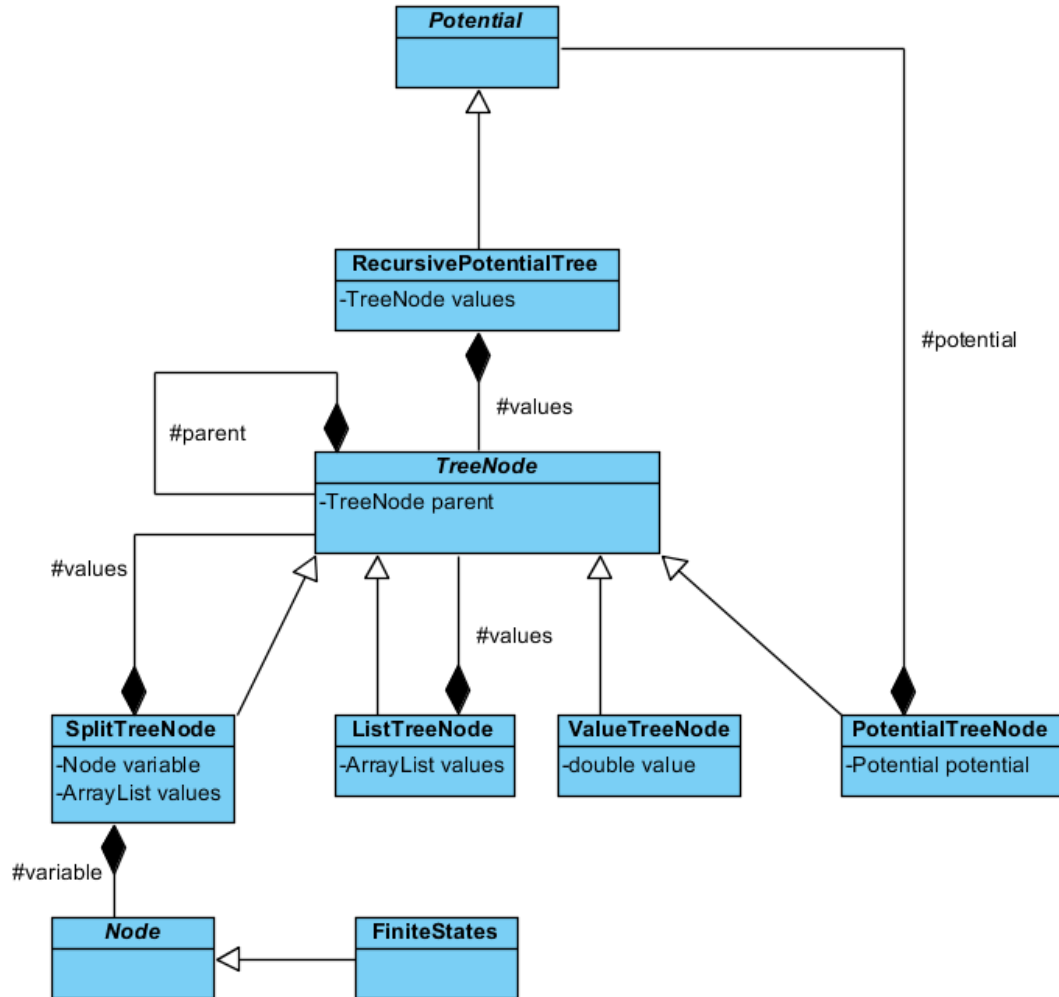


Figure 1: UML diagram for the RPTs classes

- `public double factorisationDegree2(FiniteStates x)`
 - Computes an upper bound of the *factorisation degree* of a tree for a given variable, using Jensen’s inequality (Eq. 5.9).
- `public Vector<PotentialTree> factoriseRT(Vector<FiniteStates>x)`
 - Factorises a tree as a list of two factors, as explained in Algorithm 16

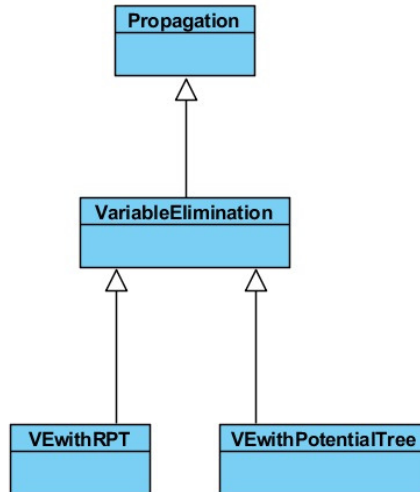


Figure 2: UML diagram for the VariableElimination classes

- `public Vector<FiniteStates>sortByFactorisationDegree()`
 - Returns the variables in the potential, sorted by their *factorisation degree*, in ascending order.
- `public FiniteStates bestToFactorise()`
 - Returns the variable with highest *factorisation degree*.

Learning Recursive Probability Trees

The first algorithm explained in Chapter 6 has been coded in class `learningRPT`. To use this algorithm, it is necessary to generate an instance of the class using the constructor. It asks for two parameters, that are the two thresholds that the algorithm requires. Afterwards, we call the method `factorize(Potential argPot)`, that takes the potential to be factorised as an argument. The result will be a `TreeNode` that is the root of the RPT that represents the potential.

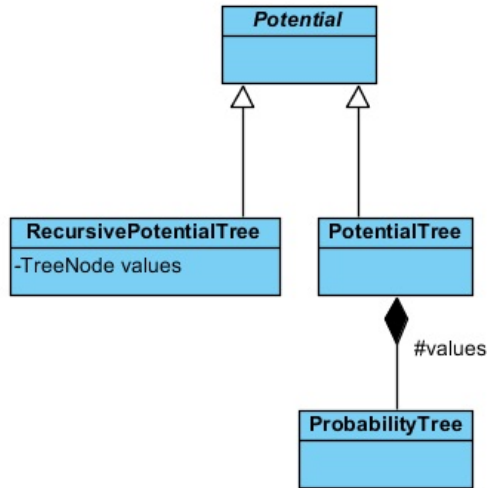


Figure 3: UML diagram for the ProbabilityTree classes

The second algorithm explained in Chapter 6 has been coded in class `learningRPT_DB`, that works in a similar way as `learningRPT`, but this time the only parameter that it needs is the database, as there are no thresholds in this implementation. Again, it is needed to instantiate the class using a constructor that asks for the database as a parameter, and then call the main method `learn()`.

The algorithm for learning RPTs using a search and score methodology as presented at the end of Chapter 6, has been coded using two classes under the package `elvira.potential.learningRPTS`. The UML diagram that shows these classes is in Fig. 4

- `LearningModel`
 - Main class for the learning algorithm.
- `CPT_rpt`
 - This class manages a conditional probability distribution.

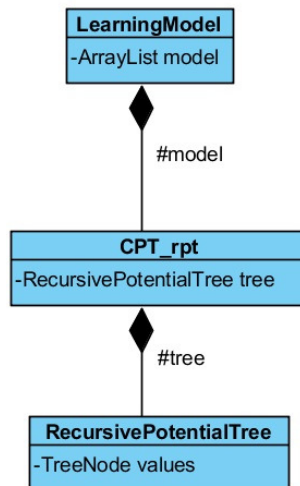


Figure 4: UML diagram for the search and score learning algorithm

Testing the code

All the code has been tested using a testing framework for the Java programming language: JUnit [83]. JUnit builds a parallel hierarchy of classes with the aim of containing a test method for each public method in the tested class. This test suite encapsulates the procedure so it automatically checks all the implemented functionality.

Appendix

Databases

This appendix reviews the databases that have been used in the experimentation of the work discussed in this thesis.

UCI repository

The following networks are publicly available at the UCI Machine Learning Repository¹. Here we give a brief summary of the used databases.

Hepatitis

Table 1: Details of Hepatitis dataset.

Number of Attributes	19
Number of Instances	155
Discretized?	yes
missing values?	yes

Dr. Peter Gregory of Stanford Hospital observed 155 chronic hepatitis patients, of which 33 died from the disease. On each patient were recorded 19 covariates summarizing medical history, physical examinations, x-rays, liver function tests,

¹<http://archive.ics.uci.edu/ml/>

and biopsies. An effective prediction rule, based on these 19 covariates, was desired to identify future patients at high risk. Such patients require more aggressive treatment.

Comments: The data was donated by Gail Gong.

Glass Identification

Table 2: Details of Glass Identification dataset.

Number of Attributes	10
Number of Instances	214 (163 after preproc.)
Discretized?	yes
missing values?	no

The study of classification of types of glass was motivated by criminological investigation. At the scene of the crime, the glass left can be used as evidence...if it is correctly identified!

Ecoli

Table 3: Details of Ecoli dataset.

Number of Attributes	8
Number of Instances	336
Discretized?	yes
missing values?	no

This data contains protein localization sites.

Diabetes

Table 4: Details of Diabetes dataset.

Number of Attributes	20 (9 after preproc.)
Number of Instances	768
Discretized?	yes
missing values?	yes

Diabetes patient records were obtained from two sources: an automatic electronic recording device and paper records. The automatic device had an internal clock to timestamps events, whereas the paper records only provided "logical time" slots (breakfast, lunch, dinner, bedtime). For paper records, fixed times were assigned to breakfast (08:00), lunch (12:00), dinner (18:00), and bedtime (22:00). Thus paper records have fictitious uniform recording times whereas electronic records have more realistic time stamps.

Comments: We used a reduced version of this database, with just 9 variables out of the original set.

Breast Cancer

Table 5: Details of Breast Cancer dataset.

Number of Attributes	9
Number of Instances	286
Discretized?	yes
missing values?	yes

This is one of three domains provided by the Oncology Institute that has repeatedly appeared in the machine learning literature.

Heart Disease

Table 6: Details of Heart Disease dataset.

Number of Attributes	14
Number of Instances	303 (294 after preproc.)
Discretized?	yes
missing values?	yes

This database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In particular, the Cleveland database is the only one that has been used by ML researchers to this date. The "goal" field refers to the presence of heart disease in the patient. It is integer valued from 0 (no presence) to 4. Experiments with the Cleveland database have concentrated on simply attempting to distinguish presence (values 1,2,3,4) from absence (value 0). The names and social security numbers of the patients were recently removed from the database, replaced with dummy values.

Other models

The following models are full Bayesian networks detailed in previous works that have been used in our experimentation.

Andes

Table 7: Details of Andes dataset.

Number of Nodes	223
Number of Arcs	338

ANDES is an Intelligent Tutoring System for Newtonian physics. ANDES' student model uses a Bayesian network to do long-term knowledge assessment, plan recognition and prediction of students' actions during problem solving [73].

Water

Table 8: Details of Water dataset.

Number of Nodes	32
Number of Arcs	66

This Bayesian network models an expert system for control of waste water treatment [74].

Barley

Table 9: Details of Barley dataset.

Number of Nodes	48
Number of Arcs	84

This Bayesian networks corresponds to a decision support system for growing malting barley without use of pesticides. One module in this system is the decision support system for mechanical weed control in malting barley. The module for weed control describes the relative reduction on the yield and the dry weight of weeds remaining in the field under a variety of conditions. The most important conditions included in the model are the amount of weeds in the spring, different methods of mechanical weed control, the row distance and the application method of nitrogen [66].

Munin1

Table 10: Details of Munin dataset.

Number of Nodes	189
Number of Arcs	282

This Bayesian network describes a diagnosis tool for diseases affecting the median nerve. This network is a graph with loops (not a tree) [72].

Cancer

Table 11: Details of Cancer dataset.

Number of Nodes	5
Number of Arcs	5

Metastatic cancer is a possible cause of a brain tumour and is also an explanation for increased serum calcium. In turn, either of these could explain a patient falling into a coma. Sever headache is also possibly associated with a brain tumour [22].

Comments: original work by Cooper.

Alarm

Table 12: Details of Alarm dataset.

Number of Nodes	37
Number of Arcs	46

This Bayesian network contains knowledge by medical experts for monitoring patients in intensive care [78].

Pedigree4

Table 13: Details of Pedigree dataset.

Number of Nodes	441
Number of Arcs	592

This Bayesian network is a subnetwork of an extremely complex real-world problem, namely estimation of genotype probabilities, for individuals in a heavily inbred pedigree containing approximately 20000 breeding pigs. Each individual may have a hereditary trait, PSE, which causes the meat to be unfit for human consumption [44].

ProstanetE

Table 14: Details of Prostanet dataset.

Number of Nodes	47
Number of Arcs	81

This Bayesian network was designed to help the diagnosis of prostate cancer. Prostate cancer is a very common disease in men over 50. However, sometimes it is not easy to diagnose it because it has symptoms and signs very similar to those produced by other benign diseases. Prostanet is a causal Bayesian network designed to help doctors to make a differential diagnosis between certain diseases related to the prostate [79].

References

- [1] T. V. D. Geiger and J. Pearl, “Identifying independencies in Bayesian Networks,” *Networks*, vol. 5, pp. 507–534, 1990. [9](#)
- [2] A. Cano, S. Moral, and A. Salmerón, “Penniless propagation in join trees,” *International Journal of Intelligent Systems*, vol. 15, pp. 1027–1059, 2000. [9](#), [22](#), [37](#), [58](#), [82](#)
- [3] D. Kozlov and D. Koller, “Nonuniform dynamic discretization in hybrid networks,” in *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence* (D. Geiger and P. Shenoy, eds.), pp. 302–313, Morgan & Kaufmann, 1997. [9](#)
- [4] R. Bouckaert, E. Castillo, and J. Gutiérrez, “A modified simulation scheme for inference in Bayesian networks,” *International Journal of Approximate Reasoning*, vol. 14, pp. 55–80, 1996. [9](#)
- [5] D. Geiger and D. Heckerman, “Knowledge representation and inference in similarity networks and Bayesian multinets,” *Artificial Intelligence*, vol. 82, pp. 45–74, 1996. [9](#), [68](#), [80](#)
- [6] D. Heckerman, D. Geiger, and D. Chickering, “Learning Bayesian networks: The combination of knowledge and statistical data,” *Machine Learning*, vol. 20, pp. 197–243, 1995. [10](#), [42](#), [43](#), [188](#)
- [7] C. Butz, K. Konkel, and P. Lingras, “Join tree propagation with prioritized messages,” *Networks*, vol. 55, pp. 350–359, 2010. [22](#)

- [8] C. Butz, K. Konkel, and P. Lingras, “Join tree propagation utilizing both arc reversal and variable elimination,” *International Journal of Approximate Reasoning*, vol. 52, pp. 948–959, 2010. [22](#)
- [9] C. Butz, A. Madsen, and K. Williams, “Using four cost measures to determine arc reversal orderings,” *Lecture Notes in Artificial Intelligence EQS-CARU’2011*, vol. 6717, pp. 110–121, 2011. [22](#)
- [10] F. Jensen, S. Lauritzen, and K. Olesen, “Bayesian updating in causal probabilistic networks by local computation,” *Computational Statistics Quarterly*, vol. 4, pp. 269–282, 1990. [22](#), [31](#), [35](#)
- [11] A. Madsen and F. Jensen, “Lazy propagation: a junction tree inference algorithm based on lazy evaluation,” *Artificial Intelligence*, vol. 113, pp. 203–245, 1999. [22](#), [35](#), [89](#)
- [12] A. Madsen, “Improvements to message computation in lazy propagation,” *International Journal of Approximate Reasoning*, vol. 51, pp. 499–514, 2012. [22](#), [89](#)
- [13] P. Shenoy, “Binary join trees for computing marginals in the Shenoy-Shafer architecture,” *International Journal of Approximate Reasoning*, vol. 17, pp. 239–263, 1997. [22](#)
- [14] A. Cano, S. Moral, and A. Salmerón, “Lazy evaluation in Penniless propagation over join trees,” *Networks*, vol. 39, pp. 175–185, 2002. [22](#), [35](#), [37](#)
- [15] V. Gogate and R. Dechter, “Samplesearch: Importance sampling in presence of determinism,” *Artificial Intelligence*, vol. 175, pp. 694–729, 2011. [22](#)
- [16] S. Moral and A. Salmerón, “Dynamic importance sampling in Bayesian networks based on probability trees,” *International Journal of Approximate Reasoning*, vol. 38, pp. 245–261, 2005. [22](#), [38](#)
- [17] A. Salmerón, A. Cano, and S. Moral, “Importance sampling in Bayesian networks using probability trees,” *Computational Statistics and Data Analysis*, vol. 34, pp. 387–413, 2000. [22](#), [38](#), [46](#), [49](#), [58](#), [114](#)

- [18] G. Cooper, “The computational complexity of probabilistic inference using Bayesian belief networks,” *Artificial Intelligence*, vol. 42, pp. 393–405, 1990. [22](#), [31](#)
- [19] P. Dagum and M. Luby, “Approximating probabilistic inference in Bayesian belief networks is NP-hard,” *Artificial Intelligence*, vol. 60, pp. 141–153, 1993. [22](#)
- [20] S. Lauritzen and D. Spiegelhalter, “Local computations with probabilities on graphical structures and their application to expert systems,” *Journal of the Royal Statistical Society, Series B*, vol. 50, pp. 157–224, 1988. [23](#), [31](#)
- [21] J. Pearl, “Fusion, propagation and structuring in belief networks,” *Artificial Intelligence*, vol. 29, pp. 241–288, 1986. [31](#)
- [22] J. Pearl, *Probabilistic reasoning in intelligent systems*. Morgan-Kaufmann (San Mateo), 1988. [31](#), [63](#), [183](#), [208](#), [239](#)
- [23] R. Shachter and M. Peot, “Simulation approaches to general probabilistic inference on belief networks,” in *Uncertainty in Artificial Intelligence* (M. Henrion, R. Shachter, L. Kanal, and J. Lemmer, eds.), vol. 5, pp. 221–231, North Holland (Amsterdam), 1990. [31](#), [38](#)
- [24] N. Zhang and D. Poole, “A simple approach to Bayesian network computations,” in *Proceedings of the 10th Canadian Conference on Artificial Intelligence*, pp. 171–178, 1994. [31](#), [58](#), [124](#)
- [25] N. Zhang and D. Poole, “Exploiting causal independence in Bayesian network inference,” *Journal of Artificial Intelligence Research*, vol. 5, pp. 301–328, 1996. [31](#), [58](#), [124](#)
- [26] F. Jensen, K. Olesen, and S. Andersen, “An algebra of Bayesian belief universes for knowledge based systems,” *Networks*, vol. 20, pp. 637–659, 1990. [31](#)
- [27] U. Kjrulff, “Triangulation of graphs – algorithms giving small total state space,” tech. rep., University of Aalborg, Denmark, 1990. [32](#)

- [28] A. Cano, *Propagación aproximada de intervalos de probabilidad en grafos de dependencias*. PhD thesis, University of Granada, 1999. [32](#)
- [29] M. J. Flores and J. A. Gámez, “Triangulation of Bayesian networks by re-triangulation,” *International Journal of Intelligent Systems*, pp. 153 – 154, 2003. [32](#)
- [30] A. L. Madsen and C. J. Butz, “On the importance of elimination heuristics in lazy propagation,” in *Proceedings of the 6th European Workshop on Probabilistic Graphical Models*, pp. 227–234, 2012. [32](#)
- [31] F. Jensen and S. Andersen, “Approximations in Bayesian belief universes for knowledge-based systems,” in *Proceedings of the 6th Conference on Uncertainty in Artificial Intelligence*, pp. 162–169, 1990. [37](#)
- [32] U. Kjærulff, “Reduction of computational complexity in Bayesian networks through removal of weak dependencies,” in *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, pp. 374–382, Morgan Kaufmann, San Francisco, 1994. [37](#)
- [33] R. van Engelen, “Approximating Bayesian belief networks by arc removal,” *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 19(8), pp. 916–920, 1997. [37](#)
- [34] D. Draper, *Localized partial evaluation of Bayesian belief networks*. PhD thesis, Computer Science department, University of Washington, Seattle, 1995. [37](#)
- [35] M. P. Wellman and C. L. Liu, “State-space abstraction for anytime evaluation of probabilistic networks,” in *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence* (R. L. de Mántaras and D. Poole, eds.), pp. 567–574, Morgan & Kaufmann (San Mateo), 1994. [37](#)
- [36] D. Poole, “Exploiting contextual independence and approximation in belief network inference,” tech. rep., University of British Columbia, 1997. [37](#)

- [37] D. Poole, “Context-specific approximation in probabilistic inference,” in *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence* (G. Cooper and S. Moral, eds.), Morgan & Kaufmann, 1998. [37](#)
- [38] A. Cano, S. Moral, and A. Salmerón, “Novel strategies to approximate probability trees in Penniless propagation,” *International Journal of Intelligent Systems*, vol. 18, pp. 193–203, 2003. [37](#)
- [39] M. Henrion, “Search-based methods to bound diagnostic probabilities in very large belief nets.,” in *Proceedings of the 7th Conference on Uncertainty in Artificial Intelligence*, 1991. [38](#)
- [40] E. J. Santos and S. Shimony, “Deterministic approximation of marginal probabilities in bayes nets.,” in *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, vol. 28(4), pp. 377–393, 1998. [38](#)
- [41] D. Poole, “The use of conflicts in searching Bayesian networks,” in *Proceedings of the 9th Conference on Uncertainty in Artificial Intelligence, Washington D.C.*, pp. 359–367, 1993. [38](#)
- [42] D. Poole, “Average-case analysis of a search algorithm for estimating prior and posterior probabilities in Bayesian networks with extreme probabilities,” in *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pp. 606–612, Morgan Kaufmann Publishers, San Mateo, 1993. [38](#)
- [43] D. Poole, “Probabilistic conflicts in a search algorithm for estimating posterior probabilities in Bayesian networks,” *Artificial Intelligence*, vol. 88, pp. 69–100, 1996. [38](#)
- [44] C. Jensen, A. Kong, and U. Kjærulff, “Blocking Gibbs sampling in very large probabilistic expert systems,” *International Journal of Human-Computer Studies*, vol. 42, pp. 647–666, 1995. [38](#), [39](#), [213](#), [240](#)
- [45] J. Pearl, “Evidential reasoning using stochastic simulation of causal models,” *Artificial Intelligence*, vol. 32, pp. 247–257, 1987. [38](#)

- [46] P. Dagum and M. Luby, “An optimal approximation algorithm for Bayesian inference,” *Artificial Intelligence*, vol. 93, pp. 1–27, 1997. [38](#)
- [47] R. Fung and K. Chang, “Weighting and integrating evidence for stochastic simulation in Bayesian networks,” in *Uncertainty in Artificial Intelligence* (M. Henrion, R. Shachter, L. Kanal, and J. Lemmer, eds.), vol. 5, pp. 209–220, North-Holland (Amsterdam), 1990. [38](#)
- [48] M. Henrion, “Propagating uncertainty by logic sampling in Bayes’ networks,” in *Uncertainty in Artificial Intelligence* (J. Lemmer and L. Kanal, eds.), vol. 2, pp. 317–324, North-Holland (Amsterdam), 1988. [38](#)
- [49] L. Hernández, S. Moral, and A. Salmerón, “Importance sampling algorithms for belief networks based on approximate computation,” in *Proceedings of the Sixth International Conference IPMU’96*, vol. II, (Granada (Spain)), pp. 859–864, 1996. [38](#), [39](#)
- [50] L. Hernández, S. Moral, and A. Salmerón, “A Monte Carlo algorithm for probabilistic propagation in belief networks based on importance sampling and stratified simulation techniques,” *International Journal of Approximate Reasoning*, vol. 18, pp. 53–91, 1998. [38](#), [39](#)
- [51] K. Fertig and N. Mann, “An accurate approximation to the sampling distribution of the studentized extreme-valued statistic,” *Technometrics*, vol. 22, pp. 83–90, 1980. [39](#), [143](#)
- [52] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction and Search*. 2nd ed, MIT Press, Cambridge, MA, USA, 2001. [41](#)
- [53] W. Buntine, “Theory refinement on Bayesian networks,” pp. 52–60, Morgan Kaufmann, 1991. [42](#)
- [54] G. Cooper and E. Herskovits, “A Bayesian method for the induction of probabilistic networks from data,” *Machine Learning*, vol. 9, pp. 309–347, 1992. [42](#)

- [55] G. Schwarz, “Estimating the dimension of a model,” *Annals of Statistics*, vol. 6, pp. 461–464, 1978. [43](#)
- [56] I. Good, *The estimation of probabilities*. MIT Press, Cambridge, 1965. [45](#)
- [57] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller, “Context-specific independence in Bayesian networks,” in *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence* (E. Horvitz and F. Jensen, eds.), pp. 115–123, Morgan & Kaufmann, 1996. [46](#)
- [58] A. Cano and S. Moral, “Propagación exacta y aproximada mediante árboles de probabilidad en redes causales (in spanish),” in *Proceedings of the VII Conference of the Spanish Association for Artificial Intelligence (CAEPIA 1997)*, pp. 635–644, 1997. [48](#), [155](#)
- [59] A. Cano, M. Gómez-Olmedo, S. Moral, and C. Pérez-Ariza, “Recursive probability trees for Bayesian networks,” *Lecture Notes in Artificial Intelligence (CAEPIA 2009)*, vol. 5988, pp. 242–251, 2009. [61](#)
- [60] M. Henrion, “Some practical issues in constructing belief networks,” in *Uncertainty in Artificial Intelligence* (L. Kanal, T. Levitt, and J. Lemmer, eds.), vol. 3, pp. 161–173, Elsevier Science Publishers, 1989. [63](#)
- [61] S. Galan and F. Diez, “Modeling dynamic causal interactions with Bayesian networks: temporal noisy gates,” in *Proceedings of the II International Workshop on Causal Networks*, pp. 1–5, 2000. [63](#)
- [62] F. Diez, “Parameter adjustment in bayes networks: the generalized noisy-or gate,” in *Uncertainty in Artificial Intelligence: Procs. of the Ninth Conf.*, pp. 99–105, D. Heckerman and A. Mamdani, 1993. [64](#)
- [63] D. Heckerman, “Causal independence for knowledge acquisition and inference,” in *Uncertainty in Artificial Intelligence: Procs. of the Ninth Conf.*, pp. 122–127, D. Heckerman and A. Mamdani, 1993. [64](#)
- [64] J. Lemmer and D. Gossink, “Recursive noisy-or - a rule for estimating complex probabilistic interactions,” *IEEE SMC*, vol. Part B, 34(6), pp. 2252–2261, 2004. [64](#)

- [65] Y. Xiang and N. Jia, “Modeling causal reinforcement and undermining for efficient cpt elicitation,” *IEEE Trans. Knowledge and Data Engineering*, vol. 19(12), pp. 1708–1718, 2007. [64](#), [226](#)
- [66] K. Kristensen and I. Rasmussen, “A decision support system for mechanical weed control in malting barley,” *Computers and Electronics in Agriculture*, vol. 33, pp. 197–217, 2002. [112](#), [139](#), [213](#), [238](#)
- [67] I. Martínez, S. Moral, C. Rodríguez, and A. Salmerón, “Factorisation of probability trees and its application to inference in Bayesian networks,” in *Proceedings of the First European Workshop on Probabilistic Graphical Models* (J. Gámez and A. Salmerón, eds.), pp. 127–134, 2002. [124](#)
- [68] I. Martínez, S. Moral, C. Rodríguez, and A. Salmerón, “Approximate factorisation of probability trees,” in *ECSQARU’05. Lecture Notes in Artificial Intelligence*, vol. 3571, pp. 51–62, 2005. [124](#), [127](#), [134](#)
- [69] I. Martínez, C. Rodríguez, and A. Salmerón, “Dynamic importance sampling in Bayesian networks using factorisation of probability trees,” in *Proceedings of the Third European Workshop on Probabilistic Graphical Models*, pp. 187–194, 2006. [124](#), [127](#)
- [70] A. Rényi, “On measures of entropy and information,” in *Proceedings of the 4th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 547–561, 1961. [134](#), [138](#)
- [71] S. Kullback and R. Leibler, “On information and sufficiency,” *Annals of Mathematical Statistics*, vol. 22, pp. 76–86, 1951. [135](#)
- [72] K. Olesen, U. Kjærulff, F. Jensen, F. Jensen, B. Falck, S. Andreassen, and S. Andersen, “A munin network for the median nerve - a case study on loops,” *Applied Artificial Intelligence*, vol. 3, pp. 385–404, 1989. [139](#), [239](#)
- [73] C. Conati, A. Gertner, K. V. Lehn, and M. Druzdzel, “On-line student modeling for coached problem solving using Bayesian networks,” in *Proceedings of the Sixth International Conference on User Modeling (UM-96)*, Springer-Verlag, pp. 231–242, 1997. [139](#), [237](#)

- [74] F. V. Jensen, U. Kjærulff, K. G. Olesen, and J. Pedersen, “Et forprojekt til et ekspertsystem for drift af spildevandsrensning (an expert system for control of waste water treatment - - a pilot project) (in danish),” tech. rep., Judex Datasystemer A/S, Aalborg, Denmark, 1989. [139](#), [238](#)
- [75] P. Kampstra, “Beanplot: A boxplot alternative for visual comparison of distributions,” *Journal of Statistical Software*, vol. 28, pp. 1–9, 2008. [139](#)
- [76] A. Salmerón, A. Cano, and S. Moral, “Importance sampling in Bayesian networks using probability trees,” *Computational Statistics and Data Analysis*, vol. 34, pp. 387–413, 2000. [154](#), [175](#)
- [77] J. Quinlan, “Induction of decision trees,” *Machine Learning*, vol. 1, pp. 81–106, 1986. [154](#)
- [78] I. Beinlich, H. J. Suermondt, R. M. Chavez, and G. F. Cooper, “The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks,” in *Proceedings of the Second European Conference on Artificial Intelligence in Medicine*, vol. 38, pp. 247–256, 1989. [213](#), [239](#)
- [79] C. Lacave and F. Díez, “Knowledge acquisition in PROSTANET: A Bayesian network for diagnosing prostate cancer,” *Lecture Notes in Computer Science*, vol. 2774, pp. 1345–1350, 2003. [213](#), [240](#)
- [80] <http://archive.ics.uci.edu/ml/>. [218](#)
- [81] Elvira Consortium, “Elvira: An environment for creating and using probabilistic graphical models,” in *Proceedings of the First European Workshop on Probabilistic Graphical Models* (J. Gámez and A. Salmerón, eds.), pp. 222–230, 2002. [228](#)
- [82] <http://programo.albacete.org>. [228](#)
- [83] <http://www.junit.org/>. [233](#)