- TESIS DOCTORAL -

Carmen: una herramienta de software libre para modelos gráficos probabilistas

Manuel Arias Calleja

Licenciado en Informática
Universidad Politécnica de Madrid

Departamento de Inteligencia Artificial
Universidad Nacional de Educación a Distancia

Septiembre de 2009

- TESIS DOCTORAL -

Departamento de Inteligencia Artificial
Universidad Nacional de Educación a Distancia

Carmen: una herramienta de software libre para modelos gráficos probabilistas

Por: Manuel Arias Calleja

Licenciado en Informática
por la Universidad Politécnica de Madrid

Director de la Tesis: Francisco Javier Díez Vegas

Tribunal calificador

Presidente: Prof. Dr. Enrique Castillo Ron

Vocal 1: Prof. Dr. Pedro Larrañaga Múgica

Vocal 2: Prof. Dr. Andrés Cano Utrera

Vocal 3: Prof. Dr. Hilbert Johan Kappen

Secretaria: Prof[a] Dra. Carmen Lacave Rodero

A mi hermana, Isabel

Resumen

En las últimas dos décadas se ha dado una proliferación de herramientas para la construcción, manual o automática de Modelos Gráficos Probabilistas (MGPs). Las herramientas disponibles están limitadas en su mantenibilidad, robustez y eficiencia. Nuestra contribución principal es una nueva herramienta, llamada Carmen, que se ha desarrollado desde cero y está basada en los principios de la ingeniería del software. Carmen tiene un diseño detallado, una documentación y un conjunto de pruebas sistemáticas para minimizar la presencia de errores.

El desarrollo de esta herramienta ha traido como consecuencia varias contribuciones secundarias: primero, un nuevo patrón de diseño llamado *permiso-ejecución*, que permite realizar operaciones en modelos complejos con múltiples restricciones; segundo, hemos desarrollado un nuevo diseño, que desacopla los diferentes conceptos que constituyen un MGP en partes distintas, permitiendo un mantenimiento posterior más sencillo; tercero, hemos desarrollado una librería genérica de grafos que puede ser utilizada en otras herramientas.

Nuestra segunda contribución principal es un método nuevo que mejora significativamente el rendimiento en las operaciones básicas sobre potenciales de variables discretas, tales como suma, multiplicación, marginalización y división. Hemos demostrado también, tanto teórica como empíricamente, que algunas operaciones compuestas pueden ser realizadas de un modo mucho más eficiente si se ejecutan de forma conjunta en lugar de secuencial. Esta mejora en las operaciones de bajo nivel nos lleva a una reducción en el tiempo y en el espacio necesarios en algoritmos de alto nivel, tales como eliminación de variables, propagación en árboles de cliques, etc.

Finalmente, la tercera contribución principal es un nuevo método para el análisis de coste-efectividad. Los métodos actuales no pueden tratar con problemas que involucran más de una decisión. Por este motivo, hemos desarrollado un nuevo método de coste-efectividad, que puede ser aplicado tanto en árboles de decisión como en diagramas de influencia. Nuestro método es capaz de manejar varias decisiones y devuelve la estrategia óptima como un conjunto de intervalos para λ, un parámetro habitualmente llamado *disponibilidad a pagar*, que representa la cantidad de dinero equivalente a una unidad de efectividad.

Abstract

In the last two decades there has been a proliferation of computer tools for the construction either manual or automatic of Probabilistic Graphical Models (PGMs). The tools currently available are limited in their maintainability, robustness and efficiency. Our main contribution is a new tool, called Carmen, which has been developped from scratch and is based on the principles of software engineering. Carmen has detailed design, a documentation, and a batch of systematic tests aimed at minimizing the presence of errors.

The development of this software tool has led to several secondary contributions: first, a new design pattern called *permission-execution*, which permits to perform operations on complex models with multiple constraints; second, we have developped a new design which decouples the different concepts that make up a PGM in different parts, allowing subsequent maintenance much easier; third, we have developped a general purpose graph library that can be used in other tools.

Our second main contribution is a new method that significantly improves the performance of basic operations on potentials of discrete variables such as addition, multiplication, marginalization and división. We have also proved, theoretically and empirically, that some compound operations can be performed much more efficiently if they are executed all together rather tan sequentially. This improvement in the low-level operations leads to a reduction in the time and space required by high-level algorithms, such as variable elimination, clique tree propagation, etc.

Finally, the third main contribution is a new method for cost-effectiveness analysis. Current methods can not deal with problems that involve more than one decision. For this reason, we have developped a new method of cost-effectiveness, which can be applied to both decision trees and influence diagrams. Our method is capable of handling several decisions and returns the optimal strategy as a set of intervals for λ, a parameter usually called *willingness to pay* wich represents the amount of money equivalent to a unit of effectiveness.

Índice general

Agradecimientos

Son numerosas las personas que, directa o indirectamente, han contribuido a la realización de este trabajo y a las que quiero agradecerles su ayuda. En primer lugar, al profesor Francisco Javier Díez Vegas, director de esta tesis, a quien deseo expresar mi más sincero agradecimiento por haberme ofrecido la posibilidad de desarrollar el proyecto Carmen y todo lo que de él se ha derivado. Es de destacar el mucho tiempo que me ha dedicado así como el apoyo que he recibido a lo largo del desarrollo de la tesis. También deseo agradecerle haber contado conmigo para participar en diversos proyectos en los que he podido conocer a los mejores investigadores de este campo, así como haber puesto a mi disposición todos los medios que he necesitado.

Agradezco también a los miembros del Departamento de Inteligencia Artificial de la UNED su apoyo y consejos en diversos aspectos de esta tesis. Quiero expresar mi gratitud a José Manuel Cuadra por su ayuda con la implementación en C++ de unas funciones, a José Ramón Álvarez por todo lo que me ha enseñado sobre Linux y sobre informática en general y a la profesora Ángeles Manjarrés por haberme prestado su apoyo durante la realización de mi tesis.

Deseo recordar especialmente al profesor José Mira Mira, que dirigió la mayor parte de las tesis doctorales de este departamento, al que todos echamos de menos desde su fallecimiento por su calidad humana y profesional. Entre muchas otras cosas, es de destacar el buen clima de convivencia que se ha vivido en el Departamento de Inteligencia Artificial gracias a su estilo de liderazgo.

Por último, agradezco a mi familia su apoyo durante la realización de la tesis, sobre todo a mi hermana Isabel y a mi padre, junto con mi tía, Francisca Calleja, por sus acertados consejos.

Conclusions

The main objectives of this thesis were two: to create an open source tool, called CARMEN, for editing and evaluating probabilistic graphical models (PGMs) and to develop a method of cost-effectiveness analysis for problems involving several decisions. This thesis has also made several secondary contributions.

Independent contributions of the Carmen tool

We detail first the contributions, that have been used in the construction of CARMEN but are independent of it and, therefore, might be used in other tools.

a) Algorithms for basic operations with potentials

We have designed a method that improves the efficiency of operations on potentials of discrete variables. As mentioned in Section 2.6, the analysis of the computational complexity of inference algorithms by measuring only the number of elementary operations, such as addition or multiplication, is incorrect, because we must also take into account the time necessary to access these values. The method that has been developed substantially improves the speed of basic operations such as marginalization, maximization, multiplication and projection, which leads to significant improvements in the time and space costs of the algorithms for PGMs.

b) Cost-effectiveness analysis

We have developed a method for cost-effectiveness analysis, whose main advantage is the possibility of including multiple decision and chance variables in the model, while previous methods only allowed one decision variable, which had to be at the root of the tree (*TreeAge*, the most widely used program for decision analysis, may yield incorrect results when performing cost-effectiveness analysis in a decision tree containing several decisions, as demonstrated by an example in Section 3.2.2) out method presents the results as a set of intervals of the parameter λ, sometimes called *willingness to pay*, which represents the money-effectiveness equivalence, i.e., the amount of money that the decision maker pay to obtain a unit of effectiveness.

CARMEN implements two versions of this method, one for decision trees and the other for influence diagrams. We expect that this method will be used widely, especially in medicine, where there is great interest in this type of studies.

c) New design pattern

We have developed a design pattern, *Permission-Execution*, for controlling the edition of objects subject to restrictions. The main advantages of this pattern is that it allows each object to have a set of constraints associated that define the operations that can be done on it, and these restrictions can be added or removed in a dynamically way. This pattern is a contribution that can be used in other problems than MGPs.

Free software tool for MGP development

a) CARMEN as a free software tool

CARMEN has been developed following the typical phases in software engineering: analysis, design, codification and testing. The main characteristics of Carmen are:

1. **Robustness**: A careful design and a suite of systematics tests, coded with *jUnit*, guarantee a low density of errors in CARMEN.

2. **Efficiency**: As said before, we have taken special care in the implementation of the most time-consumming operations. As result, CARMEN's performance is comparable with that of the best commercial tools available today.

3. **Maintanability**: This attribute is mainly a consequence of a careful design, with an architecture devised to accommodate future extensions. It is also a consequence of having used a set of coding rules that have ensured a homogeneous style.

4. Maintainability is enhanced an by an extensive **documentation**, both internal and external, that currently consists of:

   - A set of HTML pages generated automatically by the tool *Javadoc* from the comments included in the documentation of the code.
   - A paper (3), presented at the *European Workshop on Probabilistic Graphical Models* (PGM-08), in Hirsthals, Denmark.

- Chapters 4 to 6 of this thesis, which offer an extensive overview of CARMEN, including several types of UML diagrams that graphically show the structure and the dynamics of CARMEN components.

b) CARMEN license

CARMEN is distributed under the LGPL license (see http://www.gnu.org/copyleft/lesser.html), which allows any person, group or enterprise to freely use and modify the software. It also allows to develop new components and plugins, either free or propietary.

c) CARMEN components

CARMEN contains libreries for graphs, for basic operations with potentials of discrete variables, and for creating and editing PGMs. The algorithms implemented so far for bayesian networks are variable elimination, Hugin propagation and lazy propagation; the algorithm implemented for the evaluation of influence diagrams is variable elimination. CARMEN has also the possibility of learning bayesian networks from databases using standard *search and score* algorithms, with several metrics. The graphical user interface of CARMEN is still under development.

Índice de figuras

Índice de tablas

Introducción

Motivación

El objetivo principal de la IA es construir programas de ordenador que, mediante la utilización de modelos, resuelvan problemas de mundo real, como el diagnóstico médico, la predicción financiera, la robótica, el modelado del usuario, etc. En todas estas áreas, hay siempre un componente de incertidumbre, debida a nuestro conocimiento incompleto o al no determinismo del problema que queremos modelar. Los modelos gráficos probabilistas (MGP), que se han venido desarrollando desde los años 80, son una forma de tratar la incertidumbre. Las ventajas de los MGPs son básicamente dos: en primer lugar, su sólido fundamento matemático, y en segundo lugar que, al utilizar razonamiento causal en algunas ocasiones, permiten realizar razonamiento abductivo, deductivo-predictivo e intercausal. Los MGPs entre los que se encuentran las redes bayesianas, los diagramas de influencia y varios tipos de modelos temprales markovianos, al basarse en una factorización de la distribución de probabilidad, han sido capaces de resolver muchos problemas de intrés práctico que eran inabordables con métodos probabilistas anteriores.

En las dos últimas décadas ha habido una gran proliferación de herramientas informáticas para la construcción (manual o automática) de MGPs, unas comerciales y otras de código abierto. Sin embargo, todas ellas presentan limitaciones, como discutiremos en la sección 4.1. Por ello, hemos decidido construir una nueva herramienta denominada CARMEN, que pueda ser utilizada no sólo por investigadores de diferentes países, sino también para la construcción e implantación de sistemas expertos probabilistas. Para cumplir estos objetivos, la herramienta ha de ser eficiente, robusta y fácilmente mantenible. Desde el principio de este trabajo estuvimos convencidos de que solamente se pueden satisfacer estos requisitos si en la construcción de la herramienta

se siguen los principios de la ingeniería del software. A nuestro juicio, ninguna de las herramientas de código abierto existentes en la actualidad cumplen esta condición. Por este motivo, pensamos que no era factible intentar modificar alguno de los programas actuales, sino que era necesario diseñar y programar desde cero una nueva herramienta.

El objetivo de que nuestra herramienta sea muy eficiente nos ha llevado a desarrollar nuevos algoritmos para las operaciones básicas de los MGPs: no basta que los algoritmos de alto nivel sean eficientes, también es necesario que las operaciones básicas, como la multiplicación y la marginalización de potenciales, sean lo más rápidas posible (véase el capítulo 2).

El objetivo de que la herramienta sea mantenible requiere un diseño cuidadoso, tanto en la arquitectura, (véase la sección 4.3), como en cada uno de sus componentes (capítulos 5 y 6). Este diseño está preparado para permitir la extensión del programa y aísla en componentes concretos aquellas partes que se prevea que puedan cambiar.

Por otro lado, las investigaciones llevadas a cabo por nuestro grupo, que está especializado en inteligencia artificial aplicada a la medicina, pusieron de manifiesto que los métodos de análisis de coste-efectividad acuales no permiten realizar este tipo de estudios en problemas que involucren más de una decisión. De hecho, el método que ofrece la herramienta comercial más utilizada en la actualidad da resultados incorrectos para este tipo de problemas (véase la sección 3.2.2). Ello nos llevó a desarrollar un nuevo método de análisis de coste-efectividad, que puede ser aplicado tanto con árboles de decisión como con diagramas de influencia.

Objetivos

El planteamiento expuesto en la sección anterior se concreta en los objetivos siguientes:

1. Realizar el análisis, diseño e implementación de una herramienta de software libre para editar y realizar inferencia sobre MGPs, tenga estas características:

   a) Mantenibilidad y, como consecuencia, que pueda ser ampliada y utilizada por otros grupos de investigación, empresas o particulares.

   b) Eficiencia, lo que significa que es necesario implementar los mejores algoritmos, tanto de alto como de bajo nivel o la creación de nuevos algoritmos sobre temas puntuales, aunque sin sacrificar la mantenibilidad.

c) Correción, para lo cual es necesario realizar pruebas sistemáticas.

2. Desarrollar un algoritmo que pueda resolver problemas de coste-efectividad con más de una decisión; el resultado debe darse en función de un conjunto de intervalos para el valor del parámetro λ, que representa el coste económico que el usuario está dispuesto a realizar para conseguir una unidad de efectividad.

Metodología

El desarrollo de nuevos algoritmos ha seguido la metodología usual en investigación: estudio de los fundamentos matemáticos y del estado del arte, diseño, implementación y comparación, si procede, con otros algoritmos similares. Detallamos las fases que hemos seguido en CARMEN y en sus componentes.

En el caso del algoritmo para operaciones básicas hemos realizado también una serie de experimentos para medir las mejoras en tiempo y en memoria.

Las pruebas se han realizado utilizando la herramienta jUnit[1]. En general, la implementación de los algoritmos ya existentes en la literatura (inferencia, aprendizaje, etc.) ha seguido las fases usuales en ingeniería del software: análisis, diseño, codificación y pruebas como si se tratara de mini-aplicaciones autónomas antes de ser integradas en CARMEN.

En el diseño de CARMEN se han utilizado varios de los patrones de diseño propuestos en la literatura (25; 29; 67), junto con un nuevo patrón que ha surgido de la decisión de controlar la creación y modificación de MGPs mediante un conjunto de *ediciones* (por ejemplo, añadir un enlace, cambiar el nombre de una vairable, etc. y un conjunto de *restricciones* (por ejemplo, que la red no tenga ciclos o que no haya dos variables con el mismo nombre).

El proceso de diseño en CARMEN ha sido iterativo: las necesiades detectadas durante la implementación han llevado a modificar en varias ocasiones el diseño inicial, a veces incluso en algunos de los aspectos funcamentales, dando lugar a un proceso de realimentación continua. Por ejemplo, el uso de restricciones surgió de la necesidad de contar con mecanismos flexibles para la edición de las redes (las restricciones pueden añadirse dinámicamente a una red o eliminarse) y esto obligó a revisar todos los aspectos del diseño, desde los tipos de MGPs hasta los algoritmos que operan sobre ellos. Algunos

[1] Es una herramienta para implementar pruebas de unidad en Java: http://www.junit.org

de estos cambios han requerido un gran esfuerzo de programación, pero consideramos que ha merecido la pena, pues el resultado ha sido una herramienta más robusta y más flexible. En este sentido ha sido muy útil el concepto de *refactorización*: refactorizar consiste en hacer un cambio que afecta a varias partes distintas del código ya existente. La refactorización puede ser tan sencilla como cambiar el nombre de una variable o un método, o tan compleja como recodificar una acción mediante un patrón de diseño (37). La utilización de la herramienta de software libre *Eclipse*[2], que incluye explícitamente facilidades de refactorización, ha facilitado notablemente esta tarea.

Como hemos dicho ya, uno de los objetivos de este proyecto, desde su inicio, es que distintos programadores pudieran colaborar en él. Esto ha ocurrido ya durante el desarrollo de la presente tesis doctoral:

- Carlos Baena Parrado, alumno de la UNED en el Programa Interuniversitario de Doctorado *Modelos Gráficos Probabilísticos para la Inteligencia Artificial y Minería de Datos*, ha implementado el algoritmo de propagación perezosa para redes bayesianas (52).

- Jesús Oliva Gonzalo, alumno del *Máster en Inteligencia Artificial Avanzada* de la UNED, ha implementado los algoritmos estándar para el aprendizaje de redes bayesianas a partir de bases de datos, con la ayuda económica del Programa de Financiación de Acciones Específicas de CIBERESP (Centro de Investigación Biomédica en Red de Epidemiología y Salud Pública).

- Enrique Mendoza Miranda y Juan Luis Gozalo, en sus respectivos proyectos de fin de carrera de Ingeniería Informática de la UNED, han contruibuido al desarrollo de la interfaz gráfica de usuario. También ha contribuido en esta tarea Alberto Ruiz Lafuente, financiado con los fondos aportados por la Agencia Laín Entralgo a través del proyecto *Sistema de ayuda a la decisión para cirugía de cataratas*.

- En la actualidad, Jorge Fernández Suárez, alumno de la UNED en el Programa Interuniversitario de Doctorado *Modelos Gráficos Probabilísticos para la Inteligencia Artificial y Minería de Datos*, está implementando los modelos de decisión de Markov (en inglés, *Markov decision processes*, MDPs), y María Nicola García, dentro de su proyecto de fin de carrera de Ingeniería Informática de la UNED está implementando el algoritmo de evaluación de diagramas de influencia

[2]Entorno de desarrollo para múltiples lenguajes de programación, véase www.eclipse.org

con nodos supervalor (47; 49). Estos dos últimos trabajos no se describen en esta memoria porque aún no están concluidos.

- Carla Margalef Bentabol, becaria del Dpto. de Inteligencia Artificial de la UNED, al utilizar la herramienta CARMEN en la construcción de un sistema de ayuda a la decisión para cirugía de cataratas que actualmente está en fase de pruebas en el Hospital de Fuenlabrada, ha detectado algunos errores (principalmente relacionados con la compatibilidad de librerías) y ha aportado algunos métodos auxiliares para ciertas funciones.

Todas estas personas han trabajo bajo la supervisión cercana del autor y del director de esta tesis doctoral.

Como herramienta de control de versiones se ha utilizado *Subversion*, un programa de software libre, que introduce varias mejoras importantes frente a *CVS*. Para la programación de la interfaz gráfica Enrique Mendoza han utilizado *Netbeans*, que es software libre, mientras que Juan Luis Gozalo ha utilizado *SwingDesigner*, un producto comercial de la empresa Instantiations. En la normativa para programadores de CARMEN, el criterio que se da es que cada uno puede utilizar la herramienta que desee (*Eclipse, Netbeans*, etc), dentro de las que generan código "limpio", es decir, compatible con las demás herramientas.[3]

Organización de la tesis

La tesis está dividida en cuatro partes. En la primera, revisamos el estado del arte, en la segunda e, exponemos dos nuevos algoritmos: uno para operaciones con potenciales de variables discretas (capítulo 2) y otro para el análisis de coste-efectividad (capítulo 3).

La herramienta CARMEN se describe en la tercera parte, a lo largo de varios capítulos:

- El capítulo 4 da una visión general de la herramienta: comparación con otras similares, análisis de requisitos y arquitectura.

[3]Puede obtenerse más información sobre cada una de estas herramientas en las siguientes páginas web:
—Subversion: http://subversion.tigris.org,
—CVS: http://www.nongnu.org/cvs,
—SwingDesigner: http://www.instantiations.com/windowbuilder/swingdesigner,
—Eclipse: http://www.eclipse.org,
—Netbeans: http://www.netbeans.org.

- El capítulo 5 trata sobre las estructuras de datos principales. Se explica con detalle por qué su evolución fue larga y compleja, tal como hemos comentado en el apartado de metodología.

- El capítulo 6 explica los algoritmos de inferencia y las heurísticas que se han implementado.

El estilo que hemos seguido para describir la herramienta, a lo largo de dichos capítulos, ha sido de lo más general a lo más concreto. En general, no hemos considerado oportuno ahondar demasiado en los detalles o la problemática de la implementación, excepto en aquellas partes que puedan ser novedosas.

Las conclusiones se exponen en la parte cuarta (capítulo 7).

El diseño, tanto el arquitectónico como el detallado, se ha descrito con el lenguaje UML. Por este motivo, en el apéndice A ofrecemos una breve descripción de este lenguaje.

Parte I

Preliminares

Capítulo 1

Estado de la técnica

Este capítulo está dividido en dos partes: nociones relativas a los modelos gráficos probabilistas (MGPs) 1.1 y una introducción a la ingeniería del software 1.2. Por lo que respecta a los MGPs, veremos un resumen de la teoría de grafos 1.1.1, una introducción a la probabilidad 1.1.2 y, conceptos básicos sobre modelos gráficos probabilistas 1.1.3. Por lo que respecta a la ingeniería del software, describiremos los paradigmas de desarrollo 1.2.1 y las fases del ciclo de vida de en las que se divide el desarrollo de un proyecto 1.2.2.

1.1. Introducción a los MGPs

1.1.1. Grafos

Definiciones

Un *grafo* es un par $G = (N, E)$ donde N es un conjunto generalmente no vacío de elementos, llamados *vértices* o *nodos*, y E es un conjunto de *enlaces* o *aristas* que unen unos nodos con otros. El conjunto de enlaces E se define como una tripleta[1] $E \subseteq N \times N \times D$, donde D es una variable booleana que indica si el enlace es dirigido (d) o no dirigido (n). La notación que usaremos para los enlaces es: ($nodo_A$, $nodo_B$, *dirigido*) o de forma más abreviada: $A \rightarrow B$ para (A, B, d) donde A y B son nodos, y $A - B$ para (A, B, n). Se dice que un grafo es *dirigido* si todos sus enlaces son dirigidos, *no dirigido* si todos sus enlaces son no dirigidos y *mixto* si tiene enlaces dirigidos y no dirigidos.

[1] Tradicionalmente, los grafos se han venido definiendo con la notación $E \subseteq N \times N$. En nuestro caso hemos querido indicar explícitamente si el enlace es dirigido o no mediante una etiqueta booleana.

Figura 1.1: Grafo mixto. El conjunto de nodos es $N = \{A,B,C\}$ y el de enlaces $E = \{(B, A, d), (B, C, n)\}$.

En un enlace $X \rightarrow Y$, se dice que X es *padre* de Y y que Y es *hijo* de X; en un enlace $X - Y$, se dice que X e Y son *hermanos*. Los *vecinos* de un nodo son sus padres, sus hijos y sus hermanos. Dos enlaces son adyacentes si tienen un nodo en común.

Definición 1.1 (Camino) *En $G = (N,E)$ entre un nodo X inicial y un nodo Y final es una sucesión de enlaces adyacentes $\{e_1, e_2, \ldots, e_n\}$ conectados entre sí.*

El camino es *dirigido* si todos sus enlaces son dirigidos y se puede ir desde Z_1 hasta Z_n siguiendo la dirección de los enlaces. Un camino $(e_1, e_2 \ldots, e_n)$ donde $e_i = (X_i, Y_i)$ tal que $Y_i = X_{i+1}$ se dice que es *abierto* si $X_1 \neq Y_n$ y se dice que es cerrado si $X_1 = Y_n$. La *longitud* del camino se define como el número de enlaces que contiene. En el caso de grafos dirigidos distinguiremos entre *ciclo*, que es un camino cerrado que sigue la dirección de los enlaces, y *bucle*, que es un camino cerrado en el que no tenemos en cuenta su dirección. Se dice que un grafo es *acíclico* si no contiene ningún ciclo. En grafos no dirigidos sólo hablaremos de la existencia de ciclos.

En un grafo dirigido G, los *ascendientes* o *antepasados* de un nodo Y son todos aquellos nodos $\{X_1, X_2, \ldots, X_n\}$ para los que existe un camino dirigido desde X_i hasta Y. Los *descendientes* de y son aquellos nodos $\{Z_1, Z_2, \ldots, Z_n\}$ para los que exista un camino desde Y hasta Z_i. Se dice que un grafo es *conexo* si para cualquier par de nodos existe un camino entre ellos.

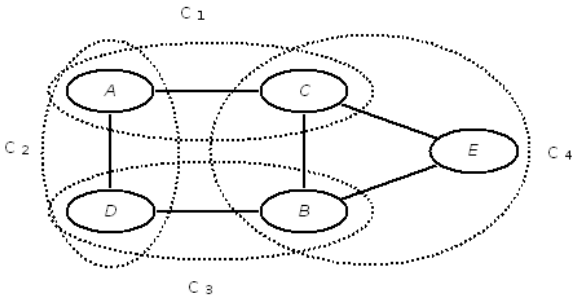
Definición 1.2 (Árbol) *Es un grafo tal que, dados dos nodos cualesquiera, sólo existe un camino entre ellos (sin tener en cuenta la dirección de los enlaces).*

Un árbol puede ser dirigido o no, pero lo más usual es hablar de árboles dirigidos y en adelante sólo trataremos este tipo.

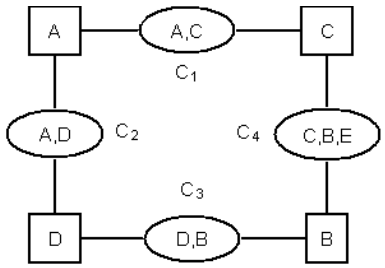
Definición 1.3 (Poliárbol) *Es un tipo de grafo en el que cada nodo puede tener varios padres pero no contiene bucles.*

En grafos no dirigidos, un grafo es *completo* si todos los nodos son hermanos dos a dos. Un subconjunto de nodos es completo si todos los nodos son adyacentes dos a dos. Si un conjunto completo es maximal, es decir, no está contenido en otro conjunto completo,

se denomina *conglomerado* (13). Dados dos conglomerados con algún nodo en común, su *separador* es el conjunto de nodos que dichos conglomerados tienen en común.



(a) Grafo no completo que contiene cuatro conglomerados: $\mathbf{C_1} : \{A,C\}$, $\mathbf{C_2} : \{A,D\}$, $\mathbf{C_3} : \{B,D\}$, $\mathbf{C_4} : \{C,B,E\}$.



(b) Grafo correspondiente de conglomerados y separadores. Los conglomerados se representan mediante elipses y los separadores en rectángulos.

Grafos triangulados

En las definiciónes de *Árbol de unión* y de *Árbol de conglomerados* suponemos que existen dos grafos: el original (G) y el árbol de grupos asociado a G.

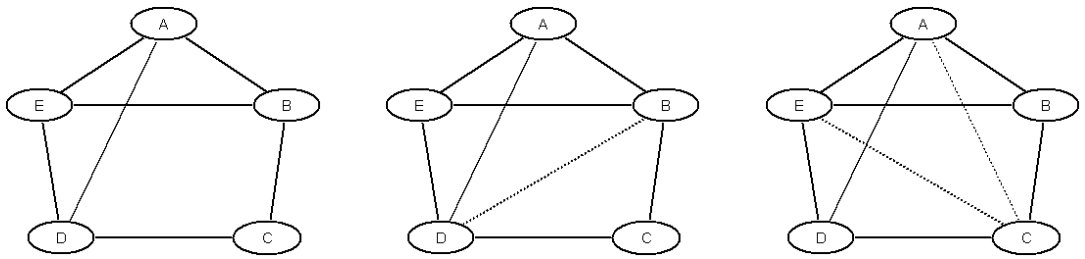
Definición 1.4 (Árbol de unión (*Join tree*)) *Es un tipo de árbol en el que cada nodo representa un conjunto de nodos de G y se cumple la siguiente condición: si un nodo de G está contenido en dos nodos del árbol, entonces también está contenido en todos los nodos del camino que conecta ambos conglomerados en el árbol.*

Definimos ahora los grafos triangulados, que como veremos más adelante, presentan un gran interés para los algoritmos de inferencia sobre MGPs.

Definición 1.5 (Cuerda) *Es un enlace que une dos nodos de un ciclo y que no pertenece al ciclo.*

Definición 1.6 (Grafo triangulado) *Un grafo no dirigido se denomina triangulado si cada bucle de longitud mayor que tres contiene al menos una cuerda.*

Triangular un grafo consiste en añadir los enlaces que sean necesarios entre los nodos del grafo para que no existan ciclos de longitud mayor que tres. Cuando se triangula un grafo generalmente los conglomerados cambian. Dado un grafo no triangulado existen varias formas distintas de triangularlo.



(a) Grafo no triangulado. (b) Posible triangulación. (c) Otra triangulación.

Figura 1.2: Diferentes posibilidades en la triangulación de un grafo.

Definición 1.7 (Árbol de conglomerados (*Clique tree*)) *Es un tipo de árbol de unión cuyos nodos y enlaces están asociados a conjuntos de nodos de un grafo no dirigido G. Cada nodo del árbol es un conglomerado de G.*

Cuando un grafo G está triangulado, sus conglomerados pueden organizarse en forma de árbol, tal que cada nodo del árbol representa un conglomerado de G y se cumple la propiedad del árbol de unión.

1.1.2. Probabilidad

Definiciones básicas sobre probabilidad

Definición 1.8 (Experimento aleatorio) *Es un proceso cuyos resultados no se conocen de antemano.*

Definición 1.9 (Suceso) *Es cada uno de los posibles resultados de un experimento aleatorio.*

Definición 1.10 (Espacio muestral) *Es el conjunto de todos los posibles resultados de un experimento aleatorio. Se suele representar por* Ω.

Definición 1.11 (Variable aleatoria) *Es aquélla que toma valores que, a priori, no conocemos con certeza.*

Interpretaciones de la probabilidad *Frecuentista*: dados n experimentos aleatorios, la probabilidad es el límite cuando $n \to \infty$ de la frecuencia relativa de aparición de un suceso A. *Bayesiana*: Se expresa como $P(A)$ y es un número del intervalo $[0,1]$ que expresa el grado de certeza subjetiva.

Definición 1.12 (Distribución de probabilidad de una variable discreta finita X) *Es un conjunto de asignaciones de probabilidad a cada uno de sus posibles estados* $\{x_1,\ldots,x_n\}$ *tales que* $0 \leq P(x_i) \leq 1$ *y* $\sum_i P(x_i) = 1$.

Definición 1.13 (Probabilidad condicionada) *Es la frecuencia relativa de aparición de un suceso A cuando ha ocurrido un suceso B. Se expresa como* $P(A|B)$.

Definición 1.14 (Independencia de sucesos) *Dos sucesos A y B son independientes si* $P(A|B) = P(A)$ *y* $P(B|A) = P(B)$.

Definición 1.15 (Configuración) *Dado un conjunto de variables discretas* $\mathbf{X} = \{X_1,X_2,\ldots,X_n\}$*, una configuración es una asignación de valores a cada una de las variables del conjunto* $\mathbf{X}$.

Definición 1.16 (Probabilidad conjunta) *Dado un conjunto de variables* $\mathbf{X} = \{X_1,X_2,\ldots,X_n\}$*, la probabilidad conjunta* $P(X_1,X_2,\ldots,X_n)$ *es una asignación de valores al espacio formado por todas las posibles configuraciones de* $\mathbf{X}$.

Definición 1.17 (Probabilidad marginal) *Dada una distribución de probabilidad conjunta de un conjunto de variables* $\mathbf{X} = \{X_1,X_2,\ldots,X_n\}$*, la probabilidad marginal de una variable* X_i *es la que se obtiene a partir de la probabilidad conjunta sumando las frecuencias de las configuraciones de las demás variables.*

Teorema 1.18 (Teorema de Bayes) *Sea* $\{A_1,A_2,\ldots,A_n\}$ *un conjunto de sucesos excluyentes tales que* $A_1 \cup A_2 \cup \ldots \cup A_n = \Omega$*. Sea B un suceso del que se conocen las probabilidades condicionales* $P(B|A_i)$*. La probabilidad de* $P(A_i|B)$ *se puede calcular por la expresión:* $P(A_i|B) = \frac{P(B|A_i)P(A_i)}{P(B)}$.

Las **variables** pueden ser de varios tipos. Una variable discreta es la que toma un conjunto reducido de valores, que se llaman estados de la variable. Por ejemplo: *Hepatitis C* = {presente, ausente}, *Días semana* = {lunes, martes, ... , domingo}. El conjunto de valores que puede tomar la variable X se llama *dominio*, y lo escribimos como $dom(X)$. Una *variable discreta infinita*, representa números naturales o variables discretas con límites a veces no muy precisos, como los nombres de las personas. Una variable continua es aquella que puede tomar cualquier valor del conjunto de los números reales. Una variable continua se puede discretizar dividiendo su dominio en intervalos; se puede tratar por tanto como si fuera discreta.

Lo más habitual es trabajar con variables discretas finitas. A partir de ahora supondremos que las variables con las que tratamos son de este tipo. Las variables se representan mediante letras mayúsculas, con o sin subíndice, y los valores que puede tomar cada variable mediante letras minúsculas. Por ejemplo, podemos representar la edad por X y los valores que puede tomar por $\{x_1, x_2, \ldots, x_n\}$.

Definición 1.19 (Evidencia) *Es la asignación de un valor a cada variable de un conjunto.*

Definición 1.20 (Probabilidad a priori de una variable) *Es la distribución de probabilidad marginal de una variable cuando ninguna de las variables del modelo tiene asignada evidencia.*

1.1.3. MGPs

Definición 1.21 (Modelo gráfico probabilista) *Es un grafo cuyos nodos representan, hablando grosso modo, variables y cuyos enlaces representan las relaciones de dependencia condicional entre las variables.*

El modelo tiene dos componentes: por un lado cualitativa, que se expresa por la existencia o no de relaciones de dependencia condicional entre las variables o de precedencia temporal si es en el caso de diagramas de influencia. Por otro lado tiene una vertiente cuantitativa dada por los parámetros del modelo.

a) MGPs púramente probabilistas

En estos modelos se manejan distribuciones de probabilidad conjunta de múltiples variables en los que existe una factorización de la distribución de probabilidad. Las variables que forman parte de cada distribución conjunta están determinadas por las relaciones que existen entre las variables en el grafo.

Definición 1.22 (d-Separación) *Se dice que dos variables A y B en una red causal están d-separadas si para todos los caminos entre A y B existe una variable intermedia V tal que ocurre una de estas dos condiciones:*

- *La conexión es serie o divergente y V está instanciada.*
- *La conexión es convergente y ni V ni ninguno de sus descendientes han recibido evidencia.*

Definición 1.23 (Propiedad de Markov) *Una terna* $(\mathbf{X}, \mathcal{G}, P)$ *formada por un conjunto de variables aleatorias* $\mathbf{X}$*, un GDA* $\mathcal{G} = (\mathbf{X}, \mathcal{A})$ *en que cada nodo representa una variable* X_i*, y una distribución de probabilidad sobre* $\mathbf{X}$*,* $P(\mathbf{X})$*, cumple la* propiedad de Markov *si y sólo si para todo nodo* X*, el conjunto de sus padres,* $Pa(X)$*, separa condicionalmente este nodo de todo subconjunto* $\mathbf{Y}$ *en que no haya descendientes de* X*. Es decir,*

$$P(X|pa(X), \mathbf{y}) = P(x|pa(X)) \tag{1.1}$$

Partiendo de los párrafos anteriores podemos definir una red bayesiana de este modo:

Definición 1.24 (Red bayesiana) *Una red bayesiana consta de tres elementos: un conjunto de variables aleatorias,* $\boldsymbol{X}$*; un grafo* $\mathcal{G} = (\boldsymbol{X}, \mathcal{A})$ *un grafo dirigido acíclico (GDA) en que cada nodo representa una variable* X_i*; y una distribución de probabilidad sobre* $\mathbf{X}, P(\mathbf{X})$*, que puede ser factorizada así:*

$$P(\mathbf{x}) = \prod_i P(x_i|pa(X_i)) \tag{1.2}$$

En la ecuación 1.2 se expresa el aspecto gráfico de la red con las probabilidades condicionadas de cada nodo donde se incluyen sus padres.

Inferencia en redes bayesianas

El objetivo de la inferencia en redes bayesianas es calcular la distribución de probabilidad *a posteriori* de un conjunto Q de variables de interés dada una evidencia e sobre un modelo probabilista $\mathcal{G} = (V, E)$. Dicho de otro modo se pretende calcular $P(Q|e)$.

En Carmen todos los algoritmos que se han implementado hasta ahora son exactos.

b) Modelos de decisión probabilistas

Un modelo de decisión representa problemas con múltiples decisiones y variables aleatorias discretas. Las decisiones se representan mediante rectángulos; las variables aleatorias se representan mediante óvalos, los nodos que representan variables aleatorias se llaman nodos de azar; las utilidades son representadas habitualmente por triángulos o a veces por rombos.

Figura 1.3: Esquema resumen de los principales métodos de propagación para redes bayesianas.

Definición 1.25 (Árboles de decisión) *Un árbol de decisión está formado por un grafo en forma de árbol y consta de tres tipos de nodos: azar, decisión y utilidad. Las utilidades están sólo en las hojas y todos los nodos hoja son utilidades.*

El árbol se lee de izquierda a derecha empezando por el nodo raíz. Los enlaces tienen etiquetas que indican la decisión tomada o el valor de la variable aleatoria que hay a la izquierda, en este segundo caso, la etiqueta tiene asociada una probabilidad $p(s|t)$ (s: estado, t: pasado).

La evaluación de un árbol se hace desde las hojas a la raiz y da como resultado una política de actuación.

En la construcción del árbol de decisión el nodo raíz es la primera variable aleatoria conocida o la primera decisión. Si no se conoce ninguna variable, se van añadiendo nodos en cada rama según se tomen decisiones o se conozcan variables en el camino que se siga. La representación gráfica empieza con el nodo raíz a la izquierda. Los hijos de cada nodo se sitúan a la derecha de su padre. Los nodos de utilidad aparecen en el extremo derecho.

La evaluación de un árbol de decisión se realiza de derecha a izquierda. La utilidad de un nodo aleatorio X es el promedio de las utilidades de sus hijos de X ponderada por su

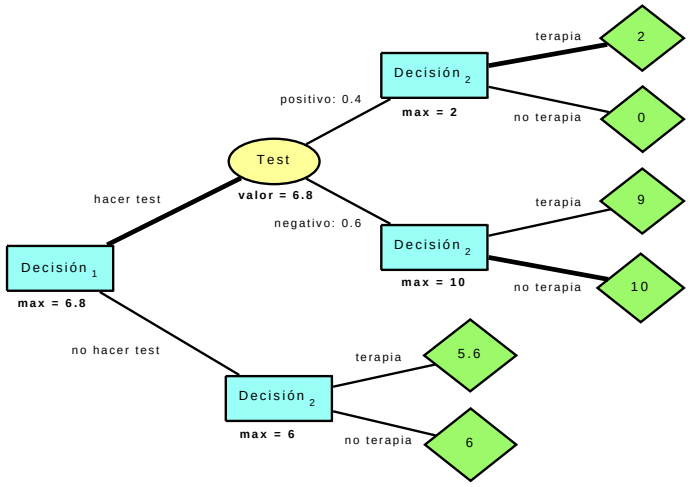


Figura 1.4:]
Árbol de decisión correspondiente a un paciente del que se sospecha que padece una determinada enfermedad. La primera decisión es si se hace un test y la segunda si se aplica una terapia.

probabilidad: $U(X) = \sum_X U(x) \times P(x)$ y la de un nodo de decisión es el máximo de las utilidades de sus hijos: $U(X) = max_X U(x) \times P(x)$.

El inconveniente de los árboles de decisión es su crecimiento exponencial con el número de variables aleatorias y decisiones.

c) Aprendizaje de MGPs

Aprendizaje probabilista Definimos aprendizaje probabilista como aquél que se basa en los principios de la teoría de la probabilidad, aunque el modelo resultante no sea probabilista, por ejemplo, se puede realizar un aprendizaje probabilista de redes neuronales aunque éstas no sean en sí un modelo probabilista.

Aprendizaje de máxima verosimilitud La verosimilitud es una función, habitualmente representada por λ, que indica en qué medida cada hipótesis o cada modelo explica los datos observados:

$$\lambda(\text{Modelo}) = P(\text{datos}|\text{modelo}) \tag{1.3}$$

El aprendizaje de máxima verosimilitud consiste en seleccionar/encontrar o el modelo que maximiza esta función.

Aprendizaje bayesiano Consiste en asignar una probabilidad a priori a cada uno de los modelos, $P(\text{modelo})$; la probabilidad a posteriori del modelo dados los datos se define usando el teorema de Bayes:

$$P(\text{modelo}|\text{datos}) = \frac{P(\text{modelo}) \cdot P(\text{datos}|\text{modelo})}{P(\text{datos})} \tag{1.4}$$

Se puede observar la semejanza con los problemas de diagnóstico probabilista. En ellos se trata de diagnosticar la enfermedad que mejor explica los síntomas. Aquí tratamos de "diagnosticar" el modelo que mejor explica los datos observados. En ambos casos, el "diagnóstico" se basa en una probabilidad a priori, que representa el conocimiento que teníamos antes de realizar las observaciones, y en una verosimilitud que indica en qué medida cada una de nuestras hipótesis (en este caso, cada modelo) explica los datos observados.

En la ecuación anterior el denominador es una constante, en el sentido de que es la misma para todos los modelos. Por tanto, si no queremos conocer la probabilidad absoluta, sino sólo cuál de los modelos tiene mayor probabilidad que los demás, podemos quedarnos con una versión simplificada de ella:

$$P(\text{modelo}|\text{datos}) \propto P(\text{modelo}) \cdot P(\text{datos}|\text{modelo}) \tag{1.5}$$

El aprendizaje de máxima verosimilitud es un caso particular del aprendizaje bayesiano porque cuando la probabilidad a priori es constante, la probabilidad a posteriori es proporcional a la verosimilitud,

$$P(\text{modelo}) = \text{constante} \Rightarrow P(\text{modelo}|\text{datos}) \propto P(\text{datos}|\text{modelo}) \tag{1.6}$$

y maximizar una de ellas es lo mismo que maximizar la otra.

d) Aprendizaje de redes bayesianas

Aprendizaje paramétrico El aprendizaje paramétrico supone que se conoce el grafo de la red bayesiana y, en consecuencia, la forma de la factorización de la probabilidad. En

el caso de variables discretas, podemos considerar que cada probabilidad condicionada $P(x_i|pa(X_i))$ es un parámetro, que llamaremos $\theta_{P(x_i|pa(X_i))}$. Para cada configuración $pa(X_i)$ se cumple que

$$\sum_{x_i} P(x_i|pa(X_i)) = 1 \tag{1.7}$$

y por tanto, si X_i toma n_{X_i} valores, el número de parámetros independientes para cada configuración $pa(X_i)$ es $n_{X_i} - 1$.

La estimación de cada parámetro se puede hacer por el método de la máxima verosimilitud. Los problemas son que no existan casos en la base de datos de una determinada configuración y el sobreajuste. Estos dos problemas se resuelven mediante la estimación bayesiana, lo cual impleca dar una distribución de probabilidad para los parámetros $P(\boldsymbol{\Theta})$. La mayor parte de los métodos propuestos en la literatura se basan en la hipótesis de *independencia de los parámetros*, que se expresa así:

$$P(\theta) = \prod_i \prod_j P(\theta_{ij}) \tag{1.8}$$

En la literatura, lo más frecuente es suponer que la distribución de probabilidad a priori para los parámetros de una configuración, $P(\theta_{ij})$ se trata de una distribución de Dirichlet. Para resolver los problemas anteriores se suele también usar la *corrección de Laplace*, que añade a la base de datos un caso ficticio por cada configuración. Existen variantes que en lugar de añadir un caso añaden varios.

Aprendizaje estructural a partir de relaciones de independencia El problema consiste en encontrar un grafo dirigido acíclico (GDA) que sea un mapa de independencias (I-mapa) de P, preferiblemente un I-mapa minimal, por cuatro motivos: porque muestra más relaciones de independencia que el primero, porque va a necesitar menos espacio de almacenamiento, porque al necesitar menos parámetros podrá estimarlos con mayor fiabilidad, reduciendo además el riesgo de sobreajuste y porque conducirá a una computación más eficiente.

Para obtener las relaciones de dependencia e independencia se suelen aplicar test de independencia como χ^2 para eliminar posibles relaciones espurias. Un problema es que el número de relaciones posibles crece super-exponencialmente con el número de

variables, pues hay que examinar cada relación del tipo $I_P(\mathbf{X}, \mathbf{Y}|\mathbf{Z})$, con la única condición de que los tres subconjuntos sean disjuntos y $\mathbf{X}$ e $\mathbf{Y}$ sean no vacíos. Por este motivo, el aprendizaje basado en relaciones sólo puede utilizarse para problemas en que el número de variables es muy reducido.

Construcción del grafo Una vez obtenida la lista de relaciones, hay que construir el grafo. El algoritmo más conocido es el denominado PC, de Spirtes et al. (71; 72). Como puede verse en el algoritmo 1, el algoritmo consta de dos fases. En la primera, toma como punto de partida un grafo completo no dirigido y va eliminando enlaces basándose en las relaciones de independencia. La segunda fase consiste en asignar una orientación a los enlaces del grafo no dirigido obtenido en la primera fase. Los detalles de este algoritmo, así como su justificación teórica, pueden encontrarse en las referencias citadas y en el libro de Neapolitan (57, cap. 10), que estudia además otros algoritmos de aprendizaje estructural similares.

Aprendizaje estructural mediante búsqueda heurística Es un método alternativo de aprendizaje estructural que consiste en utilizar una métrica para determinar cuál es el mejor modelo. El problema se descompone en dos partes: buscar el mejor grafo y buscar los mejores parámetros para cada grafo posible (con las técnicas usuales). El problema se reduce a examinar todos los grafos posibles, lo cual no es factible para problemas de tamaño mediano porque el número de grafos posibles crece de forma super-exponencial con el número de variables. La solución es aplicar una *búsqueda heurística*, que consiste en generar unas pocas posibles soluciones, seleccionar la mejor (o las mejores), y a partir de ella(s) generar otras nuevas, hasta encontrar una que satisfaga ciertos criterios. Este proceso de *búsqueda en calidad* necesita ser guiado por una *métrica* (en inglés, *score*) que indique la calidad de cada posible solución.

Por tanto, cada algoritmo de aprendizaje de este tipo se caracteriza por dos elementos: una medida de calidad y un algoritmo de búsqueda.

Definición 1.26 (Métrica de calidad) *Es un criterio mediante el cual se puede ordenar un conjunto de redes bayesianas según su calidad.*

En la literatura existente se han propuesto varias medidas de calidad para redes bayesianas, que pueden clasificarse en tres tipos: 1) bayesianas (K2 (16), Geiger y Heckerman en 1995 (30)), 2) de longitud mínima de descripción (LMD (en inglés

Algoritmo 1: Algoritmo PC.

Entrada: conjunto de nodos V y conjunto de relaciones de independencia IND
Resultado: estructura de la red aprendida

```
1  //Primera etapa:
2  Crear el grafo no dirigido completo G
3  i = 0
4  Mientras |ADJ_X| > i ∀X ∈ V hacer
5      Para cada X ∈ V hacer
6          Para cada Y ∈ ADJ_X hacer
7              Buscar un subconjunto S ⊆ ADJ_X − Y tal que |S| = i y I(X,Y|S) ∈ IND
8              Si existe ese subconjunto entonces
9                  S_XY = X eliminar la arista X - Y de G
10     i = i + 1
11 //Segunda etapa:
12 Para cada enlace del tipo X - Z - Y hacer
13     Si Z ∉ S_XY entonces
14         orientar X - Z - Y como X → Z ← Y
15 Mientras queden enlaces sin orientar hacer
16     Para cada enlace del tipo X → Z − Y hacer
17         orientar Z - Y como Z ← Y
18     Para cada enlace del tipo X - Y tal que hay un camino de X a Y hacer
19         orientar X - Y como X ← Y
20     Para cada enlace del tipo X - Z - Y tal que existe W con X → W, Y → W, y Z -
       W hacer
21         orientar Z - W como Z ← W
```

minimum description length, MDL)), 3) de información, como el *criterio de información de Akaike (1)* o el *criterio de información de Schwarz* (64).

Algoritmos de búsqueda Al ser imposible examinar todos los grafos, se utilizan técnicas de búsqueda heurística, algunos métodos trabajan en el espacio de todas las redes bayesianas y otros lo hacen en el de las clases de equivalencia de las estructuras de red. Para ampliar detalles se remite al lector a los trabajos de Spirtes y Meek (73) y Chickering (14).

El primer algoritmo de búsqueda propuesto en la literatura fue K2 (16). El punto de partida es un grafo vacío. En cada iteración, el algoritmo considera todos los enlaces posibles, es decir, todos aquellos que no forman un ciclo, y añade aquél que conduce a la red bayesiana de mayor calidad. El algoritmo 2 muestra su funcionamiento.

Algoritmo 2: Algoritmo K2.

Entrada: variables que participan en el aprendizaje, conjunto de casos
Resultado: estructura de la red aprendida

```
//Etapa de iniciación:
Ordenar las variables
Para cada variable hacer
    Π_i ← ϕ
//Etapa iterativa:
Para i=1 a n hacer
    Mientras δ > 0 y |Π_i| < número máximo de padres hacer
        seleccionar el nodo Y ∈ X_1,...,X_{i-1}
        Π_i que maximiza la métrica
        δ ← métrica(Π_i ∪ Y) - métrica(Π_i)
        Si δ > 0 entonces
            Π_i ← Π_i ∪ Y
```

Uno de los problemas del algoritmo K2 es que requiere una ordenación previa de los nodos. Para evitar este problema surge el *algoritmo B*, propuesto por Buntine en 1991 (10). La única diferencia con el algoritmo K2 es que se usa una matriz en la que se almacenan los distintos incrementos provocados en la medida de calidad al añadir un enlace (en la casilla $A[i, j]$ se almacena el incremento obtenido al añadir el enlace del nodo i-ésimo al j-ésimo). El algoritmo 3 muestra su funcionamiento.

Algoritmo 3: Algoritmo B.

Entrada: variables que participan en el aprendizaje, conjunto de casos
Resultado: estructura de la red aprendida

```
1  //Etapa de iniciación:
2  Para cada variable hacer
3      Π_i ← ϕ
4  Para i=1 a n hacer
5      Para j=1 a n hacer
6          Si i ≠ j entonces
7              A[i, j] ← m_i(X_j) − m_i(ϕ)
8          en otro caso
9              A[i, j] ← −∞ //no permitimos X_i → X_i
10 //Etapa iterativa:
11 Para i=1 a n hacer
12     Mientras A[i, j] > 0 y A[i, j] ≠ −∞, ∀i, j hacer
13         Si A[i, j] >0 entonces
14             Π_i → Π_i ∪ X_j  Para X_a ∈ Ascen_i, X_b ∈ Descen_i hacer
15                 A[a, b] ← −∞ //no permitimos ciclos
16             Para k = 1 a n hacer
17                 Si A[i,k] > −∞ entonces
18                     A[i,k] ← m_i(Π_i ∪ X_k) − m_i(Π_i)
```

Otro de los algoritmos más utilizados es el *algoritmo del gradiente*, también conocido como algoritmo "Hill Climber". Su funcionamiento se muestra en el algoritmo 4. Partiendo de un grafo vacío (o de cualquier grafo acíclico), en cada iteración del algoritmo se realiza la operación (añadir o eliminar un enlace) que maximiza la puntuación de la red, siempre que se mejore la de la red obtenida en el paso anterior. El algoritmo para cuando ninguna de las operaciones posibles mejora la puntuación de la red anterior. Véase que en cada paso si una arista con destino en el nodo X_i es añadida o eliminada, sólo es necesario reevaluar la puntuación de ese nodo en concreto. Los algoritmos de este tipo, que necesitan recalcular localmente algunas puntuaciones para obtener la puntuación de la siguiente red, se dice que tienen *actualización local de la puntuación* y, obviamente, son considerablemente más eficientes que aquellos que no la tienen.

Algoritmo 4: Algoritmo del gradiente.

```
Entrada: Red inicial, árbol de casos y métrica
Resultado: red aprendida
Mientras ha habido mejora hacer
    Se obtiene la lista de operaciones que se pueden realizar.
    Para cada operacion hacer
        La métrica calcula la puntuación de la red resultante de aplicar la operación.
        Si es mejor que la mejor puntuación entonces
            Almacenamos la puntuación y la edición.
    Si ha habido mejora entonces
        Realiza la mejor de las ediciones.
```

El principal problema de estos métodos es que se trata de *algoritmos voraces* (una vez añadido un enlace nunca se borra), con lo que laprobabilidad de quedar atrapado en un máximo local es muy alta.

Una forma de reducir (no de eliminar) este problema consiste en dar mayor flexibililidad al algoritmo, permitiendo que en cada paso se realice una de estas operaciones: añadir un enlace (siempre que no cree un ciclo), borrar un enlace o invertir un enlace (siempre que no se cree un ciclo).

El precio que se paga para reducir la probabilidad de máximos locales es una mayor complejidad computacional, pues el número de grafos que hay que examinar es mucho mayor. Otras posibles soluciones para el problema de los óptimos locales pasan por usar una mejora al algoritmo del gradiente conocida como *algoritmo del gradiente iterado* que consiste en introducir una pequeña modificación en el grafo obtenido por el algoritmo

del gradiente y volver a aplicar el algoritmo partiendo de ese nuevo grafo repetidas veces. Además, existen otros métodos alternativos como *"simulated annealing"* ((54)) o la búsqueda en calidad propuesta por Korf en 1993 (39).

e) Modelos canónicos probabilistas

El mayor problema en la construcción de un MGP es la obtención de los parámetros numéricos que describen las distribuciones de probabilidad condicionada $P(y|\mathbf{x})$. Es fácil que una determinada configuración no exista en la base de datos o si existe, que el número de casos sea poco representativo y la estimación de $P(y|\boldsymbol{x})$ sea poco fiable. Debido a todo esto, en la práctica no se construye una TPC con más de 3 o 4 padres.

Existen dos tipos de modelos canónicos: los modelos deterministas y los basados en la hipótesis de *independencia de la interacción causal* (IIC).

e.1) Modelos deterministas Es el tipo más sencillo porque no necesita ningún parámetro. El valor de Y es función del valor de los padres. La TPC es:

$$P(y|\mathbf{x}) = \begin{cases} 1 & \text{si } y = f(\mathbf{x}) \\ 0 & \text{en otro caso .} \end{cases} \tag{1.9}$$

e.2) Modelos IIC Veremos dos tipos de modelos IIC. Los modelos con ruido y los residuales, que son un caso particular.

Modelos IIC "con ruido" Los modelos IIC se construyen a partir de los modelos deterministas introduciendo n variables auxiliares $\{Z_1,\ldots,Z_m\}$, de modo que Y es una función determinista de las $Z_{i'}s$ y el valor de cada Z_i depende de forma probabilista de X_i, según la TPC $P(z_i|x_i)$. La TPC $P(y|\boldsymbol{x})$ se obtiene eliminando por suma las $Z_{i'}s$:

$$P(y|\boldsymbol{x}) = \sum_{\boldsymbol{z}} P(y|\boldsymbol{z}) \cdot P(\boldsymbol{z}|x) \tag{1.10}$$

La hipótesis de independencia de influencia causal (IIC) implica que los mecanismos causales por los cuales las $X_{i'}s$ influencian el valor de Y no interactúan entre ellas. De ahí se deduce que:

$$P(\boldsymbol{z}|x) = \prod_i P(z_i|x_i) \tag{1.11}$$

que con las ecuaciones 1.9 y 1.10 nos lleva a:

$$P(y|\boldsymbol{x}) = \sum_{\boldsymbol{z}|f(\boldsymbol{z})=y} \prod_i P(z_i|x_i) \tag{1.12}$$

Que es la ecuación general de los modelos IIC.

Cada parámetro $P(z_i|x_i)$ de un modelo IIC está asociado a un enlace particular $X_i \rightarrow Y$, mientras que para carámetro $P(y|\boldsymbol{x})$ de una TPC corresponde a una configuración $\boldsymbol{x}$ en que intervienen todos los padres de Y, y por tanto, no puede ser asociado a ningún enlace particular.

Modelos IIC residuales Cuando no se pueden incluir todas las variables que influyen en Y, si se cumplen ciertas hipótesis que permiten considerar que el resto de las variables que influyen sobre Y como un parámetro residual $P(z_L)$. Este parámetro lo representamos con la probabilidad a priori de una variable auxiliar (imaginaria).

Los dos modelos IIC más utilizados en la práctica son los OR y los MAX.

e.3) Modelos OR/MAX

Modelo OR "con ruido" Cada X_i representa una causa de Y y cada Z_i indica si X_i ha producido Y o no. El término "con ruido" significa, en este caso, que es posible que algunas causas no produzcan el efecto cuando están presentes. En este caso, $\neg z_i$ significa que X_i no ha producido Y, bien porque X_i estaba ausente, bien porque cierto inhibidor I_i ha impedido que X_i produjera Y. Si q_i es la probabilidad de que el inhibidor I_i esté activo, la probabilidad de que X_i, estando presente, produzca Y es

$$c_i = P(+z_i|+x_i) = 1 - q_i \tag{1.13}$$

Si X_i está ausente no puede ocurrir Y ($P(+z_i|\neg x_i) = 0$.

Modelo OR residual Como no es posible incluir todas las causas de un efecto se utiliza una versión residual del modelo OR, para lo que se introduce la variable Z_L, cuya probabilidad viende dada por el parámetro c_L: $P(+z_L) = c_L$.

La TPC para este modelo es:

$$P(\neg y|\mathbf{x}) = (1 - c_L) \cdot \prod_{i \in I_+(\mathbf{x})} (1 - c_i) \tag{1.14}$$

El modelo OR residual coincide con el modelo OR con ruido cuando $c_L = 0$.

Modelo MAX El modelo MAX "con ruido" es una extensión de la puerta OR "con ruido" para variables multivaluadas. Cada Z_i representa el valor de Y producido por X_i. El valor de Y es el máximo de los valores individuales producidos por cada uno de los padres: $y = f_{MAX}(\mathbf{x})$. Los parámetros para el enlace $X_i \rightarrow Y$ son:

$$c_{z_i}^{x_i} = P(z_i|x_i) \tag{1.15}$$

que equivale a

$$c_y^{x_i} = P(Z_i = y|x_i) \tag{1.16}$$

Cada $c_y^{x_i}$ representa la probabilidad de que Y tome el valor y cuando X_i toma el valor x_i y las demás causas de Y toman valores neutros.

En el modelo MAX residual, el parámetro c_y^L representa la probabilidad de que $Y = y$ cuando las demás causas de Y están en sus estados neutros:

$$c_y^L = P(y|\neg x_1, \ldots, \neg x_n) \tag{1.17}$$

f) Diagramas de influencia

f.1) Definición de diagrama de influencia Los diagramas de influencia (32) son un modelo posterior a los árboles de decisión para problemas de decisión simétricos, que corrige algunas carencias: son más compactos, fácilmente modificables, representan explícitamente la causalidad, no se realizan cálculos redundantes y son más fáciles de construir; como consecuencia, pueden representar problemas de mayor tamaño que los árboles de decisión.

Se componen de los mismos tres tipos de nodos que los árboles de decisión:

- *Nodos de azar*: representan variables, indicadas por $x_i \in X$.
- *Nodos de decisión*: representan las que el decisor puede controlar; p. ej, si se hace un test o no, si se hace una compra o no, si se aplica una terapia u otra o ningina.
- *Utilidad*: representan las preferencias del decisor, el valor subjetivo que el decisor asigna a cada uno de los escenarios posibles.

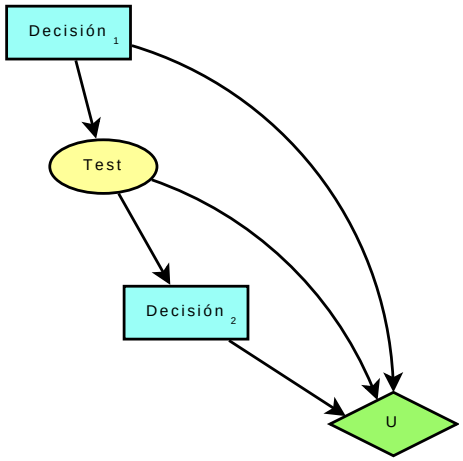


Figura 1.5: Diagrama de influencia correspondiente al árbol de decisión de la figura 1.4.

Existen dos tipos de enlaces en función de cuál sea su nodo de destino:

- *Condicionales*: si su destino es un nodo de valor o azar. Significa dependencia funcional o probabilista respecto a los nodos de origen.
- *Informativos*: si su destino es un nodo de decisión. Significa que la variable en el origen del enlace es conocida cuando se toma la decisión.

El grafo de un diagrama de influencia cumple las siguientes propiedades:

- Es un gráfo dirigido acíclico.
- Tiene que haber un camino dirigido que contenga todos los nodos de decisión.
- Los nodos de utilidad no tienen hijos[2].

Por la segunda condición aplicable al grafo, existe un orden de precedencia temporal entre las decisiones y las variables de azar están particionadas en subconjuntos disjuntos I_i dentro de ese orden temporal, lo denotamos así: $I_0 \prec D_1 \prec I_1 \prec D_2 \prec \ldots \prec D_n \prec I_n$. Cuando se toma la decisón D_i se conocen todas las variables aleatorias que la preceden (pasado de D_i).

f.2) Políticas, estrategias y valores esperados

[2]Los nodos super valor son una ampliación posterior del modelo, definida en (74)

Definición 1.27 (Política de una decisión D_i) *Es una correspondencia δ_i entre cada configuración del pasado de D_i y un valor para D_i.*

Definición 1.28 (Estrategia) *Es un conjunto de políticas, una por cada decisión.*

El objetivo de la evaluación es encontrar la política óptima Δ, que es el conjunto de todas las reglas de decisión óptimas: $\Delta = \{\delta_1^*, \ldots, \delta_n^*\}$. Esta evaluación es más eficiente que en los árboles de decisión porque no se repiten cálculos pero los algoritmos son más complejos.

- Diagramas de influencia
  - Exactos
    - Conversión en árbol de decisión (32)
    - Inversión de arcos (59; 65)
    - Conversión en red bayesiana (15)
    - Eliminación de variables
  - Aproximados
    - Algoritmos estocásticos
    - Deterministas

Figura 1.6: Esquema resumen de los principales métodos de propagación para diagramas de influencia.

Un diagrama de influencia siempre puede convertirse en un árbol de decisión equivalente que tiene como variables desde la raíz hasta las hojas las del camino dirigido que pasa por todas las decisiones. Lo contrario no es cierto en general: en un árbol de decisión cada rama puede tener un orden distinto para sus decisiones o suprimir algunas, mientras que en un diagrama de influencia, el orden de las decisiones es siempre el mismo. Por este motivo, los primeros algoritmos convertían los diagramas de influencia en árboles de decisión, posteriormente se desarrollaron algoritmos específicos.

Descripción de algunos algoritmos para evaluar diagramas de influencia En los diagramas de influencia la inferencia consiste en hallar la política que conduce a la máxima utilidad. En un diagrama de influencia está definido el orden en el que se toman las decisiones, que siempre es el mismo y que supone que existe un camino dirigido a través de los nodos que componen el diagrama de influencia que recorre todas las decisiones en el orden total.

Las variables se pueden clasificar en los conjuntos disjuntos formados por los predecesores de cada variable: $I_0, I_1, \ldots I_n$ que definen un orden arcial: $I_0 \prec D_1 \prec I \prec \ldots \prec D_n \prec I_n$.

Inversión de arcos El proceso es el siguiente:

1. Mientras haya algún antecesor X del nodo de valor V: eliminar el nodo de azar.

   - Los potenciales se actualizan: $\psi'_v = \sum_x \psi_v \phi_x$
   - Los antecesores de X pasan a ser antecesores de V.

   Se elimina el nodo de decisión D:

   - Los potenciales se actualizan: $\psi'_v = max_x \psi_v$
   - Los antecesores de D pasan a ser nodos sumidero (no se tienen en cuenta).

2. Para poder eliminar un nodo de azar no puede tener descendientes que no sean el nodo de utilidad. Para conseguirlo, si el nodo A tiene como hijo el nodo B se puede invertir la dirección del enlace, pero el precio que se paga es que hay que trazar arcos desde los padres de ambos hasta A:

Eliminación de variables El proceso es el siguiente:

1. Se determina el orden parcial $I_0 \prec D_1 \prec I \prec \ldots \prec D_n \prec I_n$, donde I_n son los nodos que no son antecesores de ningún nodo de decisión, I_{n-1} son los antecesores de D_n, etc.

2. Se van seleccionando variables para eliminar respetando el orden parcial. Cuando se selecciona una variable X los potenciales se actualizan de la siguiente forma:

   - $\Phi_X = \{\phi \in \Phi | X \in dom(\phi)\}$
   - $\Psi_X = \{\phi \in \Psi | X \in dom(\phi)\}$

   Si X es una variable de azar:

   - $\phi_X = \sum_X \prod \Phi_X$
   - $\psi_X = \sum_X \prod \Phi_X (\sum \Psi_X)$

   Si X es una variable de decisión:

   - $\phi_X = max_X \prod \Phi_X$
   - $\psi_X = max_X \prod \Phi_X (\sum \Psi_X)$

El conjunto de potenciales tras la eliminación queda:

- $\Phi = \{\Phi \backslash \Phi_X\} \bigcup \{\phi_X\}$
- $\Psi = \{\Psi \backslash \Psi_X\} \bigcup (\frac{\psi_X}{\phi_X})$

1.1.4. Otros MGPs para análisis de decisiones

Redes de análisis de decisiones (RAD)

Procesos de decisión de Markov (en inglés, *Markov Decision Process (MDP)*) Modelan procesos de decisión en los que algunas variables están bajo el control del decisor y otras son aleatorias; véase (8).

Procesos de decisión de Markov parcialmente observables (en inglés, *Partially observable Markov Decision Process (POMDP)* Es una generalización del modelo anterior en la que se asume que la dinámica del sistema está determinada por un MDP y en lugar de observar el estado del sistema, se le asigna una distribución de probabilidad; véase (5).

Diagramas de influencia con memoria reducida (en inglés Dynamic LImited-Memory Influence Diagrams (LIMIDs)) Es un método de resolución de diagramas de influencia en el que se relaja la condición de tener en cuenta la totalidad del pasado de cada decisión y sin nesidad de tener un orden parcial para las decisiones; véase (42).

Diagramas de influencia dinámicos con memoria reducida (en inglés Dynamic LImited-Memory Influence Diagrams (DLIMIDs)) Es una extensión de los LIMIDs para representar procesos de decisión con horizonte infinito; véase (76)

1.2. Ingeniería del software

La ingeniería del software se puede definir como *"una disciplina de ingeniería que comprende todos los aspectos de la producción de software, desde las etapas iniciales de la especificación del sistema hasta el mantenimiento de éste"* (69). El ámbito, por tanto, es muy amplio. Algunos aspectos tienen que ver con la tecnología de ordenadores, otros con la economía, la gestión, la psicología, etc.

1.2.1. Paradigmas de desarrollo

En la literatura se describen varios paradigmas de desarrollo, los más ampliamente utilizados son dos[3]: estructurado y orientado a objetos. El paradigma estructurado ve un sistema de software como una jerarquía piramidal de módulos y centra su atención en los procesos, en cambio, el paradigma de orientación a objetos ve un sistema como un conjunto de tipos de estructuras de datos que colaboran para realizar cientas funciones. Existe también un nuevo paradigma de orientación a aspectos (38), cuya aportación fundamental es la modularización de las incumbencias trasversales (código que realiza ciertas funciones y que, por su naturaleza, está repartido por todo el programa, por ejemplo: el tratamiento de excepciones, escribir las operaciones que se van haciendo en un registro, etc.); la orientación a aspectos es más una extensión o complemento del diseño orientado a objetos que un cambio de paradigma.

En el desarrollo de Carmen hemos utilizado el paradigma orientado a objetos. Las principales ventajas sobre el paradigma estructurado son dos: la mejor encapsulación de los datos (se accede a un objeto a través de su interfaz, que oculta la información no esencial para manipularlo) y la herencia (cada clase especializada tiene las variables y métodos de su superclase genérica), etc. En los momentos iniciales de este proyecto consideramos la posibilidad de utilizar la orientación a aspectos, pero lo desestimamos porque no es una tecnología suficientemente madura y no existen muchas personas familiarizadas con los lenguajes orientados a aspectos (lo cual dificulta el que otras personas puedan aportar contribuciones fácilmente a esta herramienta de software libre).

[3] También se habla del paradigma funcional (programación declarativa) y del paradigma lógico, pero su uso se ha limitado a áreas muy especializadas.

1.2.2. Ciclo de vida

Un *modelo de procesos* o *ciclo de vida* es una descripción simplificada de todo el proceso de producción de software. Los tres modelos generales de procesos que se estudian en la literatura son: el modelo en cascada, el modelo en espiral y el modelo basado en componentes.

En todos estos modelos el ciclo de vida se divide en una serie de fases, que son: especificación, diseño, codificación y pruebas. Casi siempre existe una fase de mantenimiento, que se puede prolongar hasta que el software deja de utilizarse.

Una metodología es una forma de sistematizar el proceso de desarrollo; básicamente es el manual o guía que se pone en práctica al abordar la construcción de un sistema. Cada metodología elige un ciclo de vida en función de las características del proyecto y organiza las fases de un modo concreto; algunas incluso están apoyadas por herramientas hechas a medida (tales como, el método RUP de Rational (34)).

Cada metodología organiza a su modo las fases; por ejemplo, según Métrica 3 (55), una fase se divide en actividades; cada actividad se descompone en un conjunto de tareas, necesita un conjunto de entradas y produce un conjunto de salidas. Las tareas son los átomos de los que está hecha la metología: son acciones elementales orientadas a un único objetivo. Las entradas y salidas de las actividades y tareas son: documentación, código fuente, planificación de tareas, pruebas y resultados, etc.

Especificación

La especificación es la fase de elicitación[4] y representación de requisitos. Existen dos tipos: funcionales y no funcionales. Un requisito funcional se puede definir como una acción del sistema que produce un resultado visible por el usuario. Un requisito no funcional es una característica del sistema que no es visible pero tiene influencia en su funcionamiento o en el mantenimiento.

Como resultado del análisis de los requisitos especificados inicialmente, se obtiene un *modelo del problema* que es necesario *representar* de una manera formal, con el fin de poder aplicar técnicas sistemáticas para su resolución. Existen diversos lenguajes y métodos de especificación, que tienen características diferentes según la metodología que se utilice posteriormente en el desarrollo; por ejemplo: diagramas entidad-relación

[4]La elicitación consiste en hacer explícito el conocimiento que estaba en la mente de un experto. (Somos conscientes de que estamos introduciendo un neologismo pero consideramos que en este caso está justificado porque no existe un término equivalente en castellano).

para modelado, diagramas contextuales, plantillas o marcos conceptuales, diagramas funcionales, etc. El método más extendido de representación es mediante técnicas orientadas a objetos y los lenguajes asociados a ellas.

El *análisis orientado a objetos* representa el problema en términos de clases, sus propiedades y sus relaciones. El modelo que resulta de este análisis utiliza técnicas muy diversas. Algunas, como la de los casos de uso, son aplicables también al análisis estructurado. Las ventajas del análisis orientado a objetos son que la representación que se logra es más próxima a la realidad y que facilita la especificación técnica de las requisitos de usuario, que inicialmente estaban definidos en lenguaje natural.

Diseño

El diseño es un proceso que, partiendo de la especificación y usando diferentes técnicas y principios, define un producto con el suficiente nivel de detalle como para poder ser implementado en un ordenador.

El diseño es la parte técnicamente más difícil de un proyecto y está menos sistematizada que la especificación o las fases posteriores. Algunos autores, como Ian Sommerville (69), piensan que el diseño es esencialmente una actividad creativa, mientras que otros, como Roger S. Pressman (61), lo ven como un proceso estructurado.

Existen dos tipos de diseño: arquitectónico y detallado.

a) Diseño arquitectónico Resuelve tres problemas importantes: la descomposición del sistema en subsistemas, la comunicación entre subsistemas y el modo de realizar el control.

a.1) Descomposición del sistema en subsistemas. La descomposición orientada a objetos crea subsistemas, cada uno de ellos formado por un conjunto de clases cuyo acoplamiento mutuo es relativamente alto. Cada subsistema es autónomo del resto y se relaciona con otros subsistemas a través de su interfaz. En principio, un subsistema puede ser sustituido por otro que implemente la misma interfaz.

a.2) Intercambio de información entre subsistemas. Según Sommerville (69) existen varios modelos organizativos: *repositorio*, que tiene o bien una base de datos central o una base de datos privada en cada uno de los componentes, los cuales intercambian mensajes entre sí; *cliente-servidor*, basado en un esquema del tipo

petición-respuesta; y, por último, el modelo en *capas*, que divide el sistema en un conjunto de capas o máquinas abstractas de modo tal que cada una de ellas se comunica con las capas adyacentes a través de las interfaces que ofrecen.

a.3) Modelo de control. El control se puede definir como la forma en la que los sub-sistemas se coordinan para funcionar como un todo. El control puede ser *centralizado*, lo que supone tener un sub-sistema que se encarga de esa tarea o, *dirigido por eventos*, en el que cada sub-sistema asume parte de la responsabilidad.

b) Diseño detallado En el diseño detallado se concretan los algoritmos y estructuras de datos. Está relacionado con el lenguaje de programación pero, a diferencia de éste, se preocupa más de los aspectos semánticos que de los sintácticos. Su punto de partida es la arquitectura. Si se cuenta con la herramienta adecuada, partiendo del diseño detallado se puede construir el código automáticamente y viceversa.

b.1) Documentación: especificación del diseño. Evidentemente uno de los puntos más importantes de la documentación es la especificación del diseño. Dado que el diseño tiene una gran componente de invención, es necesario dejar muy claramente documentadas las decisiones y los elementos utilizados en el diseño. Además, el diseño será el elemento clave para la fase siguiente de implementación, que debe seguir fielmente los dictados del diseño.

Para dejar constancia de los diseños se deben utilizar lenguajes lo más formales posible, como tablas, diagramas y pseudocódigo, que en algunos casos pueden permitir incluso la utilización de herramientas para automatizar parcialmente el proceso de construcción del código en la siguiente fase de implementación.

Uso de patrones. En otras ramas de la ingeniería establecidas desde hace más tiempo, existen prototipos de soluciones estándar para problemas conocidos; por ejemplo, los tornillos siguen una normativa en cuanto a diámetro de la rosca, longitud, material, forma de la cabeza, etc.

También en informática es frecuente encontrarse con el mismo tipo de problema más de una vez. Si se ha diseñado una solución para ese problema sería deseable aplicarla de algún modo en problemas similares. Un patrón da una solución ya ensayada para

un tipo de problema. La idea es una adaptación a la informática de los conceptos de arquitectura definidos por Christopher Alexander en los años 70 (2).

Un *patrón* de diseño identifica las clases y sus instancias, sus papeles y colaboraciones y la distribución de responsabilidades.

Un *marco* (en inglés *framework*) es un conjunto integrado de componentes que colaboran para proporcionar una arquitectura reutilizable para una familia de aplicaciones.

Las diferencias entre los patrones y los marcos son:

- El grado de abstracción es mayor en el caso del patrón.
- El tamaño: los patrones son elementos arquitectónicos más pequeños (un marco puede contener varios patrones).
- La especialización: los patrones están menos especializados que los marcos.

Tipos de patrones Hay muchas clasificaciones posibles; la que se propone en el libro de referencia de patrones (29) es esta:

- *Patrones de análisis*: Según Martin Fowler (28), *permiten capturar modelos conceptuales relativos a un dominio de aplicaciones para ser reutilizados en otras aplicaciones*. Se centran en aspectos organizativos, sociales y económicos.
- *Patrones de diseño arquitectónico*: Definen la organización estructural de un sistema. Consisten en un conjunto de subsistemas predefinidos y cada uno con sus responsabilidades. Incluyen reglas para organizar las relaciones entre ellos.
- *Patrones de diseño detallado*: Son esquemas para refinar los subsistemas o componentes de un sistema o las relaciones entre ellos. Describen estructuras de comunicaciones entre componentes en un contexto.
- *Patrones de codificación*: Son el conjunto de normas o estándares que describen cómo implementar aspectos particulares de los componentes o relaciones entre ellos con un lenguaje de programación.
- *Patrones de organización*: Describen la estructuración del personal en las organizaciones.

La diferencia entre unos tipos y otros está tanto en el nivel de abstracción como en el contexto en el que son aplicables.

Además de esta clasificación general se pueden encontrar otros tipos de patrones para situaciones más específicas, tales como programación concurrente, interfaces gráficas, organización y optimización del código, etc.

Características de los patrones Para que un patrón se pueda considerar como tal debe pasar unas pruebas llamadas *test de patrones*; mientras tanto, recibe el nombre provisional de *proto-patrón*. Según Jim Coplien (17) un patrón debe cumplir estos requisitos:

- soluciona un problema,
- la solución no es obvia,
- ha sido probado,
- describe participantes y relaciones entre ellos y
- tiene un componente humano[5].

Codificación

La codificación consiste en traducir el diseño detallado a un lenguaje de programación. Cuando varios programadores trabajan en un mismo proyecto, conviene definir un *estándar de codificación*, que es un conjunto de normas para realizar la codificación, que comprende todos los aspectos de la generación de código. El objetivo es que el código fuente tenga un estilo armonioso, como si un único programador hubiera escrito todo el código de una sola vez. Las técnicas de codificación incorporan muchos aspectos del desarrollo del software. Aunque generalmente no afectan a la funcionalidad de la aplicación, sí contribuyen a una mejor compresión del código fuente. Cuando el proyecto de software incorpora código fuente previo, o bien cuando se realiza el mantenimiento de un sistema de software creado anteriormente, el estándar de codificación debería determinar cómo operar con la base de código existente. En general una técnica de codificación no pretende formar un conjunto inflexible de estándares de codificación, de hecho; las normas deberían ser sometidas a revisiones periódicas.

[5] Aunque el autor no explica el porqué de este último.

Pruebas

Las pruebas tienen dos finalidades. Por un lado, la *validación*, cuyo objetivo es garantizar que se han implementado las funcionalidades requeridas, y por otro lado la *verificación*, que consiste en asegurar en la medida de lo posible que la implementación no contiene errores. Esta distinción es más académica que práctica; en realidad, ambos tipos de comprobaciones se realizan simultáneamente. Para realizar estas pruebas se han desarrollado varias técnicas que comentaremos brevemente.

El proceso de prueba se realiza en varios niveles: las *pruebas de unidad* comprueban que cada clase o módulo realiza sus funciones correctamente, independientemente del resto del programa; las *pruebas de integración* comprueban que varias clases o módulos pueden interactuar conjuntamente sin errores; las *pruebas de validación* se centran en los requisitos del cliente; y finalmente, *las pruebas de sistema* comprueban todo el sistema en su conjunto (software y hardware) y la forma en la que se relacionan los subsistemas.

Existe una gran variedad de pruebas de verificación. La mayor parte de ellas aprovecha la modularidad del diseño y de la implementación. Se pueden agrupar en dos tipos: *pruebas de caja blanca* y *pruebas de caja negra*. Las pruebas de caja blanca tienen en cuenta la estructura del código, que se representa en forma de grafo, y que consisten básicamente en la exploración de los diferentes caminos. Las pruebas de caja negra tienen en cuenta las entradas y las salidas, considerando el código probado como un ente cuyo funcionamiento interno se desconoce. Estas pruebas consisten en encontrar clases de equivalencia en el dominio de los datos de entrada y salida[6]; se basan también en observaciones empíricas realizadas en los lugares donde es más probable encontrar errores; por ejemplo: los extremos de los intervalos permitidos en los valores de entrada.

Las técnicas anteriores se implementan mediante la creación de *casos de prueba*. Un caso de prueba consta de un conjunto de datos de entrada y el resultado correcto para dichos datos. Estos casos en las pruebas de caja blanca se diseñan de modo que recorran el grafo del programa de varias formas. En las pruebas de caja negra se diseñan para que analicen todas las posibles clases de equivalencia de los datos de entrada y salida. Según las metodologías ágiles, como por ejemplo *Extreme Programming* (7), la forma ideal de proceder en el diseño de pruebas es escribir *antes* las pruebas y después escribir el código que satisfaga esas pruebas.

Las *pruebas de regresión* son pruebas ya realizadas que se repiten cuando se hace una

[6] Las clases de equivalencia se definen por el conjunto de estados válidos o inválidos para las condiciones de entrada o salida.

modificación en el código, para comprobar que esa modificación no ha producido efectos colaterales que puedan causar errores.

Cuando se descubre un error en el funcionamiento normal del programa (finalizada la fase de pruebas), lo que conviene hacer (según la metodología *Extreme Programming*) es codificar los datos de entrada que provocan ese error, junto con la salida correcta, como un caso de prueba.

Las pruebas exaustivas no son posibles, porque la complejidad de los programas crece de un modo exponencial con su tamaño, pero con las técnicas conocidas y haciendo un diseño modular, la densidad de errores se reduce drásticamente.

Además de las técnicas comentadas, existen otras formas de garantizar la corrección; por ejemplo, los *métodos formales*, que utilizan una especificación del programa en un lenguaje basado en teoría de conjuntos y, en algunos casos, aplican algoritmos que generan el código a partir de dichas especificaciones. La corrección del programa está garantizada matemáticamente porque los algoritmos que lo construyen lo garantizan; siempre que no existan errores en la propia especificación. Los métodos formales son bastante complicados y es difícil encontrar personal formado en los mismos por lo que su aplicación se reduce a ámbitos muy concretos, en los que la fiabilidad es crucial, como el sector aeroespacial o las centrales nucleares. En cualquier caso, la especificación formal de los requisitos es una tarea que no está libre de errores.

Parte II

Nuevos algoritmos

Capítulo 2

Operaciones con potenciales de variables discretas

Un potencial es una función que asigna a cada configuración de un conjunto de variables un número real. En los modelos gráficos probabilistas cada distribución de probabilidad y cada función de utilidad definidas sobre un conjunto de variables discretas es un potencial. El proceso de inferencia produce nuevos potenciales. En principio, los potenciales definidos sobre variables discretas pueden representarse como arrays multidimensionales. En este capítulo demostraremos que en el caso de operaciones con potenciales grandes, el coste computacional de recuperar los elementos es significativamente mayor que el de multiplicarlos, hallar el máximo o sumarlos. Presentamos un algoritmo que recupera secuencialmente los elementos de un potencial implementado como un array lineal sin multiplicar las coordenadas de cada configuración por los desplazamientos. Se analiza teórica y empíricamente la mejora conseguida cuando se utiliza en operaciones de suma, multiplicación, marginalización y condicionamiento de potenciales. También consideramos las ventajas de multiplicar varios potenciales al mismo tiempo y las de integrar la multiplicación y la marginalización de potenciales.

2.1. Introducción

Un *potencial* ψ definido sobre un conjunto de variables $\mathbf{V}$ es una función que hace corresponder cada posible configuración $\mathbf{v}$ con un número real, conocido como el *valor* de la configuración. En los modelos gráficos probabilistas (MGP), como

redes bayesianas, diagramas de influencia, modelos de Markov, grafos en cadena, etc., cada familia de distribuciones de probabilidad (por ejemplo, la distribución de probabilidad de un nodo dados sus padres, generalmente expresado como una tabla de probabilidades condicionales) y cada función de utilidad es un potencial (13; 18; 35; 60). La inferencia en MGPs conlleva operaciones con potenciales tales como marginalización, suma, multiplicación y división, que producen nuevos potenciales. Una *variable de estados* es una variable discreta que tiene un dominio finito. Si todas las variables con las que se define un potencial son de estados, el potencial puede representarse como un array multidimensional. Los lenguajes de programación generales admiten arrays multidimensionales, pero sólo si sus dimensiones son conocidas en tiempo de compilación. El problema en el caso de los algoritmos para modelos gráficos probabilistas es que las dimensiones de los potenciales no se conocen hasta que el programa está en ejecución. La solución estándar es almacenar todos los valores del potencial en un array lineal. El orden en el que se almacenan los valores en el array lineal lo llamaremos *orden natural* y depende del orden y dimensiones de las variables. La posición de cada valor se puede calcular multiplicando las coordenadas de la correspondiente configuración por el desplazamiento de las variables.

A menudo cuando se opera con potenciales es necesario recuperar sus valores en un orden impuesto por la operación que se va a realizar (véase 2.3), que suele ser distinto del orden natural. El método habitual de recuperación consiste en generar las configuraciones en el orden necesario y calcular la posición de cada configuración, como se menciona más arriba, lo cual implica N multiplicaciones y $N-1$ sumas para cada configuración de N variables. Por lo tanto el coste de recuperar el valor de una configuración es mucho mayor que el coste de realizar una suma, multiplicación o comparación para ese valor. Por este motivo el cálculo de la complejidad de los algoritmos que sólo tiene en cuenta el número de operaciones elementales sobre los potenciales es incorrecto.

El objetivo de este capítulo es describir un método alternativo para recuperar los elementos de un potencial en un orden cualquiera, es decir, un orden que no se conoce cuando se construye el potencial. En lugar de multiplicar las coordenadas por los desplazamientos, la posición correspondiente a cada configuración se calcula sumando un *desplazamiento acumulado* a la posición de la configuración previa. La ventaja de esta aproximación es que sólo necesita una suma por cada configuración.

El resto del capítulo se organiza como sigue. La sección 2.2 presenta las definitiones básicas y una forma de calcular los desplazamientos acumulados dados dos conjuntos

ordenados de variables: **X** son las variables en el orden en el que aparecen en el potencial (orden natural) e **Y** indica el orden en el que las configuraciónes deben recuperarse. La sección 2.3 muestra cómo usar los desplazamientos acumulados en operaciones básicas sobre potenciales, tales como marginalización, suma, multiplicación, división y condicionamiento; se compara el rendimiento del sistema tradicional con el nuevo tanto teórica como empíricamente. La sección 2.4 propone dos formas de obtener mejoras adicionales operando con potenciales tanto con el método tradicional como con el de los desplazamientos acumulados: multiplicar varios potenciales simultaneamente en vez de secuenciamente (apartado 2.4.1) y multiplicar y marginalizar simultaneamente (apartado 2.4.2). La sección 2.5 analiza trabajos relacionados en la literatura y menciona algunas posibles líneas de investigación.

2.2. Desplazamientos acumulados

2.2.1. Desplazamientos y posiciones

Dado un conjunto ordenado **X** de $\mathbf{N_X}$ variables de estados finitos $\{X_0,\ldots,X_{N_X-1}\}$, una *configuración* **x** es un elemento de $\mathbf{X}^*$ (el producto cartesiano de los dominios de las variables), que puede ser expresado como un array de $N_\mathbf{x}$ enteros, $[x_0,\ldots,x_n]$; para cada *coordenada* x_i, tenemos que min(x_i)=0 y max(x_i)=$|X_i|-1$, donde $|X_i|$ es el tamaño del dominio de la variable X_i.

Definición 2.1 *Dada una configuración* **x** *distinta de la última, la función* varIncr(**x**) *se define como*[1]*:*

$$varIncr(\mathbf{x}) = \min_{0\leq i<N_\mathbf{X}} \{i | x_i < \max(x_i)\} \tag{2.1}$$

Por ejemplo, si todas las variables son binarias, $varIncr([1,1,0,1]) = 2$, dado que la primera variable que no ha alcanzado su máximo valor es X_2.

Evidentemente, cuando **x** no es la última configuración las variables anteriores a la que se va a incrementar han llegado a su máximo valor.

$$0 \leq i < varIncr(\mathbf{x}) \Longrightarrow x_i = \max(x_i) = |X_i| - 1 \tag{2.2}$$

En nuestro ejemplo, $x_0 = x_1 = 1$

[1] Cuando se recorre un potencial siguiendo su orden natural.

Definición 2.2 *Dada una configuración* $\mathbf{x}$ *distinta de la última, la* siguiente configuración (sgte($\mathbf{x}$)) *se define como:*

$$sgte(\mathbf{x}) = \mathbf{x}' | \begin{cases} x'_i = 0 & si\ i < varIncr(\mathbf{x}) \\ x'_i = x_i + 1 & si\ i = varIncr(\mathbf{x}) \\ x'_i = x_i & si\ i > varIncr(\mathbf{x})\ . \end{cases} \tag{2.3}$$

Por ejemplo, $sgte([1,1,0,1]) = [0,0,1,1]$. Hay que tener en cuenta que estamos incrementando las variables de izquierda a derecha para que las definiciones y los algoritmos sean más intuitivos.

Definición 2.3 *Dado un conjunto ordenado* $\mathbf{X}$*, el desplazamiento de un entero* i *en* $\{-1,\ldots,N_X - 1\}$ *es*

$$despl_{\mathbf{X}}(i) = \begin{cases} 1 & si\ i = 0 \\ |X_{i-1}| \times despl_{\mathbf{X}}(i-1) & si\ 0 < i < N_{\mathbf{X}} \\ 0 & si\ i = -1\ . \end{cases} \tag{2.4}$$

Cuando $i \neq -1, i$ representa el índice de una variable, X_i, y $despl_{\mathbf{X}}(i)$ indica cuantas posiciones hacia delante debemos desplazarnos en el array lineal que representa un potencial cuando la coordenada x_i se incrementa en una unidad y las otras coordendas de la configuración no varían. El motivo de definir $despl_{\mathbf{X}}(-1) = 0$ se verá en la sección 2.2.3.

Definición 2.4 *La posición de una configuración* $\mathbf{x}$ *de* $\mathbf{X}$ *es un entero dado por*

$$pos_{\mathbf{X}}(\mathbf{x}) = \sum_i x_i \times despl_{\mathbf{X}}(i) \tag{2.5}$$

Proposición 2.5 *Si* $\mathbf{x}$ *no es la última configuración,*

$$pos_{\mathbf{X}}(sgte_{\mathbf{X}}(\mathbf{x})) = pos_{\mathbf{X}}(\mathbf{x}) + 1 \tag{2.6}$$

La función $pos_{\mathbf{X}}$ induce una correspondencia uno a uno entre las configuraciones y el conjunto de $|\mathbf{X}^*|$ enteros, de modo que la primera configuración es 0 y la posición de la última es $|\mathbf{X}^*| - 1$. En consecuencia, una función ψ (por ejemplo, un potencial) definido en un conjunto de variables $\mathbf{X}$ puede representarse como un array lineal almacenando el valor de $\psi(\mathbf{x})$ en la posición $pos_{\mathbf{X}}(\mathbf{x})$.

2.2.2. Ejemplo de desplazamientos acumulados

Supongamos que tenemos un potencial $\psi_{\mathbf{X}}$ definido sobre $\mathbf{X} = \{A, B, C\}$ y necesitamos un nuevo potencial $\psi_{\mathbf{Y}}$ definido sobre $\mathbf{Y} = \{B, D, A, C\}$ como sigue: $\psi_{\mathbf{Y}}(b, d, a, c) = \psi_{\mathbf{X}}(a, b, c)$. La primera columna de la tabla 2.1 representa las configuraciones de $\psi_{\mathbf{Y}}$ y la segunda sus posiciones en $\psi_{\mathbf{Y}}$. Para que el ejemplo quede más claro hemos escrito las configuraciones con el formato $[b_0, d_0, a_0, c_0]$ en vez de $[0, 0, 0, 0]$. La tercera columna representa la configuración de $\mathbf{X}$ para cada configuración de $\mathbf{Y}$ y la cuarta columna representa su posición en $\psi_{\mathbf{X}}$; las configuraciones no están en el orden natural de $\mathbf{X}$ sino en el orden impuesto por $\mathbf{Y}$. La quinta columna representa el índice de la variable que será incrementada para calcular la nueva configuración según la ecuación 2.14, con el nombre de la variable indicado en paréntesis. La última columna representa el *desplazamiento acumulado*, es decir, el número entero que se tiene que sumar a la posición de la configuración actual (en $\psi_{\mathbf{x}}$) para obtener la siguiente configuración. La clave del método propuesto reside en que el desplazamiento acumulado (sexta columna) es función de la variable que se va a incrementar (quinta columna). Esta función, que llamamos $desplAcc_{\mathbf{X},\mathbf{Y}}$ depende del orden de las variables en $\mathbf{X}$, el orden en $\mathbf{Y}$ y la dimensión de las variables. Como esta función devuelve un entero por cada variable en $\mathbf{Y}$, puede ser representado por un vector, es decir, un array unidimensional de $N_{\mathbf{Y}}$ enteros. En el ejemplo el vector de desplazamientos acumulados es $[\underset{2}{B}, \underset{-2}{D}, \underset{-1}{A}, \underset{1}{C}]$.

2.2.3. Cálculo de los desplazamientos acumulados

Después de la explicación intuitiva basada en el ejemplo anterior veremos las definiciones formales y el teorema principal. En las próximas secciones i se referirá a una variable en $\mathbf{X}$ y j a una variable en $\mathbf{Y}$.

Definición 2.6 *Dados dos conjuntos ordenados de variables,* $\mathbf{X} = \{X_0, \ldots, X_{N_{\mathbf{X}}-1}\}$ *e* $\mathbf{Y} = \{Y_0, \ldots, Y_{N_{\mathbf{Y}}-1}\}$*, tales que* $\mathbf{X} \subseteq \mathbf{Y}$*, definimos una ordenación* σ *como una función* $\sigma : \{0, \ldots, N_{\mathbf{Y}}-1\} \longmapsto \{-1, \ldots, N_{\mathbf{X}}-1\}$ *como sigue:*

$$\sigma(j) = \begin{cases} i & si\ \exists i, 0 \leq i < N_{\mathbf{X}} - 1, X_i = Y_j \\ -1 & en\ otro\ caso\ . \end{cases} \tag{2.7}$$

En la implementación almacenaremos σ como un vector de $N_{\mathbf{Y}}$ elementos. En el ejemplo anterior, $\sigma = [\underset{B}{1}, \underset{D}{-1}, \underset{A}{0}, \underset{C}{2}]$, porque $Y_0 = B = X_1$, $Y_1 = D$, que no está en $\mathbf{X}$,

$\mathbf{y}$	$pos_{\mathbf{Y}}(\mathbf{y})$	$\mathbf{y}^{\downarrow\mathbf{X}}$	$pos_{\mathbf{X}}(\mathbf{y}^{\downarrow\mathbf{X}})$	$varIncr(\mathbf{y})$	$desplAcc$
$[b_0,d_0,a_0,c_0]$	0	$[a_0,b_0,c_0]$	0	0 (B)	$+2$
$[b_1,d_0,a_0,c_0]$	1	$[a_0,b_1,c_0]$	2	1 (D)	-2
$[b_0,d_1,a_0,c_0]$	2	$[a_0,b_0,c_0]$	0	0 (B)	$+2$
$[b_1,d_1,a_0,c_0]$	3	$[a_0,b_1,c_0]$	2	2 (A)	-1
$[b_0,d_0,a_1,c_0]$	4	$[a_1,b_0,c_0]$	1	0 (B)	$+2$
$[b_1,d_0,a_1,c_0]$	5	$[a_1,b_1,c_0]$	3	1 (D)	-2
$[b_0,d_1,a_1,c_0]$	6	$[a_1,b_0,c_0]$	1	0 (B)	$+2$
$[b_1,d_1,a_1,c_0]$	7	$[a_1,b_1,c_0]$	3	3 (C)	$+1$
$[b_0,d_0,a_0,c_1]$	8	$[a_0,b_0,c_1]$	4	0 (B)	$+2$
$[b_1,d_0,a_0,c_1]$	9	$[a_0,b_1,c_1]$	6	1 (D)	-2
$[b_0,d_1,a_0,c_1]$	10	$[a_0,b_0,c_1]$	4	0 (B)	$+2$
$[b_1,d_1,a_0,c_1]$	11	$[a_0,b_1,c_1]$	6	2 (A)	-1
$[b_0,d_0,a_1,c_1]$	12	$[a_1,b_0,c_1]$	5	0 (B)	$+2$
$[b_1,d_0,a_1,c_1]$	13	$[a_1,b_1,c_1]$	7	1 (D)	-2
$[b_0,d_1,a_1,c_1]$	14	$[a_1,b_0,c_1]$	5	0 (B)	$+2$
$[b_1,d_1,a_1,c_1]$	15	$[a_1,b_1,c_1]$	7	—	—

Tabla 2.1: Esta tabla muestra para cada configuración $\mathbf{y}$ su posición en el potencial $\psi_{\mathbf{Y}}$, su proyección en $\mathbf{X}$, la posición de su proyección en $\psi_{\mathbf{X}}$, la variable a incrementar y el desplazamiento acumulado que el el valor que tenemos que añadir a $pos_{\mathbf{X}}(\mathbf{y}^{\downarrow\mathbf{X}})$ para conseguir $pos_{\mathbf{X}}(sgte(\mathbf{y})^{\downarrow\mathbf{X}})$. El valor de los desplazamientos acumulados (la sexta columna) es función de la variable a incrementar (quinta columna).

$Y_2 = A = X_0$ e $Y_3 = C = X_2$.

La función inversa, σ^{-1} es

$$\sigma^{-1}(i) = j|(0 \leq j < N_{\mathbf{Y}}, X_i = Y_j) \tag{2.8}$$

En el ejemplo anterior $\sigma^{-1} = [\underset{2}{A},\underset{0}{B},\underset{3}{C}]$, porque $X_0 = A = Y_2$, $X_1 = B = Y_0$ y $X_2 = C = Y_3$.

Definición 2.7 *La proyección de y sobre* $\mathbf{X}$ *que representamos mediante* $y^{\downarrow x}$*, se define como:*

$$y^{\downarrow\mathbf{x}} = \mathbf{x}|(\forall i, 0 \leq y < N_{\mathbf{X}}), x_i = y_{\sigma^{-1}(i)} \tag{2.9}$$

Esta proyección también se puede calcular como sigue:

$$\mathbf{y}^{\downarrow\mathbf{X}} = \mathbf{x}|\left(\forall j, 0 \leq \sigma(j) < N_{\mathbf{X}} \Rightarrow x_{\sigma(j)} = y_j\right). \tag{2.10}$$

En la sección 2.2.4 demostraremos que la ecuación 2.10 es más eficiente que la ecuación 2.9, y por eso usaremos σ en lugar de σ^{-1}.

Definición 2.8 *Dados dos conjuntos ordenados de variables,* $\mathbf{X}$ *e* $\mathbf{Y}$*, tales que* $\mathbf{X} \subseteq \mathbf{Y}$ *el desplazamiento inducido de* $\mathbf{X}$ *a* $\mathbf{Y}$*, definido sobre* $\{0,\ldots,N_{\mathbf{Y}-1}\}$ *es*

$$\mathit{despl}_{\mathbf{X},\mathbf{Y}}(j) = \begin{cases} \mathit{despl}_{\mathbf{X}}(\sigma(j)) & \text{si } \sigma(j) \geq 0 \\ 0 & \text{en otro caso .} \end{cases} \tag{2.11}$$

En el ejemplo anterior, $\mathit{despl}_{\mathbf{X}} = [\underset{A}{1},\underset{B}{2},\underset{C}{4}]$ y $\mathit{despl}_{\mathbf{X},\mathbf{Y}} = [\underset{B}{2},\underset{D}{0},\underset{A}{1},\underset{C}{4}]$.

Definición 2.9 *El desplazamiento acumulado de un conjunto de variables* $\mathbf{X}$ *inducido por otro conjunto ordenado* $\mathbf{Y}$ *tal que* $\mathbf{X} \subseteq \mathbf{Y}$*, es una función* $\mathit{desplAcc}_{\mathbf{X},\mathbf{Y}} : \{0,\ldots,N_{\mathbf{Y}}-1\} \longmapsto \{-1,\ldots,N_{\mathbf{X}}-1\}$ *como sigue:*

$$\mathit{desplAcc}_{\mathbf{X},\mathbf{Y}}(j) = \mathit{despl}_{\mathbf{X},\mathbf{Y}}(j) - \sum_{j'=0}^{j-1} (|Y_{j'}|-1) \times \mathit{despl}_{\mathbf{X},\mathbf{Y}}(j') \,. \tag{2.12}$$

Por motivos de eficiencia, cuando $j > 0$ esta función se puede calcular recursivamente como sigue:

$$\mathit{desplAcc}_{\mathbf{X},\mathbf{Y}}(j) = \begin{cases} \mathit{despl}_{\mathbf{X},\mathbf{Y}}(0) & \text{si } j = 0 \\ \mathit{desplAcc}_{\mathbf{X},\mathbf{Y}}(j-1) + \mathit{despl}_{\mathbf{X},\mathbf{Y}}(j) - & \\ \quad - |Y_{j-1}| \times \mathit{despl}_{\mathbf{X},\mathbf{Y}}(j-1) & \text{si } j > 0 \,. \end{cases} \tag{2.13}$$

En el ejemplo anterior $\mathit{desplAcc}_{\mathbf{X},\mathbf{Y}} = [\underset{B}{2},\underset{D}{-2},\underset{A}{-1},\underset{C}{1}]$, como se muestra en la tabla 2.1

Teorema 2.10 *Dados dos conjuntos ordenados de variables* $\mathbf{X}$ *e* $\mathbf{Y}$*, tales que* $\mathbf{X} \subseteq \mathbf{Y}$*, la posición* $\mathbf{X}$ *de una configuración* $\mathbf{Y}$ *es la posición de la configuración previa más el desplazamiento acumulado:*

$$\mathit{pos}_{\mathbf{X}}(\mathit{sgte}(\mathbf{y})^{\downarrow \mathbf{X}}) = \mathit{pos}_{\mathbf{X}}(\mathbf{y}^{\downarrow \mathbf{X}}) + \mathit{desplAcc}_{\mathbf{X},\mathbf{Y}}(\mathit{varIncr}(\mathbf{y})) \,. \tag{2.14}$$

La demostración está en el apéndice 2.7.

Este teorema es la piedra angular del capítulo. Supongamos que tenemos un potencial $\psi_{\mathbf{X}}$ y necesitamos un nuevo potencial $\psi_{\mathbf{Y}}$, tal que $X \subseteq Y$ definido por

$$\psi_{\mathbf{Y}}(\mathbf{y}) = \psi_{\mathbf{X}}(\mathbf{y}^{\downarrow\mathbf{X}}) \,. \tag{2.15}$$

La solución "tradicional" sería calcular para cada configuración $\mathbf{y}$, su proyección $\mathbf{y}^{\downarrow\mathbf{X}}$ y luego $pos_{\mathbf{X}}(\mathbf{y}^{\downarrow\mathbf{X}})$, es decir, la posición de $\mathbf{y}^{\downarrow\mathbf{X}}$ en el array correspondiente a $\psi_{\mathbf{X}}$ mediante la ecuación 2.5. El método alternativo que describimos en este capítulo no necesita calcular explícitamente $\psi_{\mathbf{Y}}$ porque accede a sus elementos (según el orden impuesto por $\mathbf{Y}$) directamente desde $\psi_{\mathbf{X}}$. Además, no es necesario calcular cada configuración $\mathbf{y}^{\downarrow\mathbf{X}}$ y obtener su posición en $\psi_{\mathbf{X}}$ multiplicando las coordenadas por los desplazamientos: basta con aplicar secuencialmente la ecuación 2.14.

En la próxima sección compararemos la complejidad computacional de ambos métodos, el tradicional y el de los desplazamientos acumulados para calcular $\psi_{\mathbf{Y}}$ a partir de $\psi_{\mathbf{X}}$ (véase la ecuación 2.15.

2.2.4. Complejidad computacional

Tanto el método tradicional como el de los desplazamientos acumulados tienen que generar secuencialmente todas las configuraciones $\mathbf{y}$, que para ambos supone calcular $varIncr(\mathbf{y})$ y $sgte(\mathbf{y})$ para cada configuración (véase la sección 2.3). El método tradicional calcula $sgte(\mathbf{y})^{\downarrow\mathbf{X}}$ y $pos_{\mathbf{X}}(sgte(\mathbf{y})^{\downarrow\mathbf{X}})$, mientras que el método de los desplazamientos acumulados calcula $pos_{\mathbf{X}}(sgte(\mathbf{y})^{\downarrow\mathbf{X}})$ sin calcular $sgte(\mathbf{y})^{\downarrow\mathbf{X}}$. Ahora analizaremos la complejidad computacional de cada operación.

Complejidad de *varIncr*(y) y *sgte*(y)

Dada la definición 2.1, está claro que para cada configuración $\mathbf{y}$, el coste de calcular $varIncr(\mathbf{y})$ es proporcional al valor devuelto. Hay $(|Y_0|-1)\cdot|Y_1|\cdot\ldots\cdot|Y_{N_{\mathbf{Y}}-1}|$ configuraciones en las que la variable que se incrementa es la primera; definimos k como el coste (en tiempo) de calcular *varIncr* para esas configuraciones. Las configuraciones en las que se incrementa la segunda variable son $(|Y_1|-1)\cdot\ldots\cdot|Y_{N_{\mathbf{Y}}-1}|$ y el coste $2k$; y así sucesivamente. Finalmente, el coste para las $|X_{N_{\mathbf{Y}}-1}|-1$ configuraciones en las que se incrementa la última variable es aproximadamente $N_{\mathbf{Y}}\cdot k$.

Si todas las variables tienen un dominio de tamaño m, por cada j hay $(m-1)m^{N_{\mathbf{Y}}-j-1}$ configuraciones en las que la j-ésima variable se incrementa, cada una con un coste proporcional a $(j+1)$. El número total de configuraciones (excluyendo la última) es

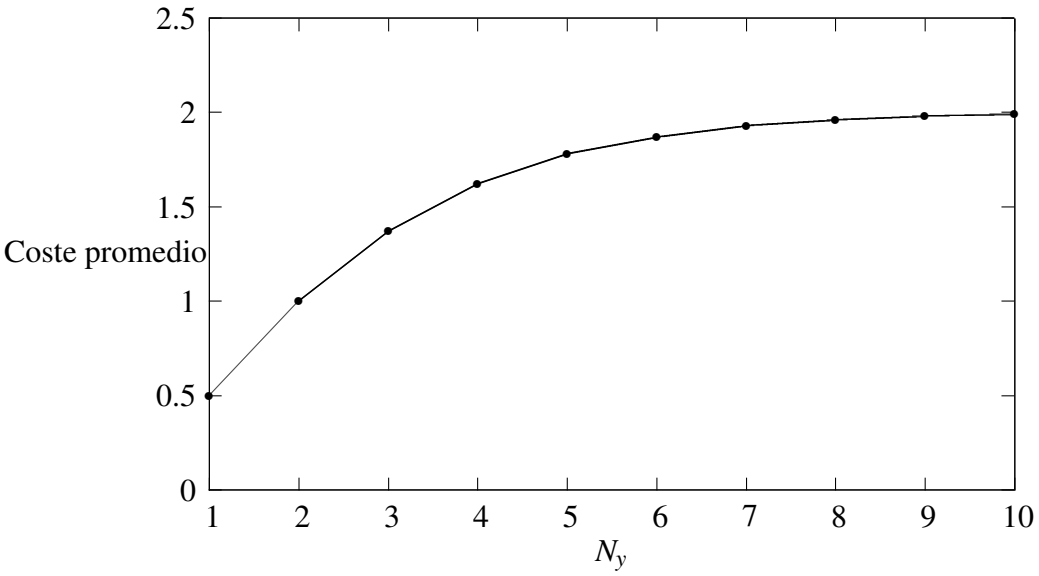


Figura 2.1: Coste promedio de *varIncr*(**y** cuando todas las variables son binarias ($m = 2$), en función del número de variables en $\mathbf{Y}$, es decir, la longitud de $\mathbf{y}$. La unidad del eje vertical es k, una medida de tiempo.

$m^{N_\mathbf{Y}} - 1$ y el coste promedio:

$$\begin{aligned} costeMedio(N_\mathbf{Y}, m) &= \frac{1}{m^{N_\mathbf{Y}} - 1} \sum_{j=0}^{N_\mathbf{Y}-1} (m-1) m^{N_\mathbf{Y}-j-1} (j+1) k \\ &= \frac{m^{N_\mathbf{Y}+1} - N_\mathbf{Y}(m-1) - m}{(m^{N_\mathbf{Y}} - 1)(m-1)} k \,. \end{aligned} \tag{2.16}$$

Tenemos que:

$$\lim_{N_\mathbf{Y} \to \infty} costeMedio(N_\mathbf{Y}, m) = \frac{m}{m-1} k \,. \tag{2.17}$$

La figura 2.2.4 representa el coste en el caso de que todas las variables sean binarias. Podemos comprobar que cuando el número de variables es grande el coste promedio es aproximadamente igual a $2k$.

El coste promedio se reduce cuanto mayores son los dominios de las variables. Cuando los dominios de las variables son distintos, el coste promedio es menor que $costeMedio(N_\mathbf{Y}, \text{mín}_j |Y_j|)$.

El coste de $next(\mathbf{y})$, definido en la ecuación 2.3, es proporcional a $varIncr(\mathbf{y})$, porque sólo las primeras variables de $next(\mathbf{y})$ cambian con respecto a las de $\mathbf{y}$.

De modo que no hay diferencia entre el método tradicional y el de los desplazamientos

acumulados-véase la tabla

Coste promedio de $\mathbf{pos_X}(\mathbf{sgte}(\mathbf{y})^{\downarrow\mathbf{X}})$

La primera diferencia entre el método tradicional y el de los desplazamientos acumulados reside en el hecho de que el método tradicional necesita calcular explícitamente $sgte(\mathbf{y})^{\downarrow\mathbf{X}}$, que se puede hacer de dos formas. La primera, basada en la definición 2.7 (ec. 2.9) consiste en fijar el valor de cada coordenada x_i en $sgte(\mathbf{y})^{\downarrow\mathbf{X}}$ como el valor de la correspondiente coordenada en $sgte(\mathbf{y})$ usando la función σ^{-1}. La otra forma de calcular $sgte(\mathbf{y})^{\downarrow\mathbf{X}}$ es partiendo de $\mathbf{y}^{\downarrow\mathbf{X}}$ (la proyección de la configuración previa) y usar σ (ec. 2.10) para actualizar únicamente las coordenadas cuyo valor ha cambiado al pasar de $\mathbf{y}$ a $sgte(\mathbf{y})$, es decir, las primeras coordenadas de $varIncr(\mathbf{y})$. La segunda forma es más eficiente que la primera porque el número de coordenadas que cambian es aproximadamente proporcional a $m/(m-1)$ —el análisis es parecido al del coste promedio de *varIncr*— mientras que la ecuación 2.9 utiliza σ^{-1} para todas las coordenadas[2] de $N_{\mathbf{X}}$.

Método	*varIncr*	$sgte(\mathbf{y})$	$sgte(\mathbf{y})^{\downarrow\mathbf{X}}$	$pos_{\mathbf{X}}$	Conjunta
tradicional, con ec. 2.9	$O\left(\frac{m}{m-1}\right)$	$O\left(\frac{m}{m-1}\right)$	$O(N_{\mathbf{X}})$	$O(N_{\mathbf{X}})$	$O(N_{\mathbf{X}})$
tradicional, con ec. 2.10	$O\left(\frac{m}{m-1}\right)$	$O\left(\frac{m}{m-1}\right)$	$O(\frac{m}{m-1})$	$O(N_{\mathbf{X}})$	$O(N_{\mathbf{X}})$
desplazamientos acumulados	$O\left(\frac{m}{m-1}\right)$	$O\left(\frac{m}{m-1}\right)$	—	1	$O\left(\frac{m}{m-1}\right)$

Tabla 2.2: Comparación del coste computacional de ambos métodos.

Una vez que tenemos $sgte(\mathbf{y})^{\downarrow\mathbf{X}}$, el tiempo necesario para calcular $pos_{\mathbf{X}}(sgte(\mathbf{y})^{\downarrow\mathbf{X}})$ es proporcional a $N_{\mathbf{X}}$. Por lo tanto, este último paso es el que más contribuye al coste computacional del método tradicional.

A diferencia del método tradicional, el método de los desplazamientos acumulados no calcula $sgte(\mathbf{y})^{\downarrow\mathbf{X}}$ y sólo necesita una suma para calcular $pos_{\mathbf{X}}(sgte(\mathbf{y})^{\downarrow\mathbf{X}})$.

Coste total

En resumen, el método tradicional con la ecuación 2.9 necesita dos operaciones cuyo coste (total) es $O(m/(m-1))$ y otras dos operaciones cuyo coste es $O(N_{\mathbf{X}})$—véase la

[2]En los experimentos descritos en las secciones 2.3.1 y 2.3.2 hemos usado la versión más eficiente del método tradicional, es decir, hemos usado la ecuación 2.10 para calcular $sgte(\mathbf{y})^{\downarrow\mathbf{X}}$ partiendo de $\mathbf{y}^{\downarrow\mathbf{X}}$, en lugar de usar la ecuación 2.9.

tabla 2.2. Cuando se usa la ecuación 2.10, se necesitan tres operaciones cuyo coste es $O(m/(m-1))$ y una cuyo coste es $O(N_{\mathbf{X}})$. El método de los desplazamientos acumulados sólo necesita dos operaciones de coste $O(m/(m-1))$.

En el caso de potenciales grandes, $N_{\mathbf{X}} \gg 2 \geq m/(m-1)$. Esto explica porque el método de los desplazamientos acumulados el más eficiente, especialmente en el caso de variables con dominios grandes.

Desde luego, nuestro método tiene el coste adicional de calcular los desplazamientos acumulados (Ecs. 2.11 y 2.13), cuyo coste es $O(N_{\mathbf{Y}})$. Sin embargo, se calculan sólo una vez, mientras que los costes mostrados en la tabla 2.2 se aplican a cada configuración de $\mathbf{Y}$ y en consecuencia hay que multiplicarlos por $\exp(N_{\mathbf{Y}})$. Por tanto el tiempo invertido en calcular los desplazamientos acumulados y el espacio usado para almacenarlos (como un vector de $N_{\mathbf{Y}}$ elementos) es insignificante en comparación con el tiempo y espacio necesarios para recuperar y operar con los valores de $\psi_{\mathbf{Y}}$.

2.3. Operaciones con desplazamientos acumulados

2.3.1. Marginalización de potenciales

Supongamos que tenemos un conjunto de variables $\mathbf{X}$, cuyas primeras variables son $\mathbf{X}_e$, el resto de las variables son $\mathbf{X}_k$ y un potencial $\psi_{\mathbf{X}}(\mathbf{x})$. Podemos necesitar un potencial marginalizado, definido por $\psi^m_{\mathbf{X}_k}(\mathbf{x}_t) = \sum_{\mathbf{x}_e} \psi_{\mathbf{X}}(\mathbf{x}_e, \mathbf{x}_t)$. Los subíndices significan "variables a eliminar" y "variables a mantener" respectivamente.

Esta operación se puede realizar anidando dos bucles for. El bucle externo itera $|\mathbf{X}^*_t|$ veces. El bucle interno itera $|\mathbf{X}^*_e|$ veces, cada una de ellas añadiendo el valor de $\psi_{\mathbf{X}}$ a una variable auxiliar `sum` e incrementando la posición en $\psi_{\mathbf{X}}$. Cuando el bucle interno termina, el bucle externo almacena el valor de `sum` en la posición actúal de $\psi^m_{\mathbf{X}_t}$ e incrementa la posición en $\psi^m_{\mathbf{X}_t}$. Este procedimiento es bastante eficiente porque no necesita incrementar las configuraciones de $\mathbf{X}$ y $\mathbf{X}_t$ ni calcular las posiciones de cada configuración en $\psi_{\mathbf{X}}$ o en $\psi^m_{\mathbf{X}_t}$; se calculan las posiciones añadiendo 1 en cada iteración—véase la ecuación 2.6.

Sin embargo, cuando las n variables que se van a marginalizar no son las n primeras variables de $\mathbf{X}$, el método de los dos bucles no puede aplicarse directamente. Una posible solución sería construir primer un nuevo conjunto $\mathbf{Y}$ reordenando las variables en $\mathbf{X}$ y un potencial $\psi_{\mathbf{Y}}(\mathbf{x}_e, \mathbf{x}_t)$, es decir, el potencial $\psi_{\mathbf{Y}}$—dado por la ecuación 2.15— es una versión reordenada de $\psi_{\mathbf{X}}$ sobre el que se puede aplicar la marginalización de los dos

bucles.

La forma tradicional de construir el potencial reordenado sería calculando $\mathbf{y}^{\downarrow \mathbf{X}}$ para cada $\mathbf{y}$ con la Ecuación 2.9 o 2.10, luego $pos_{\mathbf{X}}(\mathbf{y}^{\downarrow \mathbf{X}})$ con la Ecuación 2.6, y finalmente $sgte(\mathbf{y})$ con la ecuación 2.3. Sin embargo, la complejidad de este método es $O(|\mathbf{X}^*| \cdot N_{\mathbf{X}})$ debido a que el coste de cada configuración de $\mathbf{X}^*$ es proporcional al número de variables en $\mathbf{X}$, como se vió en la sección 2.2.4, véase la Tabla 2.2.

Una aproximación más eficiente, basada en los desplazamientos acumuados, sería calcular $\psi_{\mathbf{Y}}$ sin construir explícitamente el potencial reordenado $\psi_{\mathbf{Y}}$, sino recuperando los elementos de $\psi_{\mathbf{Y}}$ directamente de $\psi_{\mathbf{X}}$ en la ejecución del procedimiento de los dos bucles: los sucesivos valores de $\psi_{\mathbf{Y}}$ se pueden obtener aplicando la ecuación 2.14 iterativamente. Como se analizó anteriormente, la complejidad de éste método es sólo $O(|\mathbf{X}^*|)$, es decir, proporcional al número de configuraciones de las variables.

Para realizar una comparación empírica hemos aplicado ambos métodos a la marginalización de un potencial con n variables binarias—véase la tabla 2.3.[3] En estos experimentos la mitad de las variables se marginalizaron, pero el tiempo del algoritmo no depende de este parámetro. La figura 2.2 muestra que las ganancias de tiempo son mayores cuando se incrementa el tamaño del potencial. Experimentos adicionales demuestran que las ganancias de tiempo son mayores cuando mayores son los dominios de las variables, conforme a lo que se dijo en la sección 2.2.4—véase la última columna de la Tabla 2.2.

2.3.2. Multiplicación de potenciales

La multiplicación de dos potenciales , $\psi^1_{\mathbf{X}_1}$ y $\psi^2_{\mathbf{X}_2}$ definidos sobre dos conjuntos de variables, $\mathbf{X}_1$ y $\mathbf{X}_2$, respectivamente, es un nuevo potencial $\psi_{\mathbf{Y}}$ definido sobre $\mathbf{Y} = \mathbf{X}_1 \cup \mathbf{X}_2$ (el orden de las variables en $\mathbf{Y}$ puede ser escogido arbitrariamente) como sigue:

$$\psi_{\mathbf{Y}}(\mathbf{y}) = \psi^1_{\mathbf{X}_1}(\mathbf{y}^{\downarrow \mathbf{X}_1}) \times \psi^2_{\mathbf{X}_2}(\mathbf{y}^{\downarrow \mathbf{X}_2}) \,. \tag{2.18}$$

Tradicionalmente, los valores de $\psi^1_{\mathbf{X}_1}(\mathbf{y}^{\downarrow \mathbf{X}_1})$ se recuperan primero calculando la proyección $\mathbf{y}^{\downarrow \mathbf{X}_1}$ y luego su posición en el array lineal que representa $\psi^1_{\mathbf{X}_1}$. La complejidad

[3] Los algoritmos se han implementado en Java y se han ejecutado en Windows sobre un procesador AMD K7 (1.1 GHz). Los resultados para valores menores de n no se muestran porque Java no permite medir tiempos tan pequeños. Dados los recursos limitados de memoria del ordenador (1 GB), en los experimentos mostrados los valores no se recuperaron de potenciales reales almacenados en memoria, pero esto no afecta significativamente al rendimiento de los algoritmos dado que el coste principal se debe a calcular las posiciones y sumar los valores

n	t_{trad} (ms)	t_{desplAcc} (ms)	t_{trad} / t_{desplAcc}
10	0,44	0,42	1,07
12	2,61	0,86	3,03
14	9,94	1,78	5,59
16	42,5	6,47	6,58
18	191	24,4	7,87
20	798	101	7,86
22	$3{,}38\times10^{3}$	405	8,37
24	$1{,}45\times10^{4}$	$1{,}62\times10^{3}$	8,93
26	$6{,}16\times10^{4}$	$6{,}46\times10^{3}$	9,53

Tabla 2.3: Tiempo (en milisegundos) necesario para marginalizar un potencial de n variables binarias por el método tradicional y por el de los desplazamientos acumulados.

de este método es $O(|\mathbf{Y}^*|\cdot(|\mathbf{X}_1|+|\mathbf{X}_2|))$.

Una forma alternativa de multiplicar los potenciales es extenderlos a $\mathbf{Y}$ como se indica en la ecuación(2.15) y aplicar luego la expresión:

$$\psi_{\mathbf{Y}}(\mathbf{y}) = \psi^1_{\mathbf{Y}}(\mathbf{y}) \times \psi^2_{\mathbf{Y}}(\mathbf{y})\,. \tag{2.19}$$

Afortunadamente esta operación puede realizarse sin calcular explícitamente $\psi^1_{\mathbf{Y}}$ y $\psi^2_{\mathbf{Y}}$: sus valores se pueden recuperar directamente desde $\psi^1_{\mathbf{X}_1}$ y $\psi^2_{\mathbf{X}_2}$ aplicando el método de los desplazamientos acumulados, reduciendo de este modo la complejidad computacional a $O(|\mathbf{Y}^*|\cdot m/(m-1))$.

La Tabla 2.4 muestra el tiempo necesario para multiplicar dos potenciales cuando la mitad de las variables son comunes a ambos potenciales.

$2n$	t_{trad}	t_{desplAcc}	t_{trad} / t_{desplAcc}
4	$2{,}04\times10^{-2}$	$1{,}02\times10^{-2}$	1,99
6	$1{,}54\times10^{-1}$	$5{,}17\times10^{-2}$	2,97
8	1,36	0,37	3,67
10	11,8	2,93	4,01
12	104	23,4	4,43

Tabla 2.4: Tiempo (en milisegundos) necesario para multiplicar dos potenciales, cada uno definido sobre $2n$ variables binarias, n de las cuales son comunes a ambos potenciales.

Obviamente, el mismo método puede ser aplicado a la suma y división de potenciales.

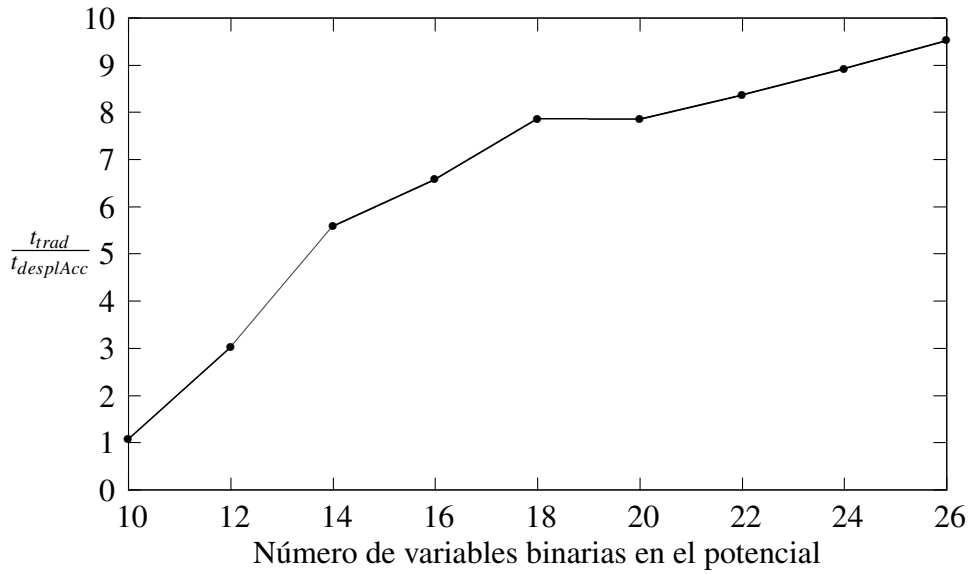


Figura 2.2: Ganancias de tiempos en la marginalización de un potencial de n variables binarias.

2.3.3. Proyección (condicionamiento)

A veces tenemos un potencial $\psi_{\mathbf{X}}$ y necesitamos operar con un potencial nuevo obtenido seleccionando algunas de las variables de $\mathbf{X}$. Por ejemplo, tenemos la evidencia $\mathbf{X}_o = \mathbf{x}_o$, donde $\mathbf{X}_o$ es el conjunto de variables observadas. El potencial nuevo, definido sobre las variables no observadas es $\mathbf{X}_u = \mathbf{X} \setminus \mathbf{X}_o$

$$\psi_{\mathbf{X}_u}^{\mathbf{x}_o}(\mathbf{x}_u) = \psi_{\mathbf{X}}(comb(\mathbf{x}_o, \mathbf{x}_u)) \,, \tag{2.20}$$

donde $comb(\mathbf{x}_o, \mathbf{x}_u)$ es la configuración de $\mathbf{X}$ consistente con $\mathbf{x}_o$ y $\mathbf{x}_u$. Por ejemplo, dado un potencial $\psi_{\mathbf{X}}(a,b,c,d)$ y la evidencia $\mathbf{x}_o = [a_0, c_1]$, el potencial nuevo sería $\psi_{\mathbf{X}_u}^{[a_0,c_1]}(b,d) = \psi_{\mathbf{X}}(a_0,b,c_1,d)$.

El tiempo necesario para obtener la configuración $comb(\mathbf{x}_o, \mathbf{x}_u)$ dado $\mathbf{x}_o$ y $\mathbf{x}_u$ es proporcional a $|\mathbf{X}|$ y el tiempo necesario para obtener la posición correspondiente utilizando la ecuación 2.6 es también proporcional a $|\mathbf{X}|$. En consecuencia, la complejidad computacional de obtener todo el potencial es $\psi_{\mathbf{X}_u}^{\mathbf{x}_o}$ is $O(|\mathbf{X}_u^*| \cdot |\mathbf{X}|)$.

Una forma más eficiente de obtener $\psi_{\mathbf{X}_u}^{\mathbf{x}_o}$ consiste en crear un potencial reordenado $\psi_{\mathbf{Y}}(\mathbf{x}_u, \mathbf{x}_o) = \psi_{\mathbf{X}}(comb(\mathbf{x}_o, \mathbf{x}_u))$, donde $\mathbf{Y} = append(\mathbf{X}_u, \mathbf{X}_o)$. En este potencial todas las configuraciones $[\mathbf{x}_u, \mathbf{x}_o]$ correspondientes a cierto $\mathbf{x}_o$ (fijo) y diferentes $\mathbf{x}_u$ ocupan

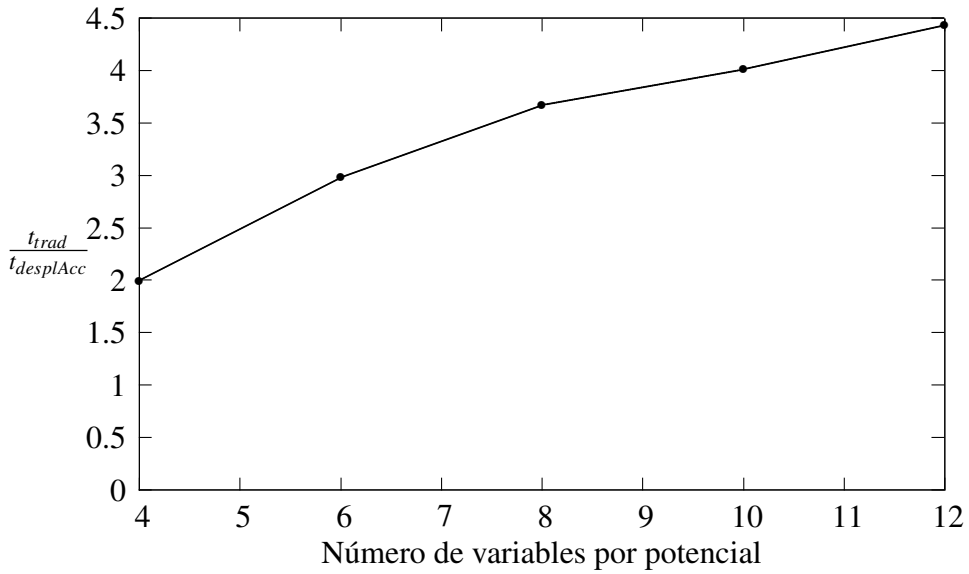


Figura 2.3: Ganancias de tiempos en la multiplicación de dos potenciales de $2n$ variables binarias, n de las cuales son comunes a ambos potenciales.

posiciones contiguas. En el ejemplo anterior, el potencial reordenado es $\psi_{\mathbf{Y}}(b,d,a,c)$, y las cuatro configuraciones del tipo $[b,d,a_0,c_1]$ ocupan las posiciones 8 a 11 (véanse las dos primeras columnas de la tabla 2.1). Consecuentemente, no es necesario calcular la posición en $\psi_{\mathbf{Y}}$ de cada configuración: es suficiente calcular la posición de la configuración $[\mathbf{x}_u = \mathbf{0}, \mathbf{x}_o]$ aplicando la ecuación 2.4 —pero sólo la primera vez— y recuperar secuencialmente $|\mathbf{X}_u^*|$ elementos de $\psi_{\mathbf{Y}}$ usando la ecuación 2.14. (En nuestro ejemplo, $[\mathbf{x}_u = \mathbf{0}, \mathbf{x}_o]$ es $[b_0,d_0,a_0,c_1]$, que ocupa la octava posición en $\psi_{\mathbf{Y}}(b,d,a,c)$). De este modo, la complejidad se reduce de $O(|\mathbf{X}_u^*| \cdot |\mathbf{X}|)$ a $O(|\mathbf{X}_u^*|)$. Evidentemente, no construiremos $\psi_{\mathbf{Y}}$, porque podemos recuperar los elementos de $\psi_{\mathbf{X}_u}^{\mathbf{x}_o}$ directamente desde $\psi_{\mathbf{X}}$.

En algunos casos será conveniente almacenar $\psi_{\mathbf{X}_u}^{\mathbf{x}_o}$ y borrar $\psi_{\mathbf{X}}$ para ahorrar memoria. En otros, en los cuales no nos interesa guardar $\psi_{\mathbf{X}}$ en la memoria de trabajo (generalmente RAM), será mejor recuperar los elementos de $\psi_{\mathbf{X}_u}^{\mathbf{x}_o}$ directamente de $\psi_{\mathbf{X}}$ al vuelo, ahorrando de este modo el espacio necesario para almacenar $\psi_{\mathbf{X}_u}^{\mathbf{x}_o}$.

La proyección de potenciales descrita en esta sección se puede usar para reducir las tablas de probabilidades de un modelo gráfico en función de la evidencia disponible. Sin embargo, esto no supone un ahorro significativo sobre el método tradicional dado que los potenciales que definen un modelo son relativamente pequeños (en otro caso no

sería factible la obtención de dichos parámetros). Una aplicación más interesante de la proyección de potenciales usando los desplazamientos acumulados son los algoritmos de condicionamiento, en los cuales los potenciales grandes se dividen en varias proyecciones que caben en la memoria disponible, reduciendo de este modo la complejidad espacial del problema (19). Cuanto mayor sea el número de variables en los potenciales originales, mayor es el ahorro.

2.4. Otras mejoras

En esta sección discutiremos dos métodos adicionales de operar con potenciales que, en principio, no están relacionados con el método de los desplazamientos acumulados y pueden ser utilizados también con el método tradicional de acceso. Sin embargo, en nuestros experimentos sólo hemos analizado el caso en que se usa el método de los desplazamientos acumulados.

2.4.1. Multiplicación simultanea de varios potenciales

La multiplicación de n potenciales de la forma $\psi^k_{\mathbf{X}_k}$ se puede definir como sigue:

$$\mathbf{Y} = \bigcup_{k=1}^{n} \mathbf{X}_k, \quad \psi_{\mathbf{Y}}(\mathbf{y}) = \prod_{k=1}^{n} \psi^k_{\mathbf{X}_k}(\mathbf{y}^{\downarrow \mathbf{X}_k}) \,. \tag{2.21}$$

La aplicación directa de esta definición se puede llamar *multiplicación por lotes*. Sin embargo, es más común multiplicar de forma *secuencial*, hacia adelante,

$$\psi_{\mathbf{Y}} = ((\psi^1_{\mathbf{X}_1} \times \psi^2_{\mathbf{X}_2}) \times \dots) \times \psi^n_{\mathbf{X}_n} \,, \tag{2.22}$$

o hacia atrás,

$$\psi_{\mathbf{Y}} = \psi^1_{\mathbf{X}_1} \times (\dots \times (\psi^{n-1}_{\mathbf{X}_{n-1}} \times \psi^n_{\mathbf{X}_n})) \,. \tag{2.23}$$

¿Qué método es más eficiente? Depende de los potenciales. Hemos analizados tres casos extremos que nos dan una idea del coste computacional de cada método.

Primer casos: todos los potenciales dependen de las mismas variables

Cuando todos los potenciales $\psi^k_{\mathbf{X}_k}$ tienen el mismo dominio, $\mathbf{Y} = \mathbf{X}_k$ para todo k. El número de multiplicaciones elementales, es decir, multiplicaciones de los valores de

los potenciales (números en coma flotante), es el mismo para multiplicación en lote y secuencial. Sin embargo, la multiplicación en lote es más eficiente porque recorre las configuraciones de $\mathbf{Y}$ sólo una vez, mientras que las multiplicaciones secuenciales lo hacen $n-1$ veces, una por cada multiplicación de dos potenciales.

La Tabla 2.5 y la Figura 2.4 muestran el tiempo necesario por ambos métodos y su ratio cuando cada potencial tiene 10 variables binarias.

$2n$	t_{trad}	t_{desplAcc}	t_{trad} / t_{desplAcc}
2	0,49	0,13	3,78
3	0,67	0,15	4,36
4	0,85	0,18	4,84
5	1,04	0,20	5,24
6	1,22	0,22	5,51
7	1,41	0,25	5,70
8	1,59	0,27	5,95
9	1,79	0,31	5,83
10	1,98	0,33	6,03
11	2,16	0,35	6,08
12	2,34	0,38	6,21
13	2,52	0,40	6,27
14	2,71	0,43	6,36
15	2,89	0,46	6,30
16	3,09	0,49	6,34
17	3,29	0,52	6,37
18	3,47	0,55	6,35

Tabla 2.5: Tiempo (en milisegundos) necesario para multiplicar n potenciales, todos con el mismo dominio.

Segundo caso: dominios anidados

Supongamos ahora que $\mathbf{X}_k = \{X_1, \ldots, X_k\}$, es decir, cada subconjunto $\mathbf{X}_k$ tiene k variables y $\mathbf{X}_k \subset \mathbf{X}_{k+1}$. También, $\mathbf{Y} = \mathbf{X}_n$. Si todas las variables son binarias, la multiplicación en lote necesita $n-1$ multiplicaciones elementales para cada configuración $\mathbf{y}$ (véase la ecuación 2.21), es decir, un total de $2^n(n-1)$, mientras que la multiplicación hacia adelante necesita sólo $\sum_{k=2}^{n} 2^k = 2^n - 1$. Sin embargo, la multiplicación hacia adelante debe pasar por todas las configuraciones de $\mathbf{X}_2$, luego por todas las de $\mathbf{X}_3$, etc., hasta las de $\mathbf{X}_n$, mientras que la multiplicación en lote sólo recorre las configuraciones correspondientes a $\mathbf{Y}$. Dado que es este caso es difícil determinar teóricamente qué

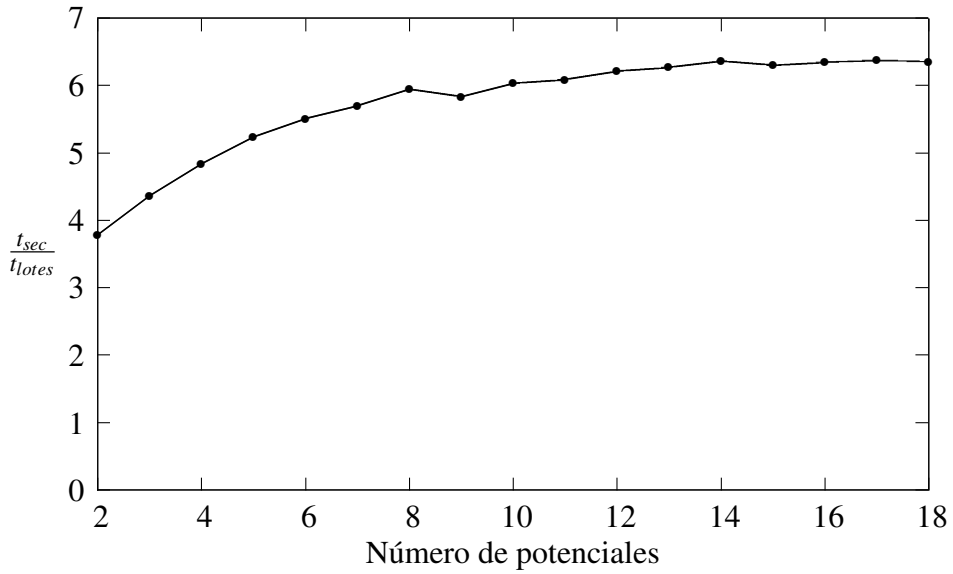


Figura 2.4: Ratios de tiempo de multiplicación en lotes y secuencial en la multiplicación de n potenciales, todos definidos en el mismo dominio (10 variables binarias).

método es más eficiente, hemos comparado empíricamente los dos mejores candidatos: multiplicación en lote y hacia adelante. (Es evidente que la multiplicación hacia atrás es el peor método en este caso porque necesita $2^n(n-1)$ multiplicaciones elementales y recorre las configuraciones de $\mathbf{X}_n$ $n-1$ veces.)

La Tabla 2.6 y la Figura 2.5 muestran el tiempo necesario por ambos métodos y su ratio.

$2n$	t_{trad}	t_{desplAcc}	t_{trad} / t_{desplAcc}
6	$4{,}66 \times 10^{-2}$	$1{,}86 \times 10^{-2}$	2,50
7	0,10	0,03	3,12
8	0,23	0,062	3,67
9	0,51	0,12	4,10
10	1,16	0,25	4,66
11	2,60	0,51	5,13
12	5,80	1,05	5,52
13	12,91	2,20	5,86
14	28,73	4,64	6,19

Tabla 2.6: Tiempo (en milisegundos) necesario para multiplicar n potenciales con dominios anidados—véanse los detalles en la Sección 2.4.1.

Cláramente se puede comprobar que la multiplicación por lotes es más eficiente, sobre

todo cuando el número de potenciales es grande

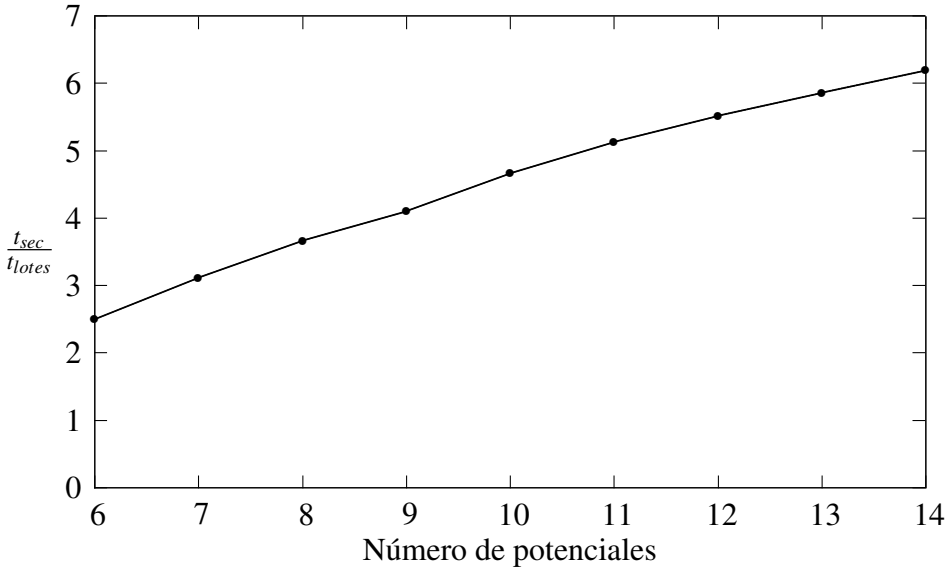


Figura 2.5: Ratios de tiempo de multiplicación por lotes y secuencial en la multiplicación de n potenciales con dominios anidados—véanse los detalles en la sección 2.4.1.

Tercer caso: todos los potenciales menos uno son constantes

En los casos anteriores la multiplicación por lotes es cláramente más eficiente que la multiplicación secuencial. Vamos a ver que esto no es siempre lo que ocurre. Analizamos el caso en el que el i-ésimo potencial $\psi^i_{\mathbf{X}_i}$, contiene un gran número de variables, 20, y los otros potenciales son constantes, es decir, no dependen de ninguna variable: $\mathbf{X}_j = \emptyset$ para $j \neq i$. Si el potencial grande es el último ($i = n$), la multiplicación hacia adelante es más eficiente que la multiplicación por lotes: ambos recorren las configuraciónes de $\mathbf{Y}$ sólo una vez, pero el último necesita muchas mas multiplicaciones elementales que el anterior. La multiplicación secuencial hacia atrás sería el método más ineficiente porque necesita tantas multiplicaciones elementales como la multiplicación en lostes y además debe recuperar las configur aciones de $\mathbf{Y}$ $n-1$ veces. Por el contrario, si el potencial grande es el primero ($i = 1$), la multiplicación hacia atrás es más eficiente que la multiplicación por lotes, y ambas superan claramente a la multiplicación hacia adelante.

En los casos en los que i está en una posición intermedia es difícil determinar analíticamente qué método es más eficiente. Por este motivo, hemos llevado a cabo algunos experimentos en los que i va desde la primera posición hasta la última. La

Tabla 2.7 y la Figura 2.6 muestran que en 7 casos de 10 la multiplicación por lotes es más eficiente, incluso en casos en los que hace más multiplicaciones. En particular, cuando $i = 6$ la multiplicación hacia adelante sólo realiza 3 multiplicaciones por configuración $\mathbf{y}$, mientras que la multiplicación por lotes hace 9 (200 % más); no obstante, la última es más eficiente que la anterior porque recorre los valores de $\mathbf{y}$ sólo una vez en lugar de tres.

Esto confirma otra vez nuestra afirmación de que es incorrecto analizar la complejidad computacional de los algoritmos de inferencia midiendo sólo el número de operaciones elementales en los valores de los potenciales.

i	t_{sec}	t_{lotes}	t_{sec} / t_{lotes}
0	952	295	3,23
1	952	292	3,25
2	832	295	2,82
3	711	292	2,43
4	593	294	2,02
5	474	292	1,62
6	358	292	1,22
7	236	304	0,78
8	117	304	0,39
9	$7{,}34 \times 10^{-2}$	303	$2{,}41 \times 10^{-4}$

Tabla 2.7: Tiempo (en milisegundos) necesario para multiplicar 10 potenciales, tales que el i-ésimo potencial depende de 20 variables y los otros son constantes, es decir, no dependen de ninguna variable.

Observación adicional: ordenación de potenciales

El caso anterior nos sugiere seguir la heurística de multiplicar primero los potenciales que dependen de menos variables. Sin embargo, es fácil ver que esta heurística falla en muchos casos. Por ejemplo, dados cuatro potenciales, $\psi^1(a)$, $\psi^2(b)$, $\psi^3(c)$, y $\psi^4(b,c)$, definidos sobre tres variables binarias, A, B, y C, la regla anterior nos lleva a $[[[\psi^1(a) \times \psi^2(b)] \times \psi^3(c)] \times \psi^4(b,c)]$, que supone $4+8+8 = 20$ multiplicaciones elementales, mientras que el orden inverso, $[[[\psi^4(b,c) \times \psi^3(c)] \times \psi^2(b)] \times \psi^1(a)]$, sólo necesita $4+4+8 = 16$. Así pues, el orden óptimo no se puede deducir de los tamaños de los potenciales—es necesario tener en cuenta sus dominios.

Dado el parecido de este problema con otros problemas de optimización, podemos suponer que encontrar el orden óptimo para la multiplicación secuencial de potenciales es

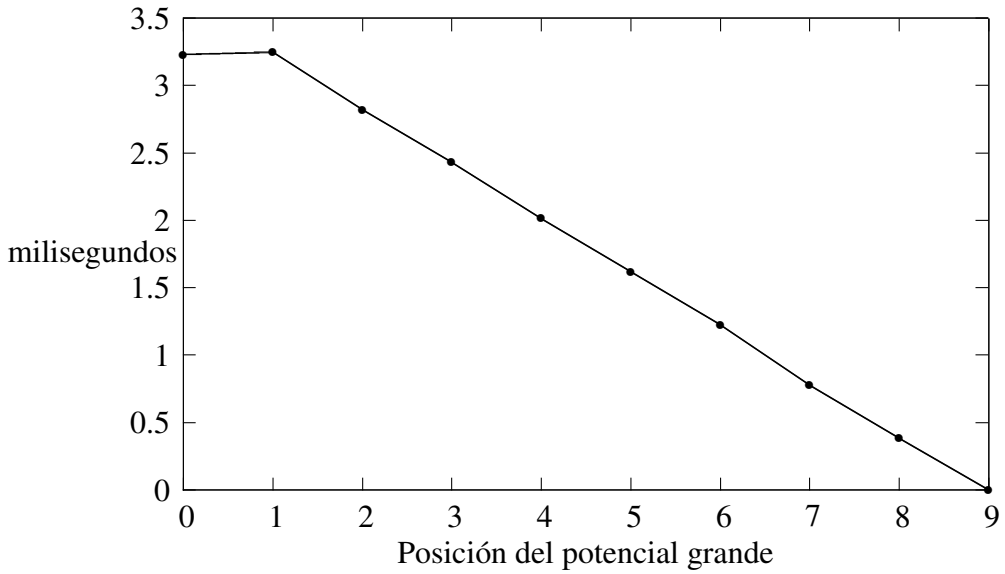


Figura 2.6: Tiempo (en milisegundos) necesario para multiplicar 10 potenciales, tales que el *i*-ésimo potencial depende de 20 variables y los otros son constantes, es decir, no dependen de ninguna variable.

NP-duro o NP-completo. Así pues, la multiplicación secuencial puede beneficiarse de un orden óptimo pero el precio es un tiempo extra para aplicar algunas heurísticas, que no pueden garantizar que el nuevo orden sea óptimo. Por el contrario, la multiplicación por lotes es indiferente al orden de los potenciales, lo que evita gastar tiempo en reordenarlos.

Conclusión

El coste de multiplicar varios potenciales es la suma de (1) el coste de recuperar sus elementos y (2) el coste de multiplicarlos. El coste (1) es la suma del coste de $\mathit{varIncr}(\mathbf{y})$ y $\mathit{sgte}(\mathbf{y})$ para cada configuración, más el coste de calcular $\mathbf{y}^{\downarrow \mathbf{X}_i}$ y $\mathit{pos}_{\mathbf{X}_i}(\mathbf{y}^{\downarrow \mathbf{X}_i})$ para cada configuración $\mathbf{y}$ y para cada potencial ψ_i. Este coste es mucho mayor para el método tradicional que para el de los desplazamientos acumulados—véase otra vez la tabla 2.2.

La multiplicación por lotes calcula $\mathit{varIncr}(\mathbf{y})$, $\mathit{sgte}(\mathbf{y})$, $\mathbf{y}^{\downarrow \mathbf{X}_i}$, y $\mathit{pos}_{\mathbf{X}_i}(\mathbf{y}^{\downarrow \mathbf{X}_i})$ sólo una vez por cada configuración y por cada potencial, reduciendo de este modo el coste (1), pero a costa de realizar frecuentemente operaciones elementales innecesarias—coste (2). Por el contrario, la multiplicación secuencial tiene a reducir el coste (2) cuando los potenciales están correctamente ordenados, pero a costa de incrementar el coste (1). Los análisis cualitativos y empíricos anteriores ponen de manifiesto que la multiplicación por lotes es casi siempre más eficiente, sobre todo cuando se usa el método de los

desplazamientos acumulados y cuando los potenciales no están correctamente ordenados. La diferencia a favor de la multiplicación por lotes sería mucho mayor usando el método tradicional de recuperación. También hemos analizado el motivo por el que ordenar los potenciales antes de multiplicar no es digno de consideración en general porque la heurística sencilla de poner los potenciales más pequeños primero está a menudo lejos de lo óptimo y las heurísticas más sofisticadas tomarían incluso más tiempo.

También hemos argumentado por qué es necesario revisar la literatura que trata la complejidad de los algoritmos exactos de inferencia en redes bayesianas porque la mayoría de los artículos publicados hasta ahora[4] sólo tienen en cuenta el número de operaciones elementales de suma y multiplicación—coste (2)—mientras que nosotros hemos demostrado que acceder a los valores de los potenciales—coste (1)—tiene más influencia en la complejidad temporal de los algoritmos.

2.4.2. Multiplicación y marginalización

La inferencia en los modelos gráficos probabilistas a menudo supone multiplicar varios potenciales y marginalizar el producto sumando o maximizando. Cuando se calcula el producto, sería deseable saber de antemano que variables serán eliminadas después, porque de este modo se podría construir $\mathbf{Y}$ (el dominio del producto—véase la ecuación 2.21) de modo que las variables a mantener, $\mathbf{Z}$, son las primeras variables en $\mathbf{Y}$. De este modo, la subsiguiente marginalización se puede llevar a cabo aplicando el método de los dos bucles descrito en la Sección 2.3.1.

Además, en lugar de calcular explícitamente el producto $\psi_{\mathbf{Y}}$, el método de los dos bucles puede multiplicar y marginalizar al mismo tiempo: el bucle externo itera sobre las configuraciones de $\mathbf{Z}$ y el interno sobre las de $\mathbf{Y}\backslash\mathbf{Z}$. Cada iteración del bucle interno calcula $\psi_{\mathbf{Y}}(\mathbf{y})$ —usando la Ecuación 2.21— para una configuración $\mathbf{y}$, y cada vez que termina una iteración, el bucle interno devuelve $\psi_{\mathbf{Z}}(\mathbf{z})$, es decir, la suma o el máximo de los $\psi_{\mathbf{Y}}(\mathbf{y})$ para los $\mathbf{y}$ compatibles con el $\mathbf{z}$ (determinado por el bucle externo). La integración de las dos operaciones ahorra tiempo porque recorre las configuraciones de $\mathbf{Y}$ sólo una vez y también ahorra espacio porque el potencial $\psi_{\mathbf{Y}}$ nunca es creado.

Hemos ejecutado un experimento en el que hemos multiplicado dos potenciales de $2n$ variables, n de las cuales eran comunes a ambos potenciales y hemos marginalizado

[4]Por ejemplo, un artículo que prueba que encontrar el árbol de cliques óptimo para redes bayesianas es NP-duro ((77)) sólo tiene en cuenta el número de sumas y multiplicaciones elementales. Otros artículos que comparan los diferentes algoritmos para modelos gráficos probabilistas sólo tienen en cuenta tales operaciones elementales.

el producto en una variable (el tiempo es independiente del subconjunto sobre el cual marginalizamos). Nuevamente hemos usado los desplazamientos acumulados. Los resultados de este experimento, resumido en la tabla 2.8 y en la figura 2.7 muestran como la integración de la multiplicación y la marginalización es mucho más eficiente que multiplicar primero y marginalizar después. La diferencia, a favor de realizar las dos operaciones simultaneamente, habría sido mayor si no hubieramos usado los desplazamientos acumulados porque cuando se marginaliza después de multiplicar es necesario iterar sobre los valores de **y** dos veces y esta operación es más costosa en el método tradicional de acceso.

$2n$	t_{trad}	t_{desplAcc}	t_{trad} / t_{desplAcc}
6	$2{,}55 \times 10^{-4}$	$5{,}88 \times 10^{-5}$	4,33
8	$2{,}26 \times 10^{-3}$	$4{,}34 \times 10^{-4}$	5,20
10	$2{,}03 \times 10^{-2}$	$3{,}35 \times 10^{-3}$	6,07
12	0,18	$2{,}71 \times 10^{-2}$	6,74
14	1,61	0,22	7,48

Tabla 2.8: Tiempo (en milisegundos) necesario para multiplicar dos potenciales de $2n$ variables, n de las cuales son comunes y marginalizar sobre una variable.

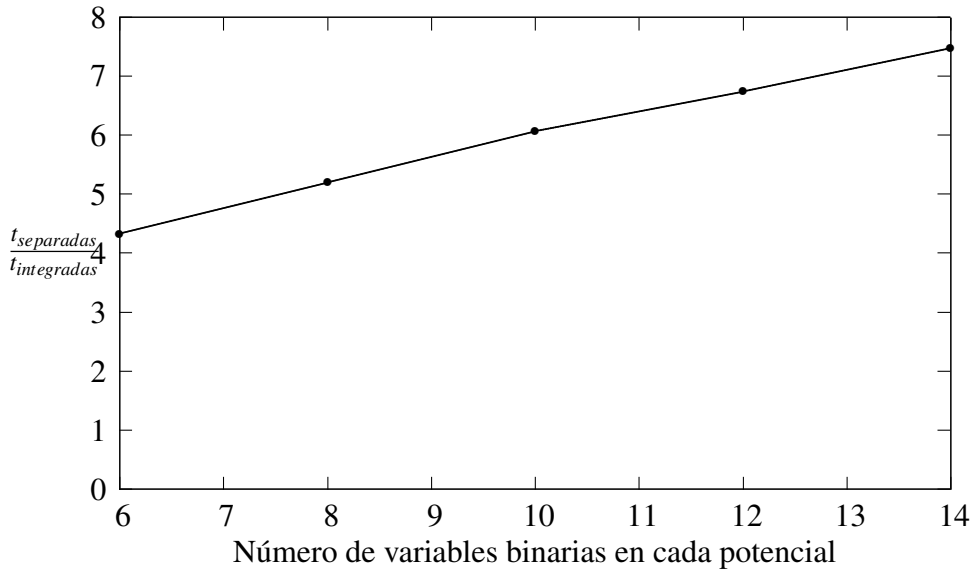


Figura 2.7: Ratio de tiempo necesario para multiplicar y marginalizar simultaneamente dos potenciales de $2n$ variables, n de la cuales son comunes, comparado con el tiempo necesario cuando se multiplican primero y se marginalizan después.

Hay que tener en cuenta que en el caso de inferencia en redes bayesianas, la integración de ambas operaciones puede hacerse sólo en algoritmos como el de Shenoy-Shafer ((68)), SPI ((45; 66)), eliminación de variables ((20)) o lazy propagation ((51)), que multiplican potenciales e inmediatamente después marginalizan el producto. Por el contrario, es más difícil, si no imposible, aplicar esta integración en algoritmos de clusterización como Lauritzen-Spiegelhalter (43) o HUGIN (36), en los cuales la multiplicación de potenciales no es seguida por una marginalización.

2.5. Trabajos relacionados e investigacíón futura

Este capítulo ha sido publicado en un artículo de revista. Uno de los revisores ha mencionado que las ideas presentadas han sido usadas en sistemas comerciales para modelos gráficos probabilistas. Sin embargo, se han guardado como secretos de empresa y nunca se han publicado.

Hay mucha literatura sobre operaciones con arrays multidimensionales, principalmente sobre la realización de operaciones algebráicas sobre matrices grandes (arrays bidimensionales)—véase por ejemplo el libro (31). A primera vista, el problema tratado en este capítulo es bastante parecido a esos problemas. De hecho, la composición de matrices es un caso particular de multiplicación de dos potenciales, cada uno definido sobre dos variables, seguido de una marginalización. Sin embargo, esa investigación trata con "potenciales" definidos sobre exáctamente dos "variables" y por esta razón el cálculo de la posición en el array no es un problema significativo: el problema reside en el gran tamaño del dominio de cada "variable", no en el número de "variables'. Análogamente, hay mucha literatura sobre imágenes en dos o tres dimensiones, que pueden ser tratados como "potenciales" definidos en dos y tres "variables", cada una con un dominio muy grande. Contrastando con lo anterior, en modelos gráficos probabilistas las variables suelen tener dominios pequeños, la mayoría son binarias y las que tienen más de cuatro o cinco estados son poco comunes. El motivo de este trabajo es reducir la carga computacional que se deriva del cálculo de las posiciones (en el array lineal) para potenciales de muchas variables.

Hay propuestas de representar los arrays multidimensionales como árboles, en los cuales cada nodo corresponde a una variable y cada rama partiendo de un nodo a un posible valor de una variable. Una de las ventajas de esta representación es que cuando todas las hojas de un subárbol tienen el mismo valor, no es necesario representar sus

ramas, basta con representar su valor. La idea se ha usado ampliamente en proceso de imagenes—véase por ejemplo las muchas publicaciones que tratan acerca de *quadtrees* ((27))—y también para representar potenciales en modelos gráficos probabilistas. Sin embargo, este método sólo es práctico cuando el potencial tiene muchas configuraciones con el mismo valor. Es más, los resultados de éste método dependen del orden de las variables en la expansión del árbol. Cuando los potenciales tienen muchos valores, como es el caso en la inferencia probabilistica, es poco probable tener un órden cercano al óptimo y reordenar un árbol es una operación costosa. Por este motivo los árboles de probabilidad sólo son provechosos en algunos casos específicos—véase por ejemplo ((12)).

Se ha propuesto una forma alternativa ((33), Sec.~8.1) para aliviar la carga computacional del cálculo de la posición en la propagación en árboles de cliques ((36)): en vez de calcular las posiciones de las configuraciones del separador para cada configuración del cluster varias veces (una por cada mensaje que se pasa), se usa una estructura que funciona como una cache, ahorrando tiempo a costa de memoria. Obviamente, el método de los desplazamientos acumulados se puede usar para construir esa estructura eficientemente. Sin embargo, debido a la eficiencia de nuestro método, construir dicha estructura sería como mínimo poco efectivo, o incluso contraproducente cuando la memoria es escasa.

Otra línea de investigación sobre operaciones con arrays de múltiples dimensiones, de los años 60 ((53)), se centra en reducir el número de veces que cada elemento de una matríz o cada pixel de una imagen se transfiere entre los diferentes niveles de memoria con sus diferentes velocidades, por ejemplo, del disco a la D-RAM (memoria principal) o de la D-RAM a la S-RAM (memoria cache). Aplicar esos conceptos a las operaciones con potenciales (marginalización, multiplicación, etc) supondría su descomposición en operaciones sobre sub-bloques de potenciales. La dificultad principal sería de nuevo que el orden de las variables en los potenciales suele ser distinto del que nos daría la máxima eficiencia. No hemos explorado esta línea de investigación porque no creemos que tenga muchas posibilidades de éxito.

2.6. Conclusiones

En este capítulo hemos mostrado que cuando se opera con potenciales de muchas variables, el coste de recuperar los elementos de los potenciales es bastante mayor

que el coste de multiplicarlos, compararlos (para maximización o minimización), o sumarlos—véase la tabla 2.2 y Sección. 2.4.1. (En contraste con los análisis que se hacen en la literatura que hay al respecto, que sólo tienen en cuenta el coste de las operaciones elementales sobre los valores de los potenciales.)

Por este motivo, hemos definido un método alternativo que recupera secuencialmente los elementos de un potencial implementado como un array lineal. En lugar de calcular cada posición multiplicando las coordenadas de cada configuración por los desplazamientos, lo que nosotros llamamos *el método tradicional*, nuestro método calcula cada posición como la posición actual más un desplazamiento acumulado. Hemos analizado teórica y empíricamente la ganancia de tiempo conseguida cuando se aplica este método en operaciones básicas, tales como suma, multiplicación, división, maximización y marginalización. En el caso de potenciales grandes, los algoritmos que usan el método de los desplazamientos acumulados son entre 5 y 10 veces más rápidos.

Además, multiplicar varios potenciales al tiempo (multiplicación en lote) es generalmente entre 2 y 5 veces más rápido que multiplicarlos secuencialmente, a condición de que usemos los desplazamientos acumulados en ambos casos. Si hubiéramos usado el método tradicional, la diferencia habría sido aún más favorable para la multiplicación en lotes—véase la sección 2.4.1.

Por último, hemos visto que integrar la multiplicación y la marginalización de potenciales en una operación sencilla es más rápido que multiplicar primero y marginalizar después. La primera opción es alrededor de 5 veces más rápido para potenciales de tamaño medio (entre 6 y 19 variables) cuando se usa el método de los desplazamientos acumulados, y la direncia sería mayor comparando con el método tradicional—véase la sección 2.4.2.

La combinación de estas tres ideas puede hacer que las operaciones con potenciales grandes sean al menos entre 100 y 500 veces más rápidas que implementadas con el método tradicional, lo que puede mejorar de forma significativa la velocidad de la inferencia en el razonamiento probabilista para problemas del mundo real.

2.7. Apéndice: Demostraciones

En este apéndice demostraremos la proposición 2.5 y el teorema 2.10, pero antes necesitamos algunas demostraciones previas.

Proposición 2.11 *Dadas dos configuraciones,* $\mathbf{x}$ *y* $\mathbf{x}'$*, tales que* $x_i \geq x'_i$ *para todo i,*

$$pos_{\mathbf{x}}(x) - pos_{\mathbf{x}}(x') = \sum_i (x_i - x'_i) \times despl_{\mathbf{x}}(i) \tag{2.24}$$

Prueba. Es una consecuencia directa de la definición 2.4 ■

Proposición 2.12 *Para todo i,* $0 \leq i < |\mathbf{X}|$,

$$despl_{\mathbf{X}}(i) = 1 + \sum_{i'=0}^{i-1} (|X_{i'}| - 1) \times despl_{\mathbf{X}}(i') \tag{2.25}$$

Prueba. Lo demostraremos por inducción. Según la definición 2.4, la ecuación 2.25 se cumple para $i = 0$. Supongamos que se cumple para $i-1$, i.e.,

$$despl_{\mathbf{X}}(i-1) = 1 + \sum_{i'=0}^{i-2} (|X_{i'}| - 1) \times despl_{\mathbf{X}}(i')$$

Y teniendo en cuenta la definición 2.4,

$$\begin{aligned} despl_{\mathbf{X}}(i) &= |X_{i-1}| \times despl_{\mathbf{X}}(i-1) \\ &= (|X_{i-1}| - 1) \times despl_{\mathbf{X}}(i-1) + despl_{\mathbf{X}}(i-1) \\ &= (|X_{i-1}| - 1) \times despl_{\mathbf{X}}(i-1) + 1 + \sum_{i'=0}^{i-2} (|X_{i'}| - 1) \times despl_{\mathbf{X}}(i') \\ &= 1 + \sum_{i'=0}^{i-1} (|X_{i'}| - 1) \times despl_{\mathbf{X}}(i') \end{aligned}$$

lo que demuestra la proposición ■

Demostración de la proposición 2.5. Sea $v = varIncr(\mathbf{x})$. Definimos una configuración de referencia $\mathbf{x}^R$ tal que

$$\begin{cases} x_i^R = 0 & \text{para } 0 \leq i < v \\ x_i^R = x_i & \text{para } v \leq i < N_{\mathbf{X}} \end{cases}$$

Según las ecuaciones 2.2, 2.3, y 2.4, se cumple

$$pos_{\mathbf{X}}(\mathbf{x}) - pos_{\mathbf{X}}(\mathbf{x}^R) = \sum_{i=0}^{v-1} (|X_i| - 1) \times despl_{\mathbf{X}}(i)$$

$$pos_{\mathbf{X}}(sgte_{\mathbf{X}}(\mathbf{x})) - pos_{\mathbf{X}}(\mathbf{x}^R) = despl_{\mathbf{X}}(v)$$

Por lo tanto,

$$pos_{\mathbf{X}}(sgte(\mathbf{x})) - pos_{\mathbf{X}}(\mathbf{x}) = despl_{\mathbf{X}}(v) - \sum_{i=0}^{v-1} (|X_i| - 1) \times despl_{\mathbf{X}}(i)$$

Por la proposición 2.12,

$$pos_{\mathbf{X}}(sgte(\mathbf{x})) - pos_{\mathbf{X}}(\mathbf{x}) = 1$$

■

Proposición 2.13 *Dadas dos configuraciones,* $\mathbf{y}$ *y* $\mathbf{y}'$*, tales que* $y_i \geq y'_i$ *para todo i,*

$$pos_{\mathbf{X}}(\mathbf{y}^{\downarrow \mathbf{X}}) - pos_{\mathbf{X}}(\mathbf{y}'^{\downarrow \mathbf{X}}) = \sum_j (y_j - y'_j) \times despl_{\mathbf{X},\mathbf{Y}}(j) \tag{2.26}$$

Prueba. Como consecuencia de la proposición 2.11 y la definición 2.9,

$$\begin{aligned} pos_{\mathbf{X}}(\mathbf{y}^{\downarrow \mathbf{X}}) - pos_{\mathbf{X}}(\mathbf{y}'^{\downarrow \mathbf{X}}) &= \sum_i (y_{\sigma^{-1}(i)} - y'_{\sigma^{-1}(i)}) \times despl_{\mathbf{X}}(i) \\ &= \sum_{j | Y_j \in \mathbf{X}} (y_j - y'_j) \times despl_{\mathbf{X}}(\sigma(j)) \end{aligned}$$

Según la definición 2.6, $Y_j \in \mathbf{X}$ es equivalente a $\sigma(j) \geq 0$. Basta con aplicar la definición 2.8 para terminar la demostración. ■

Demostración del teorema 2.10. Al igual que en las demastraciones precedentes, definimos $v = varIncr(\mathbf{y})$ y

$$\begin{cases} y^R_j = 0 & \text{para } 0 \leq j < v \\ y^R_j = y_j & \text{para } v \leq j < N_{\mathbf{Y}} \end{cases}$$

Tenemos que

$$pos_{\mathbf{X}}(\mathbf{y}^{\downarrow\mathbf{X}}) - pos_{\mathbf{X}}((\mathbf{y}^R)^{\downarrow\mathbf{X}}) = \sum_{j=0}^{v-1}(|Y_j|-1)\times despl_{\mathbf{X},\mathbf{Y}}(j)$$

$$pos_{\mathbf{X}}(sgte(\mathbf{y})^{\downarrow\mathbf{X}}) - pos_{\mathbf{X}}((\mathbf{y}^R)^{\downarrow\mathbf{X}}) = despl_{\mathbf{X},\mathbf{Y}}(v)$$

Por tanto,

$$pos_{\mathbf{X}}(sgte(\mathbf{y})^{\downarrow\mathbf{X}}) - pos_{\mathbf{X}}(\mathbf{y}^{\downarrow\mathbf{X}}) = despl_{\mathbf{X},\mathbf{Y}}(v) - \sum_{j=0}^{v-1}(|Y_j|-1)\times despl_{\mathbf{X},\mathbf{Y}}(j)$$

Por la definición 2.9,

$$pos_{\mathbf{X}}(sgte(\mathbf{y})^{\downarrow\mathbf{X}}) - pos_{\mathbf{X}}(\mathbf{y}^{\downarrow\mathbf{X}}) = desplAcc_{\mathbf{X},\mathbf{Y}}(v)$$

■

Capítulo 3

Análisis de coste-efectividad

El análisis de coste-efectividad (ACE) compara distintas opciones considerando su coste económico y su efectividad. En algunos casos, los estudios de coste-efectividad representan explícitamente variables aleatorias y los valores que pueden tomar, junto con sus respectivas probabilidades. En estos casos, el método habitual de análisis consiste en construir un árbol de decisión probabilista (véase la definición 1.25). Este método presenta dos problemas: (1) supone que hay una única decisión y (2) el tamaño de los árboles crece exponencialmente con el número de variables.

Para superar estas limitaciónes hemos desarrollado un método que permite tratar problemas de mayor tamaño con múltiples decisiones, tanto con árboles de decisión como con árboles de influencia.

La organización del capítulo es la siguiente: en la sección 3.1 ofrecemos una introducción y definimos la terminología; en la sección 3.2 desarrollaremos nuestro método, primero para árboles de decisión y, por último, expondremos el algoritmo para diagramas de influencia.

3.1. Introducción

El problema general del análisis de decisiones es que un decisor tiene que elegir una opción entre varias. Existen dos posibles formas de tomar una decisión: unicriterio, que escoge la opción que maximiza un único objetivo, o multicriterio, que escoge la opción que intenta maximizar varios objetivos estableciendo una función de compromiso. La exposición que hicimos en la sección 1.25 suponía que el análisis era unicriterio. El análisis de coste-resultados, es, en cambio, un caso particular de análisis de decisiones

bi-criterio en el que uno de los objetivos es el coste económico (que intentamos reducir) y los resultados son específicos del campo concreto; por ejemplo, en medicina, los resultados son la salud del paciente (cantidad y calidad de vida), mientras que en el caso de una intervención educativa, los resultados son la formación que adquieren los alumnos, etc. Se supone que los resultados están expresados o se han traducido a una única escala de medida, para poder hacer las comparaciones.

Cuando se realiza una intervención se obtienen unos resultados con un coste económico. El beneficio neto, por tanto, puede definirse así:

$$\text{Beneficio neto} = \text{Resultados} - \text{Coste económico} \tag{3.1}$$

El inconveniente de este indicador es que los beneficios y el coste suelen estar medidos en unidades distintas, lo cual hace que la comparación sea difícil. Por ejemplo, en sanidad es muy complicado comparar la salud con el dinero.

3.1.1. Tipos de análisis de coste-resultados

Hay varios tipos de análisis de coste-resultados, que se distinguen por la forma de medir los resultados.

a) Análisis de coste-beneficio

Consiste en asignar a cada resultado posible un valor económico. Es el análisis más sencillo, desde el punto de vista matemático, porque el coste y el beneficio de cada opción se miden con las mismas unidades. El beneficio neto es, simplemente,

$$BN = B - C \tag{3.2}$$

Se suele utilizar para seleccionar proyectos de desarrollo o inversiones. En medicina todavía se usa poco, debido a la dificultad de establecer la equivalencia entre salud y dinero, aunque en los últimos años está aumentando considerablemente el número de estudios de este tipo (24).

b) Análisis de coste-efectividad

Una forma más sofisticada de análisis consiste en definir el beneficio neto así:

$$BN = \lambda E - C \tag{3.3}$$

donde λ indica la equivalencia entre efectividad y dinero y es un parámetro que no tiene asignado un valor fijo, porque se supone que depende de cada decisor. El objetivo es elegir la intervención que maximiza la efectividad pero sin gastar excesivamente. Esto significa dar con una solución de compromiso entre efectividad y coste. En medicina, E suele expresarse en AVACs, λ indica los recursos que un individuo o un sistema de salud está dispuesto a gastar para conseguir un AVAC y C suele medirse en euros o dólares (24). Como λ es desconocida al realizar el análisis (depende del decisor), los resultados se presentan en función de λ. En la sección 3.1.3 explicamos con más detalle el método estándar de ACE.

c) Análisis de coste-utilidad

Es un caso particular del análisis de coste-efectividad, que se ha aplicado exclusivamente en el ámbito sanitario (24). Los resultados se identifican con la *utilidad*, un concepto genérico que expresa la satisfacción de un individuo con un resultado. En el caso de la sanidad, la utilidad debe tener en cuenta la calidad y la cantidad de vida. Para ello, se introduce el concepto de *esperanza de vida ajustada en calidad* (EVAC; en inglés, *quality adjusted life expectance (QALE)*); su unidad de medida es el año de vida ajustado en calidad (AVAC; en inglés *quality adjusted life years (QALY)*). Cuando la calidad de vida varía con el tiempo, $c(t)$, la EVAC correspondiente al intervalo de tiempo $[t_1, t_2]$ es:

$$\text{EVAC} = \int_{t_1}^{t_2} c(t)\, dt \tag{3.4}$$

La calidad de vida $c(t)$ se mide en una escala de 0 a 1, donde 0 corresponde a la muerte y 1 a la salud perfecta, de modo que un AVAC equivale a un año de vida con un estado de salud perfecto (o a dos años con una calidad de vida de 0.5, etc).

La calidad de vida se puede medir con modelos enumerativos, que consisten en una lista de condiciones tales como la cefalea, la ceguera, la invalidez, etc. Algunos autores como Torrance (75), se centran sólo en aspectos relacionados con la salud, mientras que

otros, como Spilker (70), tienen también en cuenta las repercusiones de dichos aspectos sobre la interacción social y el estatus económico.

3.1.2. Comparación de intervenciones en el ACE

Centrándonos en el ACE, en esta sección vamos a presentar brevemente el método estándar de comparación de varias intervenciones, que da por supuesto que el coste y la efectividad de cada una de ellas se conocen con precisión. Cada intervención puede ser muy simple; por ejemplo I_1 puede ser no hacer un test e I_2 hacerlo; en oncología I_1 puede ser no aplicar ningún tratamiento, I_2 aplicar quimioterapia e I_3 aplicar cuidados paliativos. También puede haber intervenciones complejas; por ejemplo, I_1 puede ser *"hacer el test 1, si da positivo aplicar el tratamiento T_1; si da negativo hacer el test 2, y luego..."*. Como hemos dicho, en el ACE λ no es conocido. El análisis se realiza teniendo en cuenta todos los posibles valores de λ.

Gráficamente, las intervenciones se representan en un plano en que el eje de abscisas indica la efectividad y el de ordenadas el coste (véase la figura 3.1).

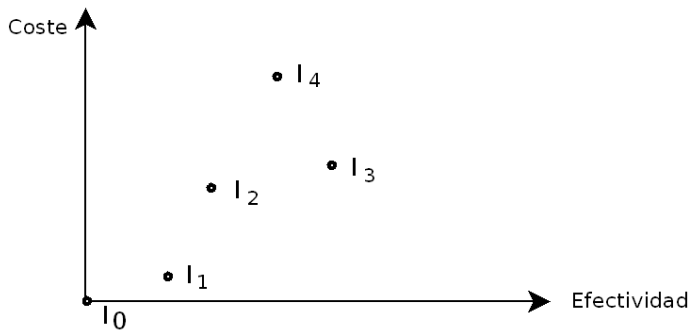


Figura 3.1: Representación de varias intervenciones.

En términos generales, las intervenciones difieren entre ellas en coste y en efectividad, y en función de λ se escogerán unas u otras. Existen, sin embargo, ciertas intervenciones que en cualquier caso van a ser peores que otras cualquiera que sea λ, y por lo tanto deben ser eliminadas de la comparativa.

a) Comparación de dos intervenciones

Sean dos intervenciones I_1 e I_2, cuyos costes son C_1 y C_2, y cuyas efectividades son E_1 y E_2 respectivamente. Vamos a considerar dos casos posibles[1]:

a.1) Dominancia simple. Se dice que una intervención I_1 domina a otra I_2 si se da una de las siguientes condiciones:

a) $C_2 > C_1$ y $E_1 > E_2$.

b) $C_2 > C_1$ y $E_1 = E_2$.

c) $C_2 = C_1$ y $E_1 > E_2$.

De cualquiera de estas tres condiciones podemos deducir que $BN_1 > BN_2$, para todo $\lambda > 0$, lo cual significa que siempre preferiremos I_1 a I_2 y por eso se dice que I_1 domina a I_2.

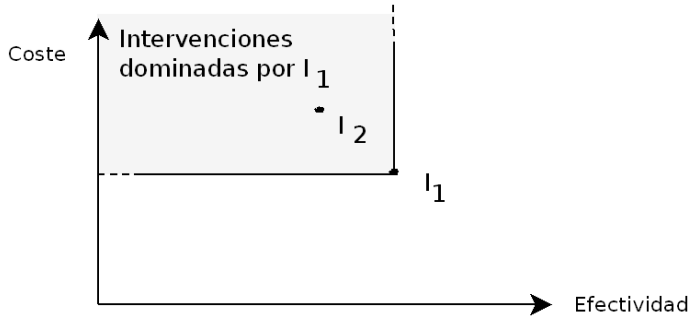


Figura 3.2: Representación gráfica de la zona del plano en la cual I_1 domina a otras opciones (por ejemplo I_2) según la definición de dominancia simple.

a.2) Razón de coste-efectividad incremental. Cuando entre dos intervenciones no hay dominancia simple, esto significa que una de ellas tiene mayor efectividad que la otra pero con un coste mayor. En este caso, una forma de compararlas, teniendo en cuenta tanto el coste como la efectividad, es calcular cuál de las dos conlleva un beneficio neto mayor. Suponemos que las intervenciones son I_1 y I_2 y que $E_2 > E_1$ y $C_2 > C_1$. Por la definición

[1] Existe un tercer caso: $C_1 = C_2$ y $E_1 = E_2$, pero no lo vamos a tener en cuenta porque en este caso no hay diferencia entre ellas.

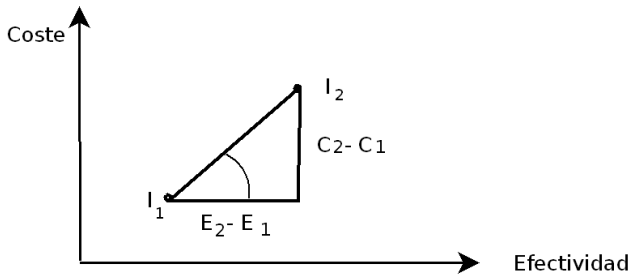


Figura 3.3: Diferencia entre la opción I_1 y la opción I_2.

de beneficio neto 3.3,

$$BN_1 = \lambda \times E_1 - C_1 \tag{3.5}$$

$$BN_2 = \lambda \times E_2 - C_2 \tag{3.6}$$

Teniendo en cuenta que $\lambda > 0$ y que $E_2 > E_1$, se deduce que:

$$\begin{aligned} BN_2 > BN_1 \Leftrightarrow \lambda \times E_2 \quad - C_2 > \lambda \times E_1 - C_1 \Leftrightarrow \lambda(E_2 - E_1) > C_2 - C_1 \Leftrightarrow \\ \Leftrightarrow \lambda > \frac{C_2 - C_1}{E_2 - E_1} \end{aligned} \tag{3.7}$$

Definición 3.1 (Razón de coste-efectividad incremental (RCEI)) *Dadas dos intervenciones I_1 e I_2 tales que $E_2 > E_1$ y $C_2 > C_1$, la razón de coste-efectividad incremental se define así:*

$$RCEI(I_1, I_2) = \frac{C_2 - C_1}{E_2 - E_1} \tag{3.8}$$

La suposición de que $\lambda > 0$ se basa en el hecho de que todo decisor está dispuesto a pagar una cantidad, aunque sea muy pequeña, para obtener una efectividad positiva.

Gráficamente, la RCEI puede interpretarse como la tangente del ángulo que forma el segmento que une los puntos I_1 y I_2 con la horizontal, tal como se muestra en la figura 3.3.

Como consecuencia de la definición 3.1 y de la ecuación 3.7 se tiene que:

$$RCEI(I_1, I_2) > \lambda \quad \Rightarrow \quad BN_1 > BN_2 \tag{3.9}$$

$$RCEI(I_1, I_2) = \lambda \quad \Rightarrow \quad BN_1 = BN_2 \tag{3.10}$$

$$RCEI(I_1, I_2) < \lambda \quad \Rightarrow \quad BN_1 < BN_2 \tag{3.11}$$

Es decir, comparando la RCEI de dos intervenciones con el valor de λ propio de cada decisor se puede determinar inmediatamente cuál es la opción preferida por ese decisor.

b) Comparación de varias intervenciones

Un problema algo más complejo consiste en seleccionar, de una lista de varias intervenciones, la de mayor beneficio neto, en función de λ. Una posible solución es aplicar el algoritmo 5, que devuelve una lista de intervalos de λ con una intervención asociada a cada uno de ellos. En esta lista no aparecen las intervenciones dominadas por otras, es decir, aquellas que nunca van a ser la opción óptima, cualquiera que sea el valor de λ.

Algoritmo 5: Selección de la intervención óptima en función de λ.

Entrada: Conjunto de intervenciones $\mathbf{I} = \{I_0 \dots I_n\}$, cada una de ellas con un coste y una efectividad.
Resultado: Subconjunto ordenado $\mathbf{J}$ de intervenciones de $\mathbf{I}$ y subconjunto de umbrales Θ.

1 // Escoger la intervención de menor coste.
2 $J_0 = \arg\min_{I_k \in \mathbf{I}} \text{coste}(I_k)$.
3 i=0
4 Eliminar de $\mathbf{I}$: J_0 y las intervenciones dominadas por J_0.
5 **Mientras** $\mathbf{I} \neq \varnothing$ **hacer**
6 //Escoger la intervención que minimice la *RCEI*.
7 $\theta_i = \min_{I_k \in \mathbf{I}} RCEI(J_i, I_k)$
8 $J_{i+1} = \arg\min_{I_k \in \mathbf{I}} RCEI(J_i, I_k)$
9 $i = i + 1$.
10 Eliminar de $\mathbf{I}$: J_i y las intervenciones dominadas por J_i.

Si el bucle se ejecuta n veces, obtendremos $n+1$ intervenciones $\{J_0, \dots, J_n\}$, y n umbrales $\{\theta_0, \dots, \theta_{n-1}\}$, que determinan $n+1$ intervalos, uno por cada intervención J_i, como indica la tabla 3.1.

En la figura 3.4 se muestran cinco intervenciones; una de ellas, I_0, corresponde a no hacer nada, por lo cual $C_0 = E_0 = 0$. Los intervalos que devolverá el algoritmo 5 son: $(0,\ RCEI(I_0, I_1))$, $(RCEI(I_0, I_1),\ RCEI(I_1, I_3))$, $(RCEI(I_1, I_3),\ +\infty)$, asignados respectivamente a las intervenciones I_0, I_1 e I_3.

Intervención	Intervalo para λ
J_0	$(0,\theta_0)$
J_1	(θ_0,θ_1)
...	...
J_{n-1}	$(\theta_{n-2},\theta_{n-1})$
J_n	$(\theta_{n-1},+\infty)$

Tabla 3.1: Intervenciones e intervalos de λ asociados.

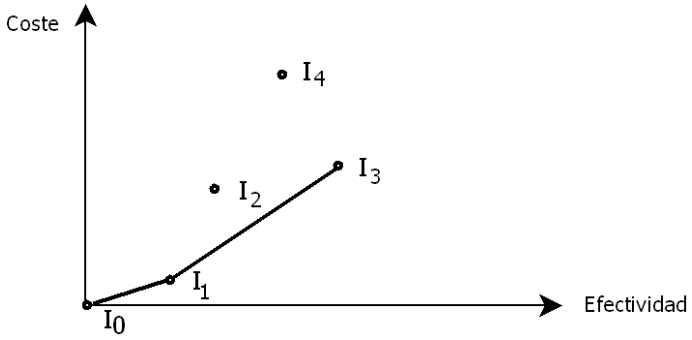


Figura 3.4: Resultado gráfico del algoritmo de comparación de varias intervenciones aplicado al ejemplo inicial de la figura 3.1.

Nótese que con este algoritmo no es necesario hacer comparaciones de tres intervenciones para detectar los casos de dominancia conjunta,[2] porque si una intervención I_2 está dominada conjuntamente por I_1 e I_3 entonces $RCEI(I_1,I_2) > RCEI(I_1,I_3)$, y por tanto, el paso 8 del algoritmo siempre va a escoger I_3 en vez de I_2.

3.1.3. Método estándar de ACE probabilista

Hemos explicado en la sección anterior como comparar varias intervenciones cuando conocemos el coste y la efectividad de cada una de ellas. Por otro lado, en la sección 1.1 explicamos como utilizar árboles de decisión para resolver problemas de toma de decisiones con incertidumbre y un solo criterio. A continuación vamos a describir el método estándar de ACE probabilista, que es una combinación de ambos.

a) Descripción del método La evaluación de los nodos de azar se realiza como en el caso de árboles de decisión unicriterio, pero computando por separado el coste y la

[2]Dadas tres intervenciones I_1, I_2 e I_3, con costes c_1, c_2 y c_3, se dice que hay *dominancia conjunta* cuando $e_1 < e_2 < e_3$, $c_1 < c_2 < c_3$ y $RCEI(I_1,I_2) > RCEI(I_1,I_3)$; véase la figura 3.4.

efectividad. Es como hacer dos evaluaciones del árbol en paralelo.

Al evaluar el nodo raíz, tenemos, por tanto, un coste y una efecividad para cada uno de sus ramas.

Como cada rama corresponde a una intervención, el análisis de coste-efectividad se realiza según el método expuesto en la sección 3.1.2.

En el método estándar, el análisis se realiza sobre un árbol de decisión con un solo nodo de decisión, situado en su raíz. Los nodos de azar se evalúan ponderando el coste y la efectividad mediante la probabilidad de cada rama. El nodo de decisión se evalúa, del mismo modo que en la sección 3.1.2, considerando cada rama como una intervención.

Ejemplo 3.1 *Sea una enfermedad E cuya prevalencia*[3] *es 0.14. Existen dos terapias posibles, t_1 y t_2. La efectividad (la cantidad y calidad de vida del paciente) depende de si tiene la enfermedad o no y de la terapia que se le aplica, tal como indica la tabla 3.2.*

Intervención	**Coste**	**Efectividad(+enf)**	**Efectividad (¬enf)**
Terapia 1	20.000 €	4,0	9,9
Terapia 2	70.000 €	6,5	9,3
No terapia	0 €	1,2	10,0

Tabla 3.2: Coste y efectividad de cada una de las tres intervenciones.

Para resolver este problema construimos el árbol de la figura 3.5. Los nodos de azar se evalúan así:

$$C_{t_i} = P(+enf) \cdot C(+enf, t_i) + P(\neg enf) \cdot C(\neg enf, t_i)$$
$$E_{t_i} = P(+enf) \cdot E(+enf, t_i) + P(\neg enf) \cdot E(\neg enf, t_i)$$

donde t_i representa cada una de las tres posibles intervenciones (t_1, t_2 o ninguna terapia) y $P(+\text{enf})$ es la prevalencia de la enfermedad, 0,14. Los resultados se muestran en la tabla 3.3 y gráficamente en la figura 3.6.

b) Limitaciones del método estándar La restricción principal de este método es que exige que haya una sola decisión y que ésta se sitúe en la raíz del árbol. Ésta es una condición muy restrictiva, pues en muchos problemas del mundo real no se cumple. Podemos verlo con un ejemplo muy sencillo, que demuestra que cuando se tienen

[3] La *prevalencia* es el porcentaje de la población que padece una determinada enfermedad.

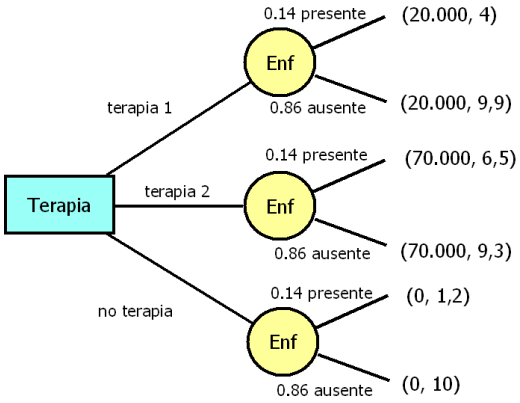


Figura 3.5: Ejemplo de árbol de decisión con una sola decisión en el nodo raíz y, (coste, efectividad) en cada hoja.

Opción	Coste	Efectividad
Terapia 1	20.000 €	9,07
Terapia 2	70.000 €	8,91
No terapia	0 €	8,78

Tabla 3.3: Costes y efectividades de las ramas del nodo de decisión de la figura 3.5, resultantes de evaluar los tres nodos de azar.

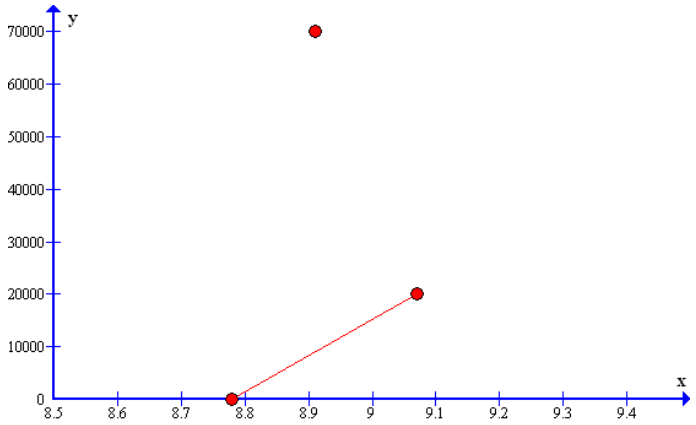


Figura 3.6: Representación gráfica del resultado de la combinación de los nodos de azar.

varias decisiones, el coste-efectividad de una decisión puede depender de las decisiones subsiguientes.

Ejemplo 3.2 *Para la enfermedad del ejemplo 3.1, si se dispone de un test con una sensibilidad*[4] *del 90 % y una especifidad del 93 % y queremos averiguar el coste y la efectividad del test.*

Para resolver este problema, podemos construir el árbol de decisión que se muestra en la figura 3.7, en la cual aparecen dos decisiones: la de hacer o no hacer el test y la de qué terapia debe aplicarse.

El problema de los nodos de decisión internos, tales como los nodos "Terapia" en el árbol de la figura 3.7, es que al evaluarlos con el algoritmo 5 (u otro equivalente) ya no se obtiene un coste y una efectividad, sino una lista de intervenciones y umbrales (o intervalos) y por eso ya no se puede continuar la evaluación hacia la izquierda, como se hacía en el método estándar que acabamos de describir. Este es el motivo por el que sólo puede haber un nodo de decisión en el árbol y debe estar en la raíz.

Soluciones:

1. Una posible solución consiste en aplicar el método estándar incluyendo tantas intervenciones como políticas posibles. En el ejemplo 3.1 las intervenciones posibles son:

   —No hacer test y no aplicar ninguna terapia.
   —No hacer test y aplicar t_1.
   —Nno hacer test y aplicar t_2.
   —Hacer test y, sea cual sea el resultado, no aplicar ninguna terapia.
   —Hacer test y, sea cual sea el resultado, aplicar t_1.
   —Hacer test y, sea cual sea el resultado, aplicar t_2.
   —Hacer test. Si es positivo no aplicar ninguna terapia, si es negativo aplicar t_1.
   —Hacer test. Si es positivo no aplicar ninguna terapia, si es negativo aplicar t_2.
   —Hacer test. Si es positivo aplicar t_1, si es negativo aplicar t_2.
   —Hacer test. Si es positivo aplicar t_1, si es negativo no aplicar ninguna terapia.
   —Hacer test. Si es positivo aplicar t_2, si es negativo aplicar t_1.
   —Hacer test. Si es positivo aplicar t_2, si es negativo no aplicar ninguna terapia.

[4] La *sensibilidad* es la probabilidad de detectar la enfermedad con el test si está presente y la *especificidad* es la probabilidad de no dar positivo cuando la enfermedad está ausente.

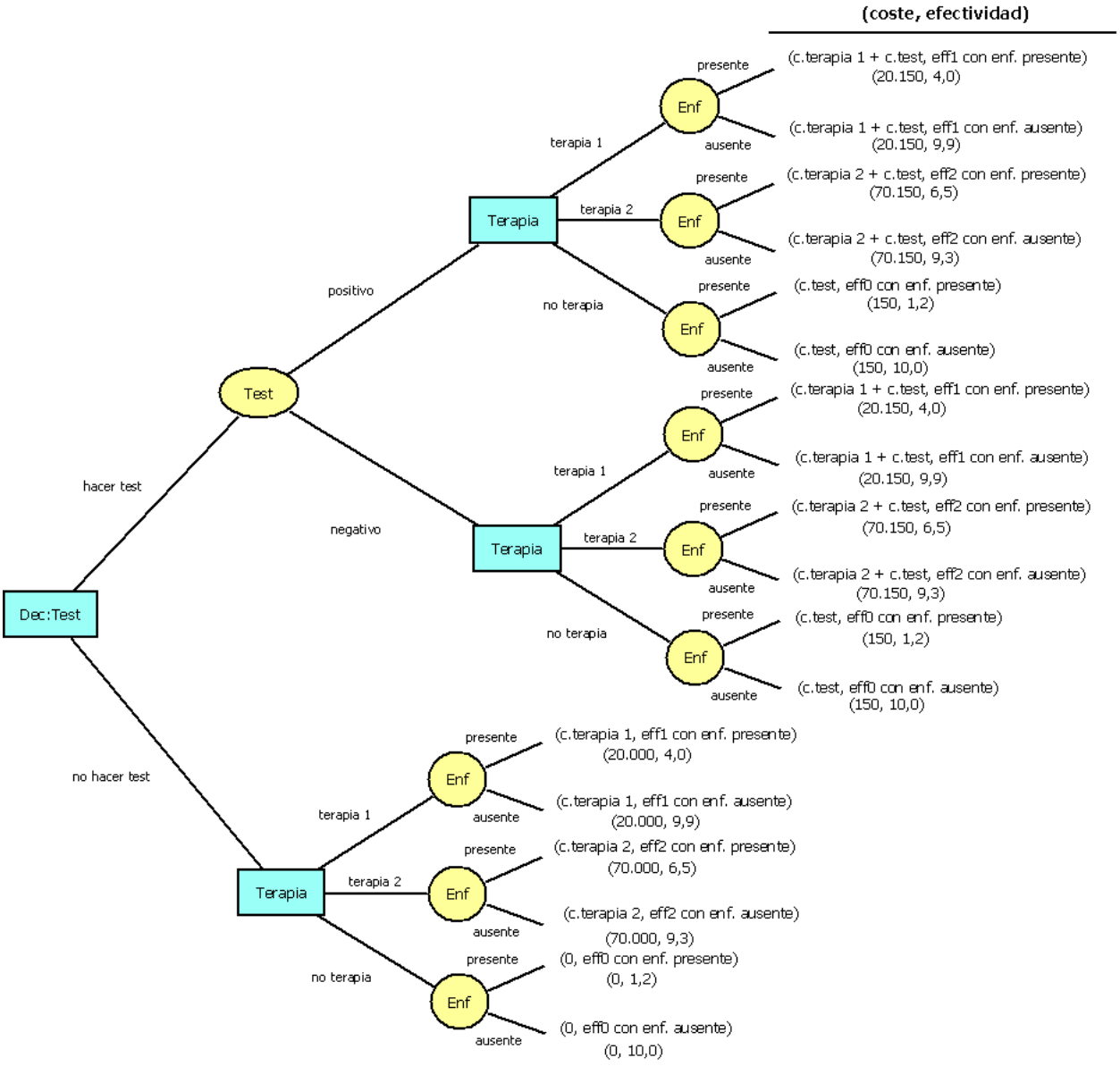


Figura 3.7: Árbol de decisión con costes y efectividades para cada rama.

Algunas de estas intervenciones se pueden rechazar por sentido común. Por ejemplo, no tiene sentido hacer un test si la decisión sobre la terapia no va a tener en cuenta el resultado del test. Tampoco tiene sentido aplicar una terapia cuando el test da negativo y no aplicarla cuando da positivo. Un inconveniente de esta solución es que el constructor del árbol tiene que decidir qué intervenciones incluye en el análisis y cuáles no. Si incluye demasiadas, la construcción del árbol y su evalución se complican considerablemete y, recíprocamente, puede ocurrir que en el intento de simplificar el problema, descarte por error la intervención óptima. Como vemos, el número de intervenciones crece super-exponenialmente con el número de decisiones. En este ejemplo, con sólo dos decisiones, tenemos doce estrategias.

2. Una "solución" incorrecta es la que ofrece el programa *TreeAge*[5]: cada nodo de decisión interno se evaúa suponiendo que el valor de λ es conocido: es el usuario el que tiene que indicar el valor de λ que se va a utilizar. Más adelante mostraremos con un ejemplo por qué esta solución no es correcta.

3. Otra solución consistiría en evaluar el diagrama para todos los valores posibles de λ. Obviamente esto es imposible porque λ puede tomar infinitos valores. Esta dificultad se puede soslayar agrupándolos por intervalos. Como no conocemos a priori cuáles son los intervalos, habría que determinarlos dinámicamente al evaluar el árbol. Ésta es la idea esencial en que se basa el nuevo método que hemos desarrollado y que exponemos a continuación.

3.2. Método para el ACE con varias decisiones

El método que hemos desarrollado puede aplicarse tanto con árboles de decisón (véase la sección 3.2.1), como con diagramas de influencia (sección 3.2.2. Veremos cada caso por separado.

[5] *TreeAge*, http://www.treeage.com/, es el programa comercial más avanzado que existe actualmente para la construcción y evaluación de árboles de decisión. Sus desarrolladores trabajan en colaboración con el *Harward Center for Risk Analysis*. El programa incorpora muchos de los avances científicos sobre análisis de decisiones a medida que se publican.

3.2.1. Árboles de decisión

a) ACE para el ejemplo 3.1

Para analizar este problema médico, nuestro método utilizaría el árbol de la figura 3.5.

En un principio, cada hoja tiene una partición formada por un solo intervalo, (0, +∞). En general, en aquellos nodos que no tienen a su derecha nodos de decisión, cada rama tiene una partición formada, al igual que en las hojas, por un único intervalo, (0, +∞); esto es debido a que los intervalos sólo se fragmentan al evaluar los nodos de decisión.

Los nodos de azar se evalúan ponderando el coste y la efectividad de sus ramas, exactamente igual que en el método estándar (sección 3.1.3):

$$C(X_i) = \sum_{ramaX_i} P(ramaX_i|x_1,\ldots,x_{i-1}) \cdot C(ramaX_i|x_1,\ldots,x_{i-1}) \tag{3.12}$$

$$E(X_i) = \sum_{ramaX_i} P(ramaX_i|x_1,\ldots,x_{i-1}) \cdot E(ramaX_i|x_1,\ldots,x_{i-1}) \tag{3.13}$$

De este modo, se obtiene un coste y una efectividad para cada nodo, que son el coste y la efectividad de la rama que lo contiene, tal como muestra la figura 3.8. La partición asociada a cada uno de estos nodos sigue teniendo un solo intervalo, (0,+∞).

Explicamos ahora cómo se evalúan los tres nodos de decisión correspondientes a la terapia. Si un nodo de decisión no tiene a su derecha otros nodos de decisión, como ocurre con los tres nodos *Terapia* de la figura 3.8, cada una de sus ramas tiene una partición formada por un solo intervalo. En este caso se realiza sobre él un análisis de coste-efectividad, como si fuera el nodo raíz del árbol en el método estándar. Por ejemplo, para el nodo *Terapia* superior, correspondiente al resultado positivo en el test, el coste y la efectividad de cada rama son los que se muestran en la figura 3.9, lo que da lugar a los intervalos mostrados en la tabla 3.4.

Intervalo	Coste	Efectividad	Mejor terapia.
(0, 10.739)	150	4,05	no aplicar terapia
(10.739, 33.384)	20.150	5,91	terapia 1
(33.384, +∞)	70.150	7,41	terapia 2

Tabla 3.4: Costes y efectividades correspondiente al nodo *Terapia* cuando el resultado del test es positivo.

Del mismo modo, la evaluación del nodo *Terapia* correspondiente al resultado negativo en el test, cuyos valores de coste y efectividad se muestran en la figura 3.10, da

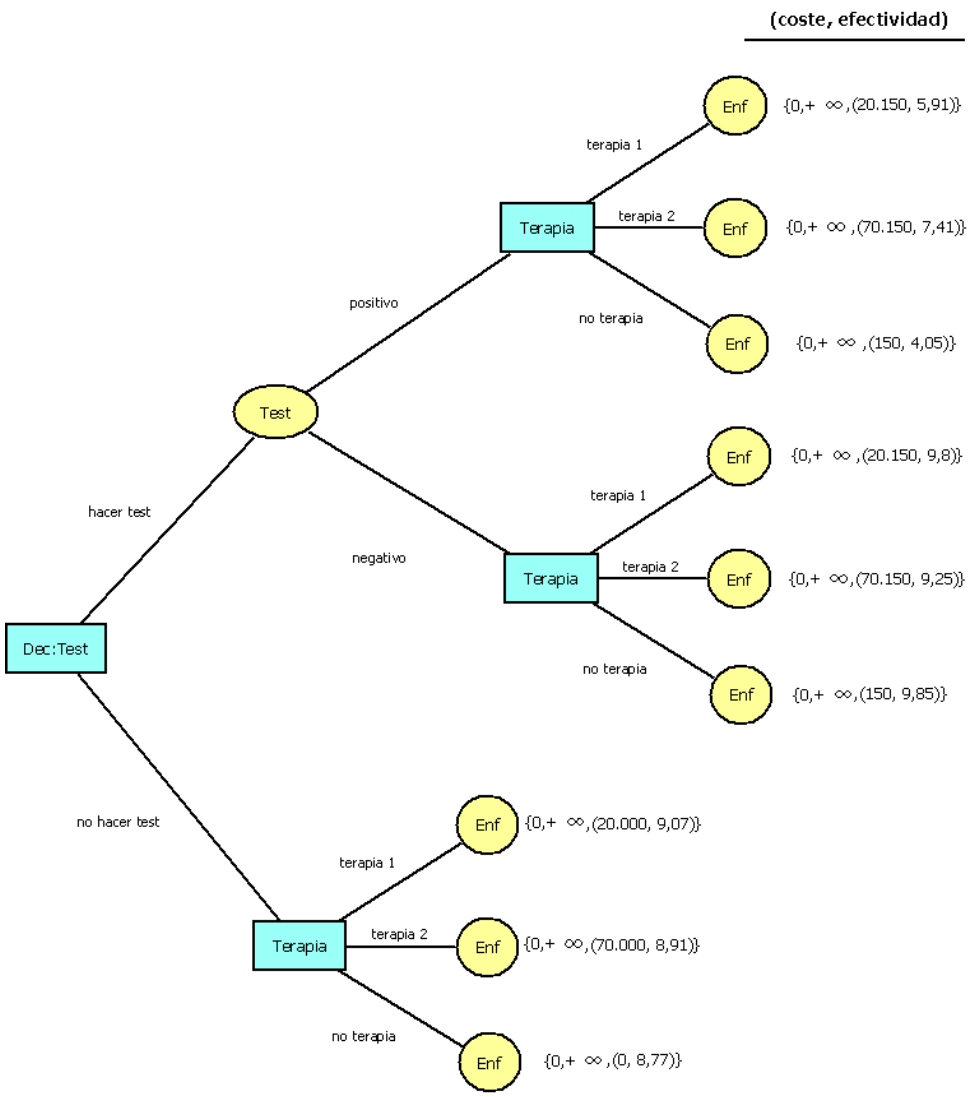


Figura 3.8: Coste y efectividad de cada nodo de azar *Enf*, resultantes de promediar las dos ramas de cada uno de ellos.

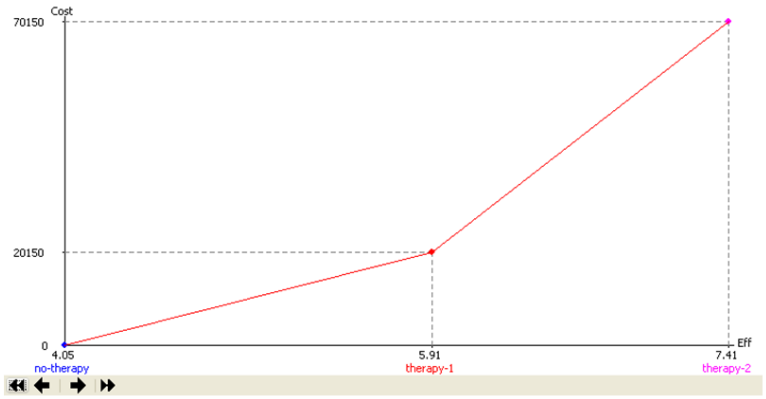


Figura 3.9: Costes y efectividades correspondientes al nodo *Terapia* cuando el resultado del test es positivo.

lugar a los intervalos mostrados en la tabla 3.5. En este caso, no se fragmenta el intervalo porque el valor *no terapia* domina a los otros dos cualquiera que sea el valor de λ. Por último, la evaluación del nodo *Terapia* correspondiente a la no realización del test, cuyos valores de coste y efectividad se muestran en la figura 3.11, da lugar a los intervalos mostrados en la tabla 3.6.

Intervalo	**Coste**	**Efectividad**	**Mejor terapia**
$(0, +\infty)$	150	9,85	no aplicar terapia

Tabla 3.5: Costes y efectividades correspondiente al nodo *Terapia* cuando el resultado del test es positivo.

Intervalo	**Coste**	**Efectividad**	**Mejor terapia**
(0, 65.539)	0	8,77	no aplicar terapia
$(65.539, +\infty)$	20.000	9,07	terapia 1

Tabla 3.6: Costes y efectividades correspondientes al nodo *Terapia* cuando no se realiza test.

La evaluación del nodo de azar *Test* plantea la dificultad de que la rama correspondiente al resultado positivo tiene una partición formada por tres subintervalos,

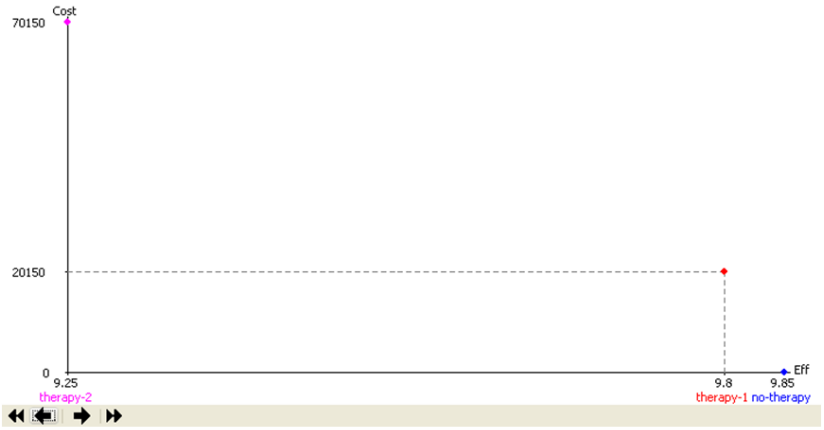


Figura 3.10: Costes y efectividades correspondientes al nodo *Terapia* cuando el test es negativo.

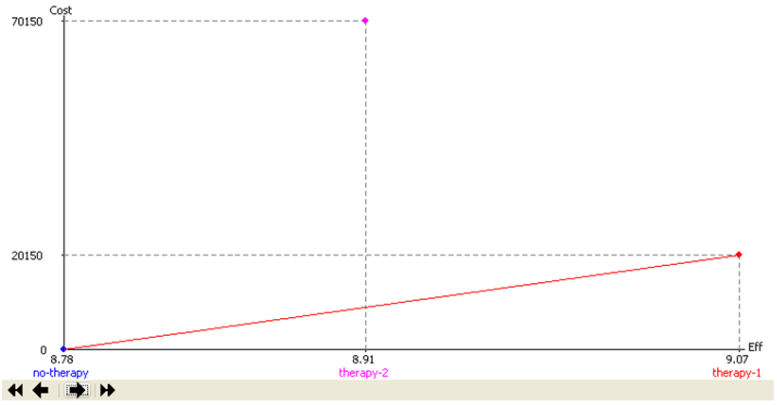


Figura 3.11: Costes y efectividades correspondientes al nodo *Terapia* cuando no se realiza test.

mientras que la otra rama tiene un solo intervalo. Este problema se puede resolver aplicando a la segunda rama la partición que obtuvimos para la primera, aunque en esta rama, el coste, la efectividad y la decisión óptima van a ser los mismos para los tres subintervalos. La combinación de ambas ramas se realiza como en el método estándar (ecuaciones 3.12 y 3.13) pero dentro de cada uno de los tres subintervalos. De este modo, la evaluación del nodo *Test* da lugar a una partición de tres subintervalos. El coste y la efectividad asociados a cada uno de ellos son los resultantes de promediar los valores de sus ramas para cada uno de estos subintervalos, de acuerdo con las ecuaciones 3.12 y 3.13.

Intervalo	**Coste**	**Efectividad**	**Mejor terapia**
(0, 10.739)	150	8,77	no aplicar terapia
(10.739, 33.384)	20.150	9,11	terapia 1
(33.384, $+\infty$)	70.150	9,39	terapia 2

Tabla 3.7: Intervalos de la rama *hacer test* en el nodo de decisión *Dec:Test*.

Por último, tenemos que evaluar el nodo de decisión *Dec:Test*. La rama *hacer test* tiene una partición formada por tres subintervalos, definidos por los umbrales 10.739 y 33.384 (tabla 3.7), mientras que la rama *no hacer test* tiene una partición formada por dos subintervalos, definidos por el umbral 65.359 (tabla 3.6). Para poder realizar el análisis de coste-efectividad en este nodo, construimos una nueva partición formada por la unión de todos los umbrales de sus ramas, $\{10{,}739, 33{,}384, 65{,}359\}$, que dan lugar a cuatro subintervalos. En cada uno de estos cuatro subintervalos, realizaremos un análisis de coste-efectividad, como en el método estándar.

En el intervalo $(0, 10{,}739)$, la opción *hacer test* tiene la misma efectividad que *no hacer test*, como se muestra en la figura 3.12, porque sea cual sea el resultado del test, no se va a aplicar ninguna terapia. Dado que la opción *hacer test* tiene un coste y no aumenta la efectividad, queda dominada por la opción *no hacer test* en todo este intervalo.

En el segundo intervalo, $(10{,}739, 33{,}384)$, el coste y la efectividad de cada opción son los que se muestran en la figura 3.13. Como ninguna opción domina a la otra, tenemos que calcular la RCEI entre ambas, que resulta ser 11.171 €/AVAC, lo cual implica que este intervalo queda dividido en dos: $(10{,}739, 11{,}171)$ y $(11{,}171, 33{,}384)$: en el primero la mejor opción es *no hacer test* y en el segundo, *hacer test*.

En el tercer intervalo, $(33{,}384, 65{,}539)$, tampoco hay una opción que domine a la otra, como se observa en la figura 3.14. La RCEI es 21.072 €/AVAC, que queda fuera del

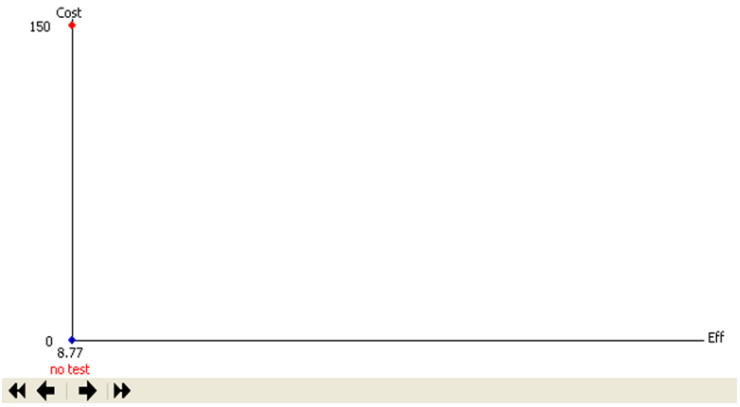


Figura 3.12: Primer intervalo: (0, 10.739).

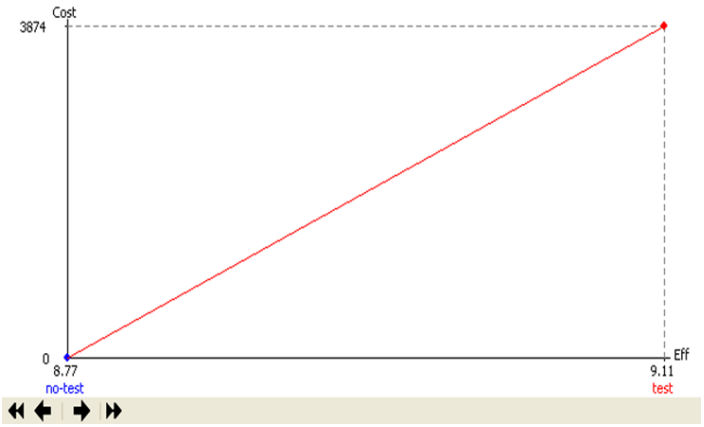


Figura 3.13: Segundo intervalo: (10.739, 33.384).

intervalo que estamos analizando. Por tanto, este intervalo no se divide. La mejor opción para este intervalo es siempre *hacer test*.

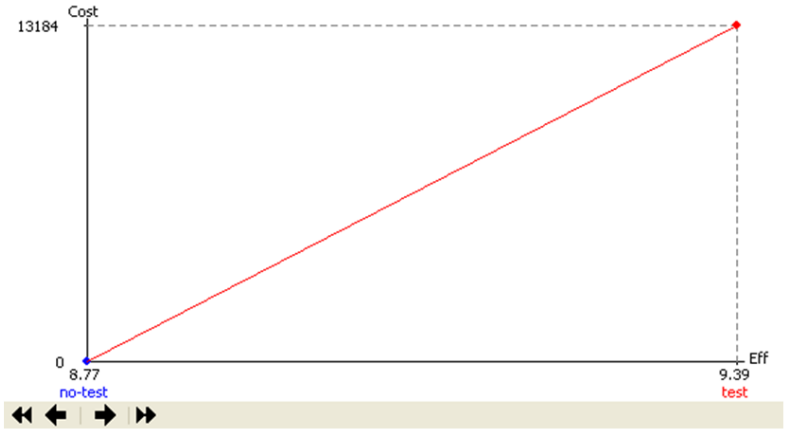


Figura 3.14: Tercer intervalo: (33.384, 65.359).

En el cuarto intervalo, $(65{,}539, +\infty)$, la opción *hacer test* domina a *no hacer test*, como puede verse en la figura 3.15. Este intervalo, por tanto, tampoco se divide.

El resultado de la evaluación del nodo *Dec:Test* se muestra en la tabla 3.8. Ahora bien, en esa tabla podemos observar que los dos primeros intervalos tienen el mismo coste, la misma efectividad y la misma terapia recomendada, por lo que podemos fusionarlos en uno solo. Lo mismo ocurre para los dos últimos intervalos. El resultado, por tanto, puede simplificarse dando lugar a la tabla 3.9, en la cual hemos incluido también la estrategia completa para cada uno de los intervalos.

Intervalos	**Coste**	**Efectividad**	**Mejor terapia**
(0, 10.739)	0	8,77	no hacer test
(10.739, 11.171)	0	8,77	no hacer test
(11.171, 33.384)	3.874	9,11	hacer test
(33.384, 65.359)	13.184	9,39	hacer test
(65.359, $+\infty$)	13.184	9,39	hacer test

Tabla 3.8: Intervalos finales de la evaluación del nodo *Dec:Test*.

Obsérvese que en las estrategias la decisión óptima para *Terapia* depende del intervalo de λ. Esto justifica nuestra afirmación de que el método de *TreeAge* es incorrecto, porque

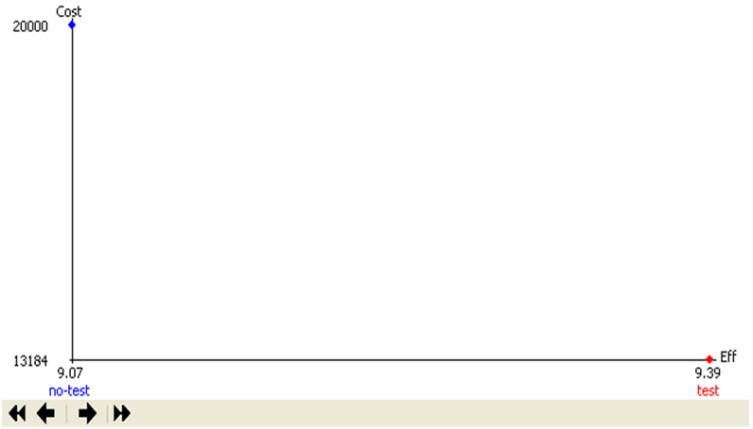


Figura 3.15: Cuarto intervalo: (65.359, $+\infty$).

en ese método la decisión óptima para *Terapia* viene dado por el valor de λ proporcionado por el usuario, que es el mismo para todos los intervalos, lo cual podría llevar a seleccionar una terapia incorrecta (es decir, distinta de la que maximiza el beneficio neto en el intervalo de interés, lo cual, a su vez llevaría a un cálculo erróneo del coste y la efectividad del *Test*, que es la primera decisión que se ha de tomar en este problema. En este ejemplo, si el usuario proporciona un valor $\lambda < 10{,}739$, al evaluar cada uno de los tres nodos *Terapia*, *TreeAge* recomendará no aplicar ninguna terapia en ningún caso y, luego, al evaluar el nodo *Test* concluirá que *no hacer test* siempre domina a *hacer test*, porque, cualquiera que sea el valor de λ, ya está decidido que no se va a aplicar ninguna terapia.

Si el usuario introduce un valor tal que $10{,}739 < \lambda < 33{,}384$, entonces *TreeAge* acertará al afirmar que cuando $\lambda < 10739$ es mejor no hacer el test y cuando $\lambda > 10{,}739$ sí conviene hacerlo, aunque se trata de un acierto por casualidad, porque ha llegado a esta conclusión suponiendo que cuando $\lambda > 10{,}739$ y el testa da positivo hay que aplicar siempre la terapia 1, lo cual es falso: cuando $\lambda > 33{,}784$ y el test da positivo la mejor terapia no es t_1 sino t_2.

En cambio, si el usuario introduce un valur $\lambda > 33{,}384$, entonces *TreeAge* concluirá que cuando el test da positivo, hay que aplicar la terapia 2, lo cual implica que la decisión de hacer el test tiene, en promedio, un coste de 13.184 € y una efectivida de 9,394 AVAC. Al comparar la opción *hacer test* con *no hacer test* se obtiene una RCEI de 21.072 €, lo cual implica que cuando $\lambda < 21{,}072$ € $/AVAC$ es mejor no hacer el test y cuando $\lambda >$

21,072€/*AVAC* conviene hacerlo y, si da positivo, hay que explicar la terapia 2. Esto es incorrecto, porque cuando 10.739<λ<33.384 la mejor intervención es hacer el testy y. si da positivo, aplicar la terapia 1. Como hemos visto, el umbral a partir del cual conviene hacer el test no es 21.072€/AVAC, sino 10.739€/AVAC.

En resumen, en este ejemplo *TreeAge* sólo determinará correctamente el umbral a partir del cual conviene hacer el test si el usuario es afortunado a la hora de indicar al programa un valor adecuado de λ, e incluso en este caso el acierto se debe hasta cierto punto a la casualidad, porque cualquiera que sea el valor de λ introducido por el usuario, *TreeAge* siempre basará su análisis en una selección de estrategias erróneas para ciertos valores de λ.

Intervalo	**Dec:Test**	**Terapia**
(0, 11.171)	no hacer test	no terapia
(11.171, 33.384)	hacer test	test positivo⟶terapia 1 test negativo⟶no terapia
(33.384, $+\infty$)	hacer test	test positivo⟶terapia 2 test negativo⟶no terapia

Tabla 3.9: Tabla simplificada fusionando los intervalos de la tabla 3.8.

Algoritmo para ACE en árboles de decisión

Partiendo del ejemplo anterior, podemos proponer el siguiente algoritmo:

1. Inicialización:
   - *a*) Construir el árbol de decisión como en el caso unicriterio.
   - *b*) Asignar a cada rama la partición $(0, +\infty)$, con el coste y la efectividad correspondientes.
2. Evaluación del árbol (de derecha a izquierda):
   - Evaluación de un nodo de azar:
     - *a*) Crear una partición definida por la unión de los umbrales de las particiones de sus ramas.
     - *b*) En cada subintervalo calcular el coste y la efectividad ponderando los costes y las efectividades de sus ramas según las ecuaciones 3.12 y 3.13.

c) Fusionar los intervalos consecutivos que tengan el mismo coste y la misma efectividad.

- Evaluación de un nodo de decisión:

  a) Crear una partición definida por la unión de los umbrales de las particiones de sus ramas.

  b) En cada subintervalo hacer un análisis de coste-efectividad, comparando las intervenciones como se indicó en la sección 3.1.2.

  c) Fusionar los intervalos consecutivos que tengan el mismo coste, la misma efectividad y la misma decisión óptima.

3.2.2. ACE con diagramas de influencia

El problema planteado en la sección anterior puede resolverse con el diagrama de influencia (DI) de la figura 3.16, el cual contiene dos nodos de utilidad: uno que representa la efectividad y otro que representa el coste económico. Dado que este problema es asimétrico (véase el árbol de la figura 3.7, en que se observa claramente que las variables que aparecen en las dos ramas del nodo *Dec:Test* son diferentes), para poder representarlo mediante un DI es necesario hacerlo simétrico, lo cual se consigue añadiendo un valor extra para el resultado del *Test*, de modo que el dominio de esta variable en el DI es {*positivo*, *negativo*, *no-realizado*}. La tabla de probabilidad para este nodo, $P(test|enf,dt)$, donde *dt* indica el valor de la variable *Dec:Test*, se muestra en la tabla 3.10. El árbol de decisión equivalente para este DI es el de la figura 3.17, que, como puede observarse, es simétrico, aunque si podáramos las ramas que tienen probabilidad = 0, obtendríamos prácticamente el árbol de la figura 3.7.

Dec:Test		*hacer test*		*no hacer test*	
Enfermedad		*+enf*	*¬enf*	*+enf*	*¬enf*
Test	*pos*	0,90	0,07	0	0
	neg	0'10	0'93	0	0
	nr	0	0	1	1

Tabla 3.10: Probabilidad del resultado del test para el diagrama de influencia de la figura 3.16. El valor *nr* significa "no realizado".

Si conociéramos el valor de λ, podríamos calcular el beneficio neto de cada uno de los escenarios (cada escenario se define por la presencia o ausencia de la enfermedad, la

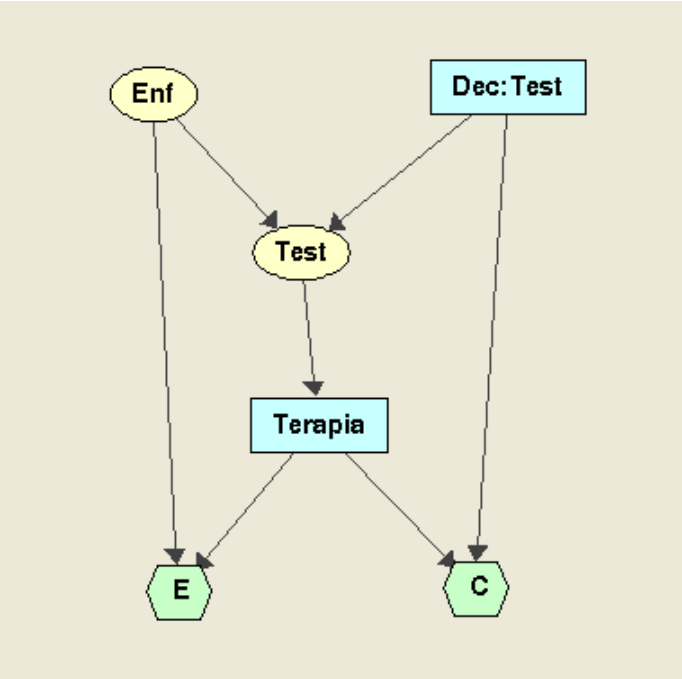


Figura 3.16: Diagrama de influencia equivalente al árbol de decisión de la figura 3.7.

realización o no del test y la terapia que hemos aplicado). De este modo, tendríamos un problema de decisión unicriterio, y el beneficio neto esperado se calcularía así:

$$BN = \max_{dt} \sum_{test} \max_{ter} \sum_{enf} P(enf) \cdot P(test|enf,dt) \cdot BN(enf,dt,ter) \tag{3.14}$$

Si consideramos $P(enf)$, $P(test|enf,dt)$ y $BN(enf,dt,ter)$ como tablas de potenciales (véase la sección 2.1) y operamos sobre ellas eliminando primero la variable *Enf* por suma, la variable *Ter* por maximización, la variable *Test* por suma y la variable *Dec:Test* por maximización (es decir, justo en el orden inverso en que aparecen en dicha ecuación), este método se denomina *eliminación de variables*.

Sin embargo, el análisis de coste-efectividad se realiza cuando no conocemos el valor de λ, y por ello el método de eliminación de variables no se puede aplicar de forma tan sencilla, sino que al eliminar cada variable de decisión hay que tener en cuenta todos los valores posibles de λ, como hemos hecho en la sección anterior para el caso de los árboles de decisión. Esta idea se puede aplicar reescribiendo la ecuación anterior así:

$$PCE = \operatorname{cea}_{dt} \sum_{test} \operatorname{cea}_{ter} \sum_{enf} P(enf) \cdot P(test|enf,dt) \cdot PCE(enf,dt,ter) \tag{3.15}$$

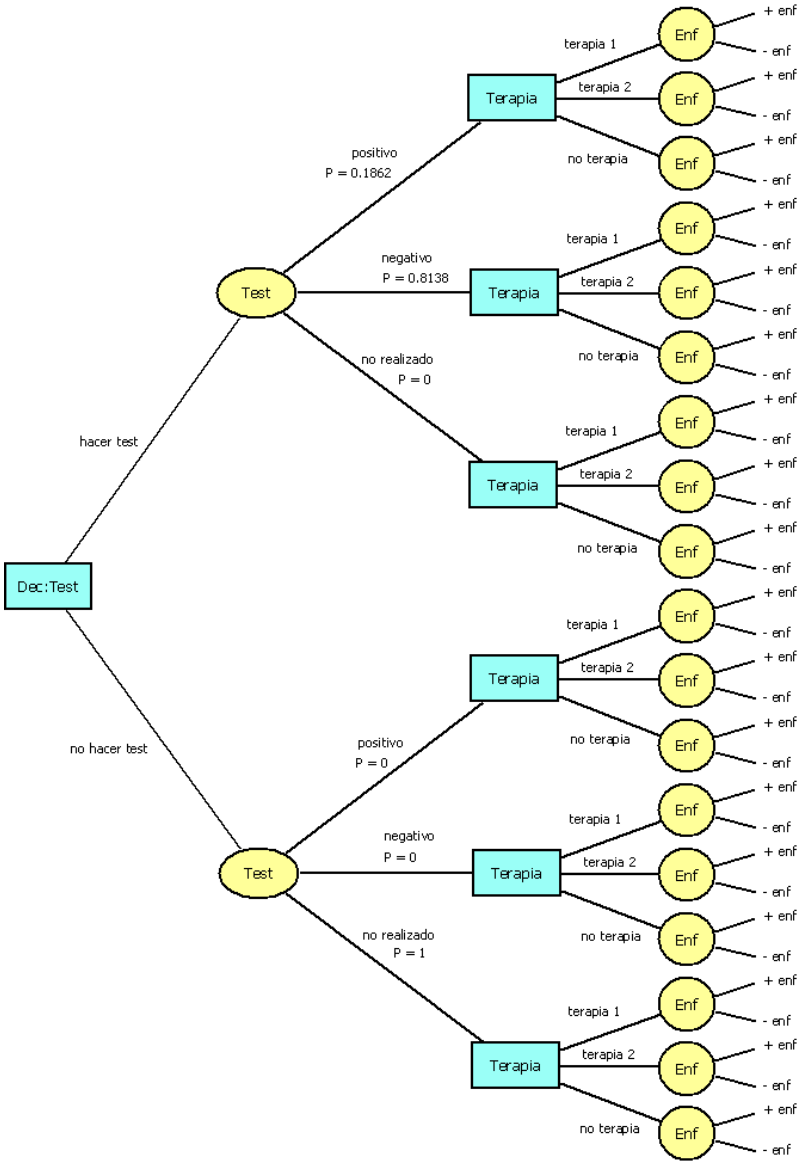


Figura 3.17: Árbol de decisión correspondiente al problema de decisión simétrico

donde *PCE* indica una partición de coste efectividad, formada por un conjunto de ternas (intervalo, coste, efectividad), de modo que los intervalos de ese conjunto forman una partición de $(0,+\infty)$. Análogamente, $PCE(enf,dt,ter)$ es una tabla que a cada configuración (enf,dt,ter) le asigna un conjunto de ternas; en este caso, cada conjunto tiene una sola terna, cuyo intervalo es $(0,+\infty)$, como indica la tabla 3.11. En esta ecuación, el signo de sumatorio no se refiere a una suma simple, sino a un operador que actúa sobre una tabla de PCE evalúando en paralelo el coste y la efectividad, y *cea* es un operador de coste-efectividad que actúa también sobre una tabla de particiones de coste-efectividad.

Enf.	**Dec:Test**	**Terapia**	**Terna**
ausente	no hacer test	no terapia	$((0,+\infty)$, 0,0, 10,0)
presente	no hacer test	no terapia	$((0,+\infty)$, 0,0, 1,2
ausente	hacer test	no terapia	$((0,+\infty)$, 150,0, 10,0)
presente	hacer test	no terapia	$((0,+\infty)$, 150,0, 1,2)
ausente	no hacer test	terapia 1	$((0,+\infty)$, 20.000,0, 9,9)
presente	no hacer test	terapia 1	$((0,+\infty)$, 20.000,0, 4,0)
ausente	hacer test	terapia 1	$((0,+\infty)$, 20.150,0, 9,9)
presente	hacer test	terapia 1	$((0,+\infty)$, 20.150,0, 4,0
ausente	no hacer test	terapia 2	$((0,+\infty)$, 70.000,0, 9,3)
presente	no hacer test	terapia 2	$((0,+\infty)$, 70.000,0, 6,5)
ausente	hacer test	terapia 2	$((0,+\infty)$, 70.150,0, 9,3)
presente	hacer test	terapia 2	$((0,+\infty)$, 70.150,0, 6,5

Tabla 3.11: *PCE(enf, dt, ter)* que contiene los valores iniciales de coste y efectividad.

b.1) Eliminación de variables sin división de potenciales Vamos a aplicar ahora el algoritmo de eliminación de variables en el orden que indican los operadores en esta ecuación. Empezamos por eliminar la variable *Enf*. Para ello, hay que multiplicar primero los dos potenciales de probabilidad, $P(enf)$ y $P(test|enf,dt)$, lo cual nos da un nuevo potencial $P(enf,test|dt)$. La multiplicación de este potencial por la tabla $PCE(enf,dt,ter)$ nos da una nueva tabla de particiones de coste-efectividad, $PCE(enf,dt,test,ter)$ en que la partición asociada a la configuración $(enf,dt,test,ter)$ tiene los mismos intervalos que en $PCE(enf,dt,ter)$ y cada valor de coste y de efectividad se multiplica por el correspondiente número real de $P(enf,test|dt)$.

Al aplicar el operador $\sum_{enf}$ obtenemos una nueva tabla de particiones de

coste-efectividad,

$$\begin{aligned} PCE(dt,test,ter) &= \sum_{enf} PCE(enf,dt,test,ter) \\ &= \sum_{enf} P(enf)\cdot P(test|enf,dt)\cdot PCE(enf,dt,ter) \end{aligned} \tag{3.16}$$

en que el valor de la configuración $(dt,test,ter)$ es una partición que combina las particiones de todas las combinaciones de $PCE(enf,dt,test,ter)$. En este caso, como sólo hay un intervalo por cada partición, que es $(0,+\infty)$, la combinación no crea nuevos umbrales: basta sumar en paralelo los costes y las efectividades.

Con esto, la ecuación 3.15 se ha reducido a

$$PCE = \underset{dt}{\text{cea}} \sum_{test} \underset{ter}{\text{cea}}\, PCE(dt,test,ter) \tag{3.17}$$

Ahora eliminamos la variable *Terapia*. Como se trata de una decisión, debemos realizar un análisis de coste efectividad para cada configuración $(dt,test)$, lo cual se realiza como en el caso de los árboles de decisión; es decir, tomamos todos las particiones de $PCE(dt,test,ter)$ que tienen los valores de dt y $test$ dados por la configuración $(dt,test)$, formamos una partición de intervalos resultante de unir los umbrales de todas las particiones que vamos a combinar, y realizamos un análisis de coste-efectividad clásico (cf. sección 3.1.3) en cada subintervalo, lo cual va a generar, a su vez, nuevos umbrales y nuevos subintervalos. Posteriormente, podemos fusionar los intervalos que tengan el mismo coste, la misma efectividad y la misma decisión óptima. De este modo, obtenemos $PCE(dt,test)$, que viene dada por

$$PCE(dt,test) = \underset{ter}{\text{cea}}\, PCE(dt,test,ter) \tag{3.18}$$

En este proceso obtenemos una política de actuación para la decisión *Terapia*, en función del valor de λ. La ecuación 3.15 queda reducida a

$$PCE = \underset{dt}{\text{cea}} \sum_{test} PCE(dt,test)$$

Luego hay que eliminar la variable de azar *Test*, lo cual nos dará una tabla de particiones de coste-efectividad,

$$PCE(dt) = \sum_{test} PCE(dt,test) \tag{3.19}$$

que se obtiene del mismo modo que para *Enf*, aunque en este caso varias de las particiones tienen umbrales generados al eliminar la decisión *Terapia*, lo cual implica que puede haber nuevas divisiones y fusiones de intervalos. La ecuación 3.15 se ha simplificado ya considerablemente, dando lugar a

$$PCE = \underset{dt}{\text{cea}}\, PCE(dt)$$

Por último, eliminamos la variable *Dec:Test*, realizando un análisis de coste-efectividad que combina las dos particiones de coste-efectividad (una para *hacer test* y otra para *no hacer test*), con lo cual llegamos a la misma partición que obtuvimos al aplicar nuestro método al árbol de decisión de la figura 3.7, es decir, a la tabla 3.9.

Sin embargo, hay que tener en cuenta un detalle: en los pasos intermedios de este algoritmo, el coste y la efectividad que aparecen en cada terna (*intervalo*, *coste*, *efectividad*) no son los mismos que se obtienen al evaluar el árbol de decisión. La razón es que en el árbol de decisión se mantienen por separado los costes, las utilidades y las probabilidades, mientras que en el proceso que acabamos de exponer, al eliminar la variable *Enf* los costes y las utilidades quedan multiplicados por las probabilidades. Si queremos obtener los costes y las efectividades reales, tenemos que introducir una modificación en este algoritmo.

b.2) Eliminación de variables con división de potenciales Como acabamos de ver, si estamos interesados en conocer los verdaderos valores de coste y de efectividad en los pasos intermedios, el método de eliminación de variables sin divisiones no nos sirve. Una posible solución consiste en aplicar el método de eliminación de variables con división de potenciales, que lleva cuenta por separado de la probabilidad y de la utilidad (en nuestro método en vez de tener una utilidad dada por un número real, tenemos una partición de coste-efectividad). La diferencia entre ambas versiones del método, con y sin divisiones, se explican en (21): las diferencias son las mismas para el caso de análisis unicriterio que para el análisis de coste-efectividad.

En nuestro ejemplo, en vez de multiplicar $PCE(enf, dt, ter)$ por el potencial de probabilidad $P(enf, test|dt)$ —resultante de multiplicar $P(enf)$ y $P(test|enf, dt)$— vamos

a dividir $P(enf, test, dt)$ en dos factores, que se calculan así:[6]

$$P(test|dt) = \sum_{enf} P(enf, test|dt) = \sum_{enf} P(enf) \cdot P(test|enf, dt) \tag{3.20}$$

$$P(enf|dt, test) = \frac{P(enf, test|dt)}{P(test|dt)} \tag{3.21}$$

El primero de ellos se conserva como el potencial de probabilidad, mientras que el segundo se multiplica por $PCE(enf, dt, ter)$. De este modo, al eliminar la variable *Enf* la partición de coste-efectividad resultante es

$$PCE(dt, test, ter) = \sum_{enf} P(enf|dt, test) \cdot PCE(enf, dt, ter) \tag{3.22}$$

y la ecuación 3.15 es equivalente a

$$PCE = \underset{dt}{\text{cea}} \sum_{test} \underset{ter}{\text{cea}}\, P(test|dt) \cdot PCE(dt, test, ter) \tag{3.23}$$

Es interesante comparar estas dos últimas ecuaciones con las ecuaciones (3.16) y (3.17) respectivamente.

El cálculo de $PCE(dt, test)$ se realiza como indica la ecuación 3.18, pero en este caso $PCE(dt, test, ter)$ es diferente que cuando no habíamos dividido los potenciales de probabilidad porque *PCE(dt, test, ta)* aún no ha sido multiplicado por *P(test|dt)*. La ecuación 3.15 se traduce en:

$$PCE = \underset{dt}{\text{cea}} \sum_{test} P(test|dt) \cdot PCE(dt, test) \tag{3.24}$$

Ahora, como ya sólo queda un potencial de probabilidad, que es la probabilidad condicionada de la variable que vamos a eliminar, no hace falta multiplicar y dividir potenciales de probabilidad: basta hacer la multiplicación que se indica en el sumatorio, y a partir de este momento la lista de potenciales de probabilidad queda vacía. Por ello, la tabla de particiones $PCE(dt)$ que se obtiene es la misma que en el método de eliminación de variables sin divisiones 3.19, porque en ambos casos $PCE(dt)$ ya ha sido multiplicado por todos los potenciales de probabilidad que había en el DI original.

[6]Este cálculo es totalmente equivalente a la inversión del arco *Enf*→*Test*, que realizaríamos para evaluar este diagrama de influencia mediante el método de inversión de arcos (59; 65): en ambos casos pasamos de tener los potenciales *P(enf)* y *P(test|enf,dt)* a tener *P(test|dt)* y *P(enf|dt,test)*.

Naturalmente, la estrategia óptima resultante es la misma para ambos métodos y para el árbol de decisión, es decir, la que se muestra en la tabla 3.9.

b) Algoritmo para ACE en diagramas de influencia

El algoritmo de eliminación de variables con divisiones puede expresarse como se indica a continuación, donde la tabla *PCE* resultante (salida) contiene una partición del intervalo $(0,+\infty)$ en varios subintervalos y, para cada uno de ellos, la estrategia óptima, su coste y su efectividad.

Entrada: un diagrama de influencia con dos nodos de utilidad, llamados "Coste" y "Efectividad".

Salida: tabla *PCE*.

1. Inicialización:

   a) Construir una lista con los potenciales de probabilidad del diagrama de influencia.

   b) Construir una tabla de particiones de coste-efectividad, *PCE*, cuyo dominio (el conjunto de variables sobre el que está definida) es la unión de los padres de los nodos de utilidad. Para cada configuración de las variables del dominio hay una terna (*intervalo*, *coste*, *efectividad*). El intervalo es $(0,+\infty)$ en todos los casos. El coste y la efectividad se obtienen de las tablas asociadas a los nodos de utilidad del diagrama de influencia.

   c) Determinar el orden de eliminación de las variables, del mismo modo que en los DI unicriterio.

2. Eliminación de las variables:

   - Eliminación de una variable de azar X:

     a) Sacar de la lista de potenciales de probabilidad todos los potenciales que dependen de X.

     b) Multiplicarlos para obtener un potencial ψ.

     c) Calcular el potencial $\psi_X = \sum_x \psi$ y devolverlo a la lista de potenciales de probabilidad.

d) Calcular el potencial $\psi' = \psi / \psi_X$, multiplicarlo por la tabla *PCE* y, en el producto resultante, sumar sobre X (es decir, para cada configuración de variables de *PCE* distintas de X, combinar todas las particiones correspondientes a distintos valores de X, del mismo modo que se combinan las ramas de un nodo de azar en un árbol de decisión). El nuevo valor de PCE será la tabla resultante de este proceso.

- Eliminación de una variable de decisión D:

  a) Si en la lista de potenciales de probabilidad hay uno o más que dependan de D, multiplicarlos y proyectar el producto sobre el resto de las variables que formaban parte del dominio de estos potenciales.[7]

  b) Hacer un análisis de coste-efectividad para D en la tabla *PCE*: sea $\mathbf{Y} = \text{dom}(PCE) \backslash \{D\}$. Para cada configuración $\mathbf{y}$, reunir todas las particiones correspondientes a diferentes valores de D, hallar la partición de invervalos resultante de combinar estas particiones, y realizar un análisis de coste-efectividad en cada subintervalo. Fusionar intervalos cuando proceda. El resultado será una nueva terna para la configuración $\mathbf{y}$, y con todas estas ternas se construye la nueva tabla *PCE*.

3.3. Discusión

El nuevo método de análisis de coste-efectividad (ACE) que hemos presentado en este capítulo permite resolver un problema de gran interés, dado que en muchos problemas del mundo real no existe una única decisión, sino una serie de decisiones que se toman secuencialmente. Sin embargo, los métodos de ACE existentes anteriormente eran incapaces de resolver problemas en que hubiera más de una decisión.

Este método puede aplicarse tanto en árboles de decisión como en diagramas de influencia. En éste último caso, nuestro algoritmo puede considerarse como un caso particular del método propuesto por Nielsen et al. (58), que permite resolver problemas de decisión multicriterio con un número de dimensiones, n, que en principio es ilimitado. En nuestro caso, al centrarnos en el análisis de coste-efectividad, n vale 2, lo cual nos permite expresar el resultado del análisis de cada decisión en forma de una partición de

[7] Cuando, al aplicar este algoritmo, vamos a eliminar una variable de decisión D y ésta aparece en uno o varios potenciales de probabilidad, entonces el producto de ellos no depende de D, como se demuestra en (47), por lo que tiene sentido proyectar este producto sobre el resto de las variables.

intervalos, separados por un conjunto de umbrales. En cambio, en el método de Nielsen et al., las regiones de decisión vienen separadas por hiperplanos en del espacio $\mathbb{R}^n$, cada uno de los cuales tiene dimensión $n-1$. En el caso del análisis de coste-efectividad, $\mathbb{R}^2$ es un plano y cada hiperplano es una recta; por eso, el método general propuesto por Nielsen et al. necesita dos parámetros para definir cada región del plano de coste-efectividad.

Sin embargo, en el análisis de decisión bi-criterio, en el cual la utilidad total puede depende de los pesos μ_1 y μ_2 utilizados,

$$U = \mu_1 \cdot C_1 + \mu_2 \cdot C_2 \tag{3.25}$$

en realidad las políticas resultantes no dependen de los valores concretos de μ_1 y μ_2, sino sólo de la proporción entre ellos. Por ello, todas las rectas resultantes contiene el origen de coordenadas, (0,0), y la única propiedad de la recta que nos interesa es su pendiente (un parámetro en vez de dos). En el caso del análisis de coste efectividad las pendientes de las rectas resultantes del análisis son precisamente los umbrales que separan los subintervalos en nuestro método.

Ésta es la razón por la que la implementación de nuestro método es mucho más sencilla que en el caso general (al parecer, el método de Nielsen et al. nunca ha sido implementado) y los resultados son mucho más fáciles de interpretar: la partición de intervalos tiene un sentido muy intuitivo: cada umbral corresponde a un valor de λ, es decir, a una razón de coste-efectividad; por ejemplo, en medicina suele considerarse que λ vale 50,000 dólares/AVAC en Estados Unidos, 30,000 libras/AVAC en Reino Unido, 30,000 euros/AVAC en España, etc. En cambio, la división del plano $\mathcal{R}^2$, donde las abscisas representan el valor de μ_1 y las ordenadas el de μ_2, de coste-efectividad mediante un conjunto de rectas no es fácil de interpretar para los profesionales de la sanidad.

Nuestro método ya está implementado como un componente (*plugin*) de la herramienta Carmen, descrita en la III de esta memoria y pensamos que va a tener una gran aceptación en la comunidad médica, pues permitirá realizar análisis de coste-efectividad en problemas que hasta ahora eran intratables.

Parte III

Herramienta Carmen

Capítulo 4

Visión general de la herramienta Carmen

Carmen es un entorno de desarrollo para MGPs. La principal diferencia de Carmen respecto de otros proyectos es el especial énfasis que se ha puesto en seguir los principios de la ingeniería del software. Las ventajas más importantes de estos principios son que al estar los proyectos documentados y tener unas especificaciones y un diseño cuidadoso, tienen menos errores, suelen ser más eficientes, son más mantenibles y como consecuencia, es fácil para un nuevo miembro del grupo empezar a hacer aportaciones en un plazo más breve.

El capítulo está organizado como sigue: en la sección 4.1 exponemos el contexto de la herramienta, los objetivos y la metodología seguida; en la sección 4.2 exponemos en detalle la especificación de requisitos y, por último, en la sección 4.3 explicamos la arquitectura de Carmen.

4.1. Introducción

La herramienta Carmen está formada por un núcleo de código fuente y la documentación del proyecto. El código fuente se compone de la definición de las estructuras de datos principales, un conjunto de algoritmos de inferencia para dichas estructuras, la interfaz gráfica y un conjunto de pruebas que tratan de eliminar en la medida de lo posible la existencia de errores[1]. La documentación del proyecto está formada por las especificaciones, la descripción de la arquitectura, el diseño detallado de cada parte y una normativa acerca de cómo se tienen que hacer las sucesivas aportaciones

[1]No es posible reducir a cero los errores de un programa porque sería necesario hacer pruebas exhaustivas, que crecen en número de forma exponencial con el tamaño del programa.

a Carmen. Las aportaciones del capítulo 2 también han sido incorporadas a la línea base[2].

El objetivo principal del proyecto Carmen es construir una plataforma de software libre que sirva como: 1) banco de pruebas para implementar y probar nuevos MGPs y nuevos algoritmos, 2) entorno para el desarrollo y la evaluación de modelos que resuelvan problemas del mundo real (medicina, ingeniería, etc), y 3) implantación de dichos modelos en un entorno de producción para ser explotada por usuarios finales.

4.1.1. Otras herramientas de código abierto para MGPs

Los programas informáticos para construir sistemas expertos se pueden clasificar en dos tipos: de código abierto, que permiten que se modifique líbremente para poder añadir funcionalidades, y convencionales o de código cerrado, que no lo permiten. Las primeras fabricados generalmente por uno o varios equipos de investigación y su finalidad es servir de base para el desarrollo de nuevos modelos y algoritmos. Ejemplos de este primer tipo son Elvira (26) y Weka (78). El segundo tipo de herramientas suelen ser comerciales y en cualquier caso, los usuarios sólo pueden desarrollar sistemas expertos con técnicas ya conocidas, porque sólo el fabricante puede añadir nuevos algoritmos.

BNT y sus sucesores

BNT (Bayes Net Toolbox) fue construida por Kevin Murphy (2001) en MATLAB (MATrix LABoratory, "*laboratorio de matrices*"), un entorno orientado a operaciones numéricas con matrices que incluye un lenguaje de programación específico. Las razones de su éxito fueron los muchos modelos implementados en BNT (redes bayesianas, diagramas de influencia, modelos dinámicos, aprendizaje e inferencia con variables discretas y continuas...), su robustez, la claridad de su código y su documentación.

Las principales limitaciones de BNT se deben a su dependencia de MATLAB, que es un programa bastante caro (aunque existe una licencia para estudiantes), es lento en comparación con otros lenguajes de programación y no está diseñado para la programación orientada a objetos.

[2]Según el IEEE 610.12/1990, se define como una especificación o producto que se ha revisado formalmente y sobre los que se ha llegado a un acuerdo, y que de ahí en adelante sirve como base para un desarrollo posterior y que puede cambiarse solamente a través de procedimientos formales de control de cambios.

Debido a estas limitaciones, ha habido varios intentos posteriores de construir un programa en un lenguaje más eficiente, pero ninguno de los intentos han conseguido reemplazar a BNT. El proyecto **Open Bayes**, iniciado por R. Dybowski y K. Murphy en 2001, se abandonó algunos meses después sin ni siquiera haber decidido qué lenguaje de programación se iba a utilizar. En 2005, Intel Labs en Rusia, con la colaboración de K. Murphy, liberó **PNL** (Probabilistic Network Library), una librería de código libre en C++ para MGPs. El proyecto se abandonó ese mismo año. De igual modo, el proyecto **Open Bayes for Python (OBP)** se paró antes de la liberación de la versión 0.2, que estaba prevista en febrero de 2007.

gR

R es un lenguaje para cálculos estadísticos y representaciones gráficas, véase http://www.r-project.org/. Es una evolución de un lenguaje previo, S. gR surgió como un subproyecto de R orientado a los MGPs. El desarrollo de gR se paró en 2003.

Weka

Este proyecto comenzó en 1993 en la Universidad de Waikato, en Nueva Zelanda, con el objetivo de incluir varios métodos de minería de datos en una única herramienta. Los clasificadores bayesianos fueron implementados por Bouckaert en 2004 (9).

Las principales ventajas de Weka son que tiene una buena documentación, que facilita el desarrollo de nuevos algoritmos de aprendizaje y, la posibilidad de comparar empíricamente muchos métodos, no solo los que aprenden redes bayesianas, sino también "algoritmos tradicionales", como C4.5, K-NN, etc. La limitación es que no es de propósito general.

Elvira

El punto de partida del proyecto Carmen es el proyecto Elvira. El programa Elvira es el fruto de un proyecto coordinado de I+D financiado por la CICYT (Comisión Interministerial de Ciencia y Tecnología), que se desarrolló entre los años 1997 y 2000, con la participación de 25 profesores de 8 universidades españolas, agrupados en cuatro subproyectos: Granada, Almería, País Vasco y UNED. En marzo de 2001, un grupo formado más o menos por los mismos investigadores solicitó un nuevo Proyecto

Coordinado, titulado *Elvira II: Aplicaciones de los Modelos Gráficos Probabilísticos*, que fue concedido por el Ministerio de Ciencia y Tecnología a finales de ese mismo año.

Los principales objetivos de Elvira eran facilitar la colaboración de los diferentes grupos que trabajaban en España sobre modelos gráficos probabilistas. Actualmente tiene implementados más algoritmos de inferencia y aprendizaje que ninguna otra herramienta; ha sido usada por varios cientos de estudiantes de 8 países y se ha utilizado para desarrollar varias aplicaciones, la mayor parte de ellas en el campo de la medicina, aunque también en otras áreas. Uno de los problemas de Elvira es que la necesidad de obtener resultados publicables en un periodo de tiempo breve impidió que se dedicara un esfuerzo mayor a optimizar las estructuras de datos y a diseñar un código más robusto y a generar documentación exhaustiva. Otro problema es que en la actualidad Elvira cuenta con unos 120.000 líneas de código aportados por un elevado número de programadores de instituciones diversas, lo que hace que su mantenimiento sea cada vez más difícil.

En Internet existen muchas aplicaciones sobre modelos gráficos probabilistas (véase la lista elaborada por K. Murphy's en www.cs.ubc.ca/~murphyk/Software/bnsoft.html). La mayor parte de ellas se desarrolla en un grupo por un espacio breve de tiempo y después el proyecto es abandonado. La intención de los autores es que se produzca un fenómeno parecido al de Linux, donde varios investigadores o equipos hagan contribuciones y el proyecto vaya creciendo. En los siguientes párrafos daremos nuestra opinión sobre los motivos por los que esto no ocurre.

Discusión sobre las herramientas de código abierto para MGPs

En términos generales, una aplicación que aspire a contener varios tipos de diferentes modelos y algoritmos debe estar diseñada ex-profeso para ser extensible. La extensibilidad no es un tema sencillo y requiere un diseño muy cuidadoso.

Para que terceras personas puedan ampliar una aplicación existente tienen que entender su arquitectura razonablemente, lo cual requiere una buena documentación, no sólo del código (comentarios), sino también las especificaciones, la arquitectura y el diseño detallado. Es recomendable también disponer de ejemplos sobre cómo extender las funcionalidades. En la mayor parte de los casos, sólo existe la documentación interna

	BNT	PNL	OBP	JavaBayes	BNJ	Riso	BayesLine	Weka	Elvira	Carmen
Lenguaje	Matlab	C++	Python	Java	Java	Java	Java	Java	Java	Java
Licencia	GPL	IOSL	GPL	GPL	GPL	GPL	LGPL	GPL		LGPL
Manuales de usuario	sí	no	sí	sí	sí	sí	no	sí	sí	sí
Foro de usuarios	sí	no	sí	no	sí	sí	sí	sí	sí	sí
Doc. de desarrolladores	no	no	no	no	no	no	no	sí	sí	sí
Lista de desarrolladores	sí	no	sí	no	sí	sí	sí	sí	sí	sí
Doc. en HTML	no	no	no	sí	no	sí	sí	sí	sí	sí
Gestión de versiones	no	no	sí	no	sí	sí	sí	sí	sí	sí
Gestión de errores	no	no	sí	sí	sí	sí	sí	sí	no	sí
Comienzo	1999	2003	2006	1996	2004	2000	2003	1993	1997	2004
Finalización	2007	2005	2007	2001	2004	2004	2003	–	–	–

Tabla 4.1: Paquetes de código abierto para modelos gráficos probabilistas. Las URL de estos paquetes están en la lista de K. Murphy. (OBP = OpenBayes for Python. IOSL = Intel Open Source License.)

del código (muchas veces escasa e incompleta), a partir de ella se generan páginas web usando herramientas como Javadoc o Doxygen, pero esto es claramente insuficiente.

Como consecuencia de la falta de documentación, cuando un usuario ajeno al equipo de desarrollo de una herramienta desea probar un nuevo algoritmo y su funcionamiento, muchas veces es más fácil hacerlo desde cero que entender y utilizar las librerías de dicha herramienta.

4.2. Especificaciones

Una especificación es una enumeración de los requisitos de la aplicación que se pretende desarrollar. La especificación es difícil, porque no siempre es factible conocer todos los requisitos. En nuestro caso, ha sido sencillo porque debido a nuestro conocimiento de los MGPs, en particular, la experiencia de nuestro grupo en el proyecto Elvira, no hemos tenido que utilizar las técnicas habituales de elicitación, hemos sido a la vez el ”cliente´´ y el ”desarrollador´´.

Otro tema importante es determinar los perfiles de usuario a los que está dirigida Carmen. Nosotros hemos identificado tres tipos:

- Investigador de redes probabilistas (generalmente un doctorando o un profesor) que ven la herramienta desde el punto de vista del programador. Estas personas estarán

interesadas en una plataforma para implementar diferentes modelos y realizar experimentos.

- Diseñador de modelos y aplicaciones, como puede ser un médico, un economista, etc.
- Usuario final de los modelos, como puede ser un médico clínico, analistas de diferentes disciplinas, etc.

Por ejemplo, en el diseño de una red para cirugía de cataratas, el primer tipo de usuarios han sido los integrantes de nuestro grupo, el segundo ha estado formado por la doctora Nuria Alonso, la doctora Nuria Fernández y el profesor Javier Diez, director de esta tesis. El tercer grupo son los oftalmólogos del hospital de Fuenlabrada que van a utilizar el sistema de ayuda a la decisión mediante una interfaz específica, integrada en el programa Selene, que accede a Carmen como una librería.

4.2.1. Requisitos funcionales

Los requisitos funcionales son una lista de los servicios que suministra el sistema a los usuarios, que estarán definidos en términos cualitativos y cuantitativos.

Carmen es un proyecto de reingeniería de la herramienta Elvira, aprovechando muchas de las ideas útiles que este programa aportó y sacando lección de la experiencia.

Desde el principio del proyecto Carmen, las funciones que detectamos, como consecuencia de nuestros objetivos, fueron las siguientes:

1. Representación interna de grafos y su edición.
2. Representación interna de MGPs y su edición.
3. Lectura y escritura de MGPs en diferentes formatos, incluido uno propio denominado *CarmenXML*.
4. Inferencia mediante diferentes algoritmos, incluidos las especificaciones de modelos canónicos.
5. Un sistema de aprendizaje para redes bayesianas.
6. Interfaz gráfica de usuario.

4.2.2. Requisitos no funcionales

Los requisitos no funcionales son características del programa, independientes de la funciones que realiza, por eso se denominan así. Por ejemplo, el rendimiento, la usabilidad, la mantenibilidad, etc. Estos requisitos son difíciles de verificar, y por ello son evaluados subjetivamente.

Una de las características del proyecto Carmen es el énfasis en los requisitos no funcionales. La meta no es sólo construir un entorno integrado capaz de representar MGPs, es necesario que dicho entorno sea correcto (libre de errores), fácil de mantener, eficiente, multiplataforma y bien documentado.

Requisitos no funcionales generales

- *Corrección*. No es factible construir un programa de tamaño realista en el que se garantice la ausencia de errores, pero hay que intentarlo. Parcialmente por este motivo hemos escogido Java como lenguaje de programación, ya que una gran cantidad de errores, en otros lenguajes, como C o C++, están asociados a una utilización incorrecta de los punteros, cosa que en este lenguaje es muy difícil que ocurra.

- *Multiplataforma*. Se desea que Carmen pueda funcionar en cualquier entorno. Por este motivo, hemos realizado la implementación en Java, a pesar de que supone una pérdida de rendimiento respecto a otras alternativas como C o C++. En el inicio del proyecto existían otras opciones, como C++ o LISP, pero no estaban lo suficientemente maduras o tenían otros inconvenientes.

- *Mantenibilidad*. Existen cuatro tipos de mantenimiento en un proyecto de software: correctivo (reparar errores), preventivo (reingeniería para mejorar algún aspecto existente), adaptativo (conseguir que siga funcionando pese a cambios tecnológicos) y perfectivo (añadir nuevas funcionalidades). A nosotros nos interesa principalmente el mantenimiento perfectivo. Para poder realizar este mantenimiento es muy importante contar con una buena documentación dirigida al programador. Otro aspecto que se ha cuidado bastante ha sido redactar una normativa acerca de cómo se tienen que realizar los cambios.

- *Reusabilidad*. Es el grado en el que un programa o partes de él pueden ser usados dentro de otro programa. Este objetivo se consigue con un diseño muy cuidadoso que aísla los diferentes aspectos del problema en componentes distintos.

- *Eficiencia*. Según la literatura, esta caracterísitca está reñida con todas las demás: un programa eficiente es menos mantenible, menos reutilizable, más propenso a errores y es difícil que sea multiplataforma. Hemos seguido dos criterios: 1) conseguir eficiencia implementando los algoritmos más rápidos y 2) si hay que elegir entre cualquier característica X y la eficiencia hemos elegido casi siempre la característica X (no ha sido así en la implementación de algunas partes críticas con gran impacto en la eficiencia, como las operaciones básicas sobre potenciales).

Requisitos no funcionales específicos

La librería de grafos tiene que ser genérica de modo que se pueda usar para otros fines, o lo que es lo mismo, tiene que estar desligada de las redes probabilistas.

Por lo que respecta a la librería de redes probabilistas, debe estar diseñada de modo que también pueda ser reutilizable para otro tipo de modelos, no sólo redes bayesianas y diagramas de influencia.

Otro factor importante es que las librerías que operan sobre los modelos descritos no deben depender de la interfaz gráfica. Esto significa que se deben poder manipular tanto desde la interfaz gráfica como desde otro programa, y por tanto, no deben incluir en su diseño consideraciones acerca de como funciona la interfaz gráfica.

Por último, como se pretende que el proyecto pueda ser fácilmente ampliado por otras personas de cualquier nacionalidad, el idioma en el que se ha escrito la documentación interna (comentarios, nombres de componentes, clases y variables) es el inglés.

4.3. Diseño arquitectónico

El diseño arquitectónico está formado por las estructuras de datos y los componentes[3] del sistema, así como sus interrelaciones. El objetivo del diseño arquitectónico es determinar cuáles con los principales componentes y cómo interaccionan. Utilizaremos como notación los diagramas del UML.

Según (6) la documentación del dieseño arquitectónico es importante por tres razones:

1. Permite la comunicación entre los desarrolladores.
2. Destaca las decisiones inicales relacionadas con el diseño que tendrán un impacto profundo en el trabajo posterior.
3. Es un modelo pequeño y comprensible de cómo está estructurado el sistema y cómo trabajan juntos sus componentes.

La descripción de la arquitectura se realiza con *vistas*. Una vista es una representación de un conjunto de los elementos del sistema y de las relaciones entre ellos. Para describir una vista se utilizan *componentes*, que son elementos de cómputo o almacenes de información y *conectores*, que indican la forma en la que los componentes intercambian información en tiempo de ejecución. Un *tipo de vista* está formada por los tipos de elementos y los tipos de relaciones que describen la arquitectura del sistema software desde una perspectiva concreta.

Según la literatura (6), existen tres tipos de vistas o formas de describir la arquitectura:

1. Perspectiva estática: visión del sistema como unidades de implementación.
2. Perpectiva dinámica: visión del sistema como procesos o unidades que se comunican en tiempo de ejecución.

[3]En la literatura existe cierta confusión entre los conceptos de *componente* y *paquete*. Nosotros entendemos por componente un elemento de un programa que tiene una finalidad bien definida; a veces un componente necesita usar los servicios de otros componentes y, a su vez, suministra servicios a través de su interfaz; dentro de un sistema, un componente se puede sustituir por otro que suministre la misma interfaz. Un *plugin* es un tipo de componente que se puede añadir al sistema, de un modo más o menos dinámico, para aumentar sus funcionalidades.
Entendemos un paquete como una agrupación de clases, interfaces y cualquier otro tipo de archivo. El concepto de paquete está más cercano a la implementación, mientras que los componentes tienen un carácter más abstracto y por eso se utilizan para describir la arquitectura. Es típico, y de hecho en Carmen ocurre con frecuencia, que un único componente esté implementado en un único paquete.

3. Perspectiva de localización: visión del sistema como la asignación de estructuras de software a estrucuras de hardware.

Daremos una descripción arquitectónica de Carmen desde las dos primeras perspectivas porque la tercera no tiene sentido en este proyecto, ya que por el momento, Carmen se ejecuta sobre un único ordenador. Cada una de las perspectivas por separado representa información parcial del sistema. Para describir la arquitectura seguiremos el criterio descendente (*top-down*), es decir, primero daremos una visión general, y para no sobrecargar los diagramas con demasiada información, describiremos luego los detalles.

4.3.1. Organización estructural

Dentro de esta perspectiva interesa destacar varios aspectos: por una lado, la *descomposición descendente*, que indica las partes en las que se divide cada componente y qué función realiza; por otro lado, interesa destacar las relaciones de *uso*, que indican dependencia funcional entre componentes. Es interesante señalar también las relaciones de *generalización* o *especialización* entre componentes cuando éstas existan. Por último, a veces conviene mostrar las relaciones de *permiso de uso*. Toda esta información la iremos describiendo en las siguientes páginas.

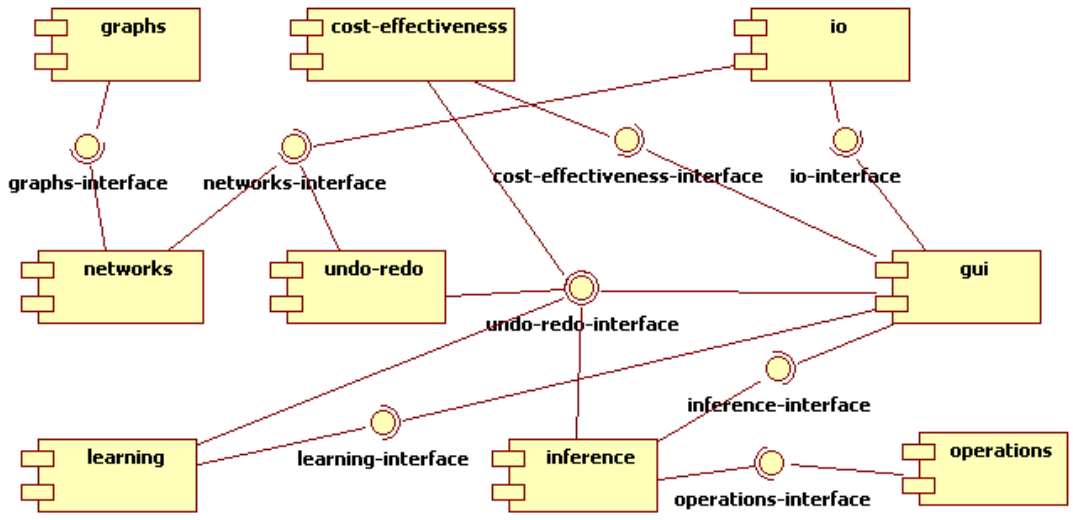


Figura 4.1: Componentes principales en Carmen.

La figura 4.1 muestra una versión resumida de la arquitectura de Carmen. Se han omitido los componentes que no tienen especial relevancia.

La interfaz gráfica de usuario (*gui*) usa a la mayor parte de los demás componentes y el componente *undo-redo* es usado por casi todos los componentes; la entrada-salida usa el componente *networks* para construir las redes que va leyendo o para escribir las redes en disco; el componente *inference* accede indirectamente al componente *networks* para actual sobre las redes en la ejecución de los algoritmos de inferencia a través de *undo-redo* y al componente *operations* para realizar operaciones básicas sobre potenciales.

En los siguientes apartados daremos una descripción más detallada de cada componente. Empezaremos por aquellos que no tienen dependencias respecto a otros y describiremos luego los más complejos.

Grafos

El componente *graphs* se utiliza para crear grafos de cualquier tipo posible. Este componente no depende de ningún otro de este proyecto. Ha sido diseñado para poder ser reutilizado. Aunque contiene pocas clases, ha tenido una evolución muy larga y compleja, y el diseño detallado está expuesto en el capítulo 5.

Redes probabilistas

El componente *networks* de la figura 4.1 es, con mucho, el más complejo. Su diseño detallado también se desarrolla en el capítulo 5.

Figura 4.2: Componentes de las redes probabilistas.

El componente *networks* implementa diferentes tipos de redes bayesianas tales como diagramas de influencia, redes de Markov, árboles de grupos, etc. Utiliza el componente *graphs* para la manipulación del grafo que subyace a cada red probabilista. Todas las modificaciones del grafo que se realicen desde otros componentes pasan a través de la interfaz de *networks*.

Entrada / Salida

Las funciones del componente de entrada/salida (io) son leer y escribir modelos gráficos probabilistas en varios formatos, mostrar los mensajes en el idioma del usuario (internacionalización) y leer y escribir las variables que describen la configuración.

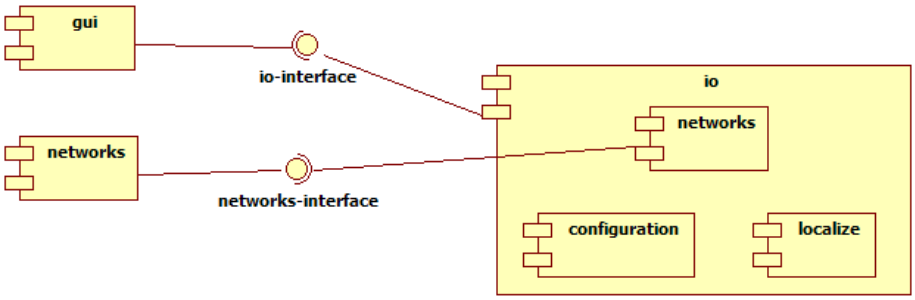


Figura 4.3: Componentes de la entrada/salida (*io*).

En la figura 4.4 se ha expandido el componente *io* de la figura 4.1. Está compuesto por tres partes distintas sin interacción entre ellas:

- *networks* se encarga de la lectura y escritura de redes. Se compone a su vez de otros más sencillos y especializados en diferentes formatos de redes que implementan la interfaz adecuada para lectura y escritura.

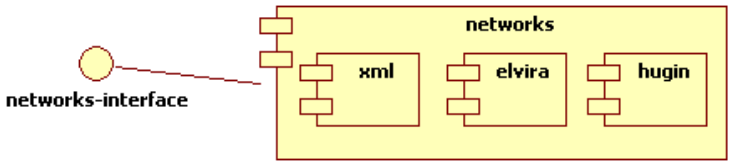


Figura 4.4: Lectura y escritura de redes.

- *configuration* se encarga de leer y escribir la configuración de variables globales del programa y de consultar y actualizar sus valores.

- *localize* se encarga de mostrar los mensajes en el idioma del usuario.

La interfaz *io-interface* está formada por la unión de los interfaces de los tres componentes. Es utilizada principalmente por la interfaz gráfica de usuario (*gui*). Tanto el componente de configuración como el de internacionalización pueden ser utilizados por todos los demás de Carmen, pero no se ha reflejado en el diagrama porque son muy sencillos.

Undo-redo

Este componente se encarga de gestionar la ejecución de algoritmos y las operaciones en la interfaz gráfica de usuario. Puede parece extraño que se haya incluido el componente de hacer-deshacer como una parte importante de la arquitectura, pero el hecho es que tiene una fuerte relación con la inferencia. El motivo es que los algoritmos de inferencia se han tenido que codificar como ediciones sobre las redes en las que operan, y estas ediciones son pasadas al componente *undo-redo*. Se explica en detalle el mecanismo en la sección 6.1 .

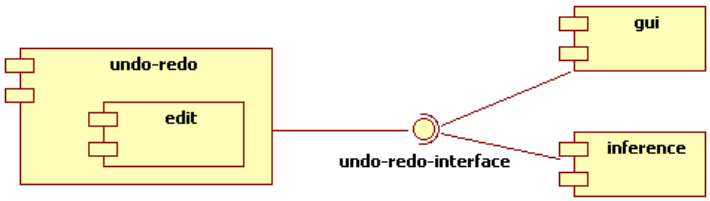


Figura 4.5: Undo-redo y sus relaciones con el resto del programa.

El componente es muy simple, consta de un par de clases y un sub-componente donde se codifican las ediciones, todo ello está basado en el uso de patrones.

Inferencia

El componente *inference* está formado por un conjunto de clases y varios componentes más pequeños. Tiene dependencias de uso con los componentes de *undo-redo* y *networks* y es usado por la interfaz gráfica, como se ve en la figura 4.6.

El desarrollo de este componente ha sido muy lento y ha sufrido muchos cambios. El motivo ha sido que era necesario contar con una buena integración con el resto de los

componentes y poder adecuar fácilmente cambios futuros. En este apartado nos interesa más la perspectiva dinámica, que se tratará en detalle en el capítulo 6.

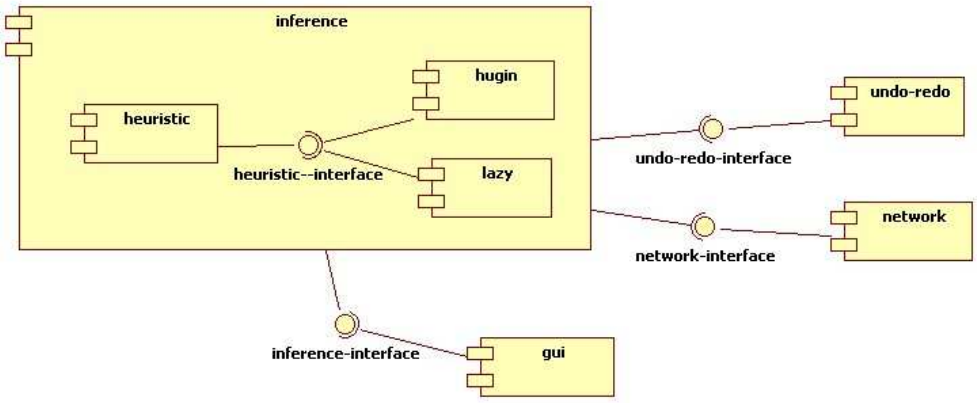


Figura 4.6: Componentes de la inferencia.

Dentro del componente *inference* están un conjunto de clases básicas (no reflejadas en la figura 4.6) relativas a la inferencia utilizando métodos de eliminación de variables sobre redes bayesianas y diagramas de influencia. Existen dos sub-componentes especializados en otros tipos de inferencia: *hugin*, que implementa un algoritmo de agrupamiento para redes bayesianas y *lazy*, que implementa el algoritmo de propagación perezosa, también sobre redes bayesianas.

El componente *heuristics* tiene una función totalmente distinta: codifica las heurísticas utilizadas por los algoritmos de inferencia para hallar secuencias de eliminación de variables.

Capítulo 5

Grafos y modelos gráficos probabilistas

El propósito de este capítulo es explicar los criterios que han guiado la construcción de dos bibliotecas: una de grafos y otra de MGPs, siguiendo los principios de la ingeniería del software. En la sección 5.1 se detallan los objetivos y requisitos de la librería de grafos, en la 5.2 se desarrollan los modelos de análisis y diseño de la librería de grafos y en la 5.3 se desarrollan los modelos de análisis y diseño de la librería de MGPs.

5.1. Introducción

Nuestro objetivo inicial era crear una librería sobre grafos genérica y eficiente que sirviera de base para otras estructuras más especializadas. El objetivo de generalidad significaba que las clases de la cima de la jerarquía tenían que ser el común denominador de todos los posibles desarrollos posteriores. El problema que se nos planteaba es que los grafos son la estructura de datos más genérica posible. Esto significaba que el objetivo de generalidad posiblemente no se pudiera conseguir más que a un nivel de análisis o que se pudiera conseguir sacrificando el rendimiento.

Dada la complejidad de la tarea tuvimos que dar muchos pasos adelante y atrás hasta que conseguimos un modelo viable. El resultado ha sido una biblioteca flexible y eficiente tanto en el uso de memoria como en velocidad.

Como norma general, los diagramas UML de clases y de secuencias que se presentan en este capítulo y en el siguiente están resumidos, es decir, omiten alguntos atributos, clases asociadas y mensajes que no son relevantes para entender la idea que se pretende explicar.

5.2. Grafos

5.2.1. Requisitos de la biblioteca de grafos

La biblioteca debe permitir la creación de grafos en memoria, la edición de sus propiedades (añadir y borrar nodos y enlaces) y asociar objetos a los nodos o a los enlaces.

Un aspecto importante es que existen muchas características que definen a un grafo: ser dirigido, admitir ciclos, admitir bucles, ser etiquetado, etc. Cada posible combinación de estas propiedades puede dar como resultado un tipo de grafo con un diseño totalmente distinto.

Los requisitos no funcionales que hemos determinado son los siguientes:

- La biblioteca debe estar diseñada sin condicionamientos por parte de otros componentes que se prevee que la usen.
- Debe ser fácil de usar por parte del programador. Esto se consigue con un buen diseño y una buena documentación, con ejemplos adecuados.
- Ausencia de fallos. No es posible garantizar que un programa no tenga fallos, pero se puede minimizar la probabilidad de que ocurran haciendo pruebas exhaustivas e invluyendo en el código documentación orientada a verificación.
- Eficiencia temporal y espacial. La eficiencia es un requisito reñido con todos los demás. El criterio que se ha seguido es conseguir la eficiencia usando los algoritmos que sean más eficientes, sin sacrificar los demás requisitos para conseguirla.

5.2.2. Modelo de análisis de grafos

Existen dos formas de resolver el problema de crear una estructura que clasifique los grafos:

1. Crear una clase por cada tipo de grafo que pueda existir y relacionar estas clases por herencia cuando sea posible. Esta aproximación tiene la ventaja de encontrar la mejor solución para cada tipo de grafo y el grave inconveniente de que el número de clases que pueden existir es enorme, porque las diferentes restricciones que se pueden aplicar a los grafos generalmente se pueden combinar entre ellas. Otro problema es que si se desea tener un conjunto de clases relacionadas lo más

lógico es utilizar la herencia para no duplicar código, lo cual obliga a tener un criterio para elegir por qué característica se empieza a dividir la jerarquía de clases. Posteriormente esto se complica cuando una clase herede características indeseadas o se necesite hacer un cambio en niveles superiores de la jerarquía. De este modo el mantenimiento resulta sumamente complicado. Un último problema es que si se desea conseguir una implementación eficiente en las clases especializadas, las estructuras de datos serán diferentes en cada una de ellas por estar adaptadas a funciones específicas y como consecuencia será difícil utilizar la herencia.

2. Crear un número reducido de clases y asociar a cada una un conjunto de *restricciones* que definan el tipo de grafo que representa. Esta aproximación es la que finalmente hemos escogido, porque nos da un modelo más simple sin renunciar a la eficiencia. Otra ventaja es que la representación de cada posible tipo de grafo está desacoplada de las demás.

Las restricciones que describen un grafo pueden ser varias; la mayor parte son propiedades estructurales, es decir, referidas a la topología del grafo. La lista siguiente describe las más importantes, o al menos, las necesarias para crear prácticamente los principales tipos de grafos que hay en la literatura:

- Puede ser etiquetado o no etiquetado.
- Puede ser dirigido, no dirigido o mixto. Cuando un grafo lleva la restricción "dirigido" sólo admite enlaces dirigidos. Un grafo mixto no lleva la restricción "dirigido" ni la "no dirigido", y por tanto, admite ambos tipos de enlaces.
- Puede ser etiquetado o no, en Carmen, un grafo es etiquetado cuando cada enlace lleva una etiqueta; entre dos nodos puede haber varios enlaces siempre que lleven etiquetas distintas. Por ejemplo, los grafos que se utilizan en teoría de autómatas son etiquetados.
- Puede admitir bucles o no. Es otra característica booleana aplicable a todos los tipos de grafos.
- Puede admitir ciclos o no. Esta característica booleana sólo se aplica al caso de grafos dirigidos o mixtos.

- Puede admitir enlaces de un nodo a sí mismo o no (no es el caso de redes bayesianas y diagramas de influencia pero sí en ciertas representaciones de modelos gráficos probabilistas temporales.)

En un grafo no etiquetado, los enlaces pueden tener objetos asociados, pero entre dos nodos sólo puede haber un enlace de cada tipo; es decir, entre dos nodos A y B sólo puede haber tres enlaces: $A \rightarrow B$, $B \rightarrow A$ y $A - B$

Además de las propiedades estructurales los nodos suelen guardar cierta información; esto también hay que reflejarlo en el modelo de análisis. La información guardada puede ser un atributo o varios; en este nivel de análisis hemos modelado esto como un *objeto genérico*.

Los enlaces, por su parte, pueden tener dos carácterísticas: *direccionalidad* (enlace dirigido o no) y *contenido* (al igual que los nodos, los enlaces pueden contener algún tipo de información).

El modelo de análisis obtenido puede ser utilizado para desarrollar, a partir de él, muchos modelos diferentes de diseño, optimizados para diferentes tipos de problemas.

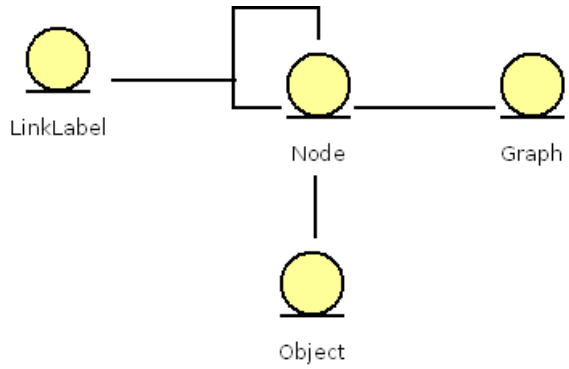


Figura 5.1: Modelo de análisis de grafo.

5.2.3. Modelo de diseño de grafos

En esta sección introducimos todos los condicionantes relativos a la implementación. En primer lugar vamos a ver las dos formas en las que se ha resuelto el problema en la

literatura que existe al respecto, comentaremos ventajas e inconvenientes y, por último, describiremos nuestra solución. El modelo resultante es más complejo que el modelo de análisis porque se han incorporado muchos condicionantes y ha sido el resultado de un proceso largo.

Aspecto estructural

Según la literatura existen dos estructuras de datos para representar grafos:

a) Una única matriz de adyacencia: En un grafo de n nodos, la matriz de adyacencia es una tabla de ceros y unos de tamaño $n \times n$ donde la casilla (i, j) tiene asignado el valor 1 si existe un enlace desde el nodo X_i hasta el X_j. Si el enlace es no dirigido tanto la casilla (i,j) como (j,i) tendrán asignado el valor 1. Un problema de esta estructura de datos en un grafo mixto es que no distingue el enlace no dirigido $X_i - X_j$ de la presencia de dos enlaces dirigidos o $X_i \rightarrow X_j$ y $X_j \rightarrow X_i$.

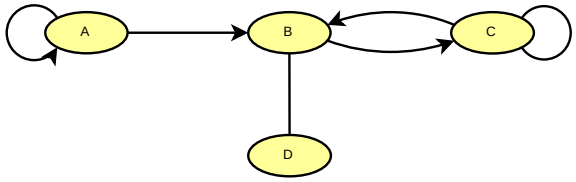
b) Dos matrices de adyacencia: Una forma de representar ambos tipos de enlaces sería usar dos matrices: una para enlaces dirigidos y otra para no dirigidos; esta última es simétrica, véase la figura 5.2(b).

c) Una lista de adyacencia por cada nodo: En este caso, cada grafo se representa mediante una lista de nodos en que cada uno de ellos tiene a su vez una lista de aquellos nodos con los que está conectado. En el caso de grafos dirigidos se pueden incluir sólo los hijos o también los padres. Si se incluyen ambos o si el grafo es mixto se debe indicar con una etiqueta el tipo de nodo al que se apunta: padre, hijo o hermano. Véase la figura 5.2(c). En realidad, cada elemento de la lista asociada a un nodo no contiene sólo un nodo, sino también información relativa a si es padre, hijo o hermano.

Las ventajas e inconvenientes de ambas representaciones son las siguientes:

Matrices de adyacencia:

- Una ventaja de esta representación es que para saber si dos elementos están conectados basta con consultar una entrada en una matriz bidimensional, lo cual tiene una complejidad constante: $O(1)$.



(a) Grafo.

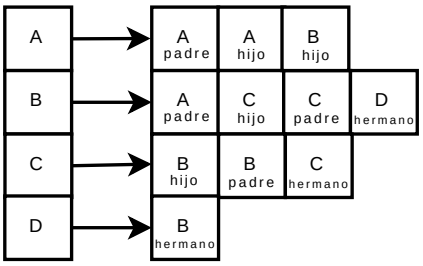
	A	B	C	D
A	1	1	0	0
B	0	0	1	0
C	0	1	0	0
D	0	0	0	0

Enlaces dirigidos

	A	B	C	D
A	0	0	0	0
B	0	0	0	1
C	0	0	1	0
D	0	1	0	0

Enlaces no dirigidos

(b) Representación del grafo mediante matrices de adyacencia.



(c) Representación del grafo mediante listas de adyacencia.

Figura 5.2: Ejemplo de un grafo mixto representado con matrices y listas de adyacencia.

- En cambio, para obtener los padres, hijos o vecinos de un nodo hay que recorrer toda la columna de la matriz de enlaces dirigidos correspondiente a ese nodo, lo cual tiene complejidad $O(N)$. En el caso de los MGPs, n va a ser un número mucho más grande que el número de vecinos de un nodo y por tanto esta representación resulta poco eficiente.
- Una matriz de adyacencia ocupa una cantidad de memoria proporcional a n^2, lo cual puede hacer inmanejables los grafos de varios miles de nodos.
- Eliminar/Añadir enlaces tiene complejidad constante porque sólo hay que modificar un elemento de la matriz.
- Si se añade un nodo hay que reconstruir la matriz entera, lo que es computacionalmente muy costoso: la complejidad temporal es $O(n^2)$ y en memoria cada nodo extra supone $2n+1$ entradas nuevas.
- Si hay objetos asociados a los enlaces tiene que haber otra matriz de adyacencia que contenga los punteros a dichos objetos.

Una lista de adyacencia por cada nodo:

- En la función de obtención de los hijos, padres o vecinos de un nodo basta con recorrer la lista de adyacencia y seleccionar el subconjunto que nos interese. Si los nodos de un grafo tienen en promedio k enlaces, en grafos poco poblados $k \ll n$, la operación cuesta $O(n)$ en el peor caso.
- Para saber si un nodo es hijo, padre o vecino de otro hay que recorrer la lista asociada a ese nodo. En el caso peor la operación tiene complejidad $O(n)$, aunque en grafos en promedio es k.
- La memoria que ocupa es proporcional al número de nodos más el número de enlaces: $O(n^2)$ en el peor caso y $O(n)$ en el promedio.
- Añadir un enlace tiene complejidad constante, $O(1)$, porque sólo hay que añadir un elemento al final de la lista de cada nodo; borrar un enlace tiene complejidad lineal, $O(n)$, porque hay que borrar dos elementos de una lista.
- Añadir un nodo supone sólo añadir un elemento a una estructura de datos de tipo lista, lo cual tiene complejidad constante, $O(1)$.

- Si hay objetos asociados a los enlaces, estos tienen que representarse de algún modo, por ejemplo, haciendo que cada elemento de la lista asociada a un nodo no tenga sólo un nodo y una indicación (padre/hijo/hermano), sino también un puntero hacia dicho objeto.

d) Multilistas de adyacencia: Son una mejora de lo anterior, consiste en asignar a cada nodo tres listas de adyacencia, una para cada tipo de nodo (hijos, padres y hermanos), como puede verse en la figura 5.3. La ventaja es que las operaciones que consisten en acceder a todos los padres, hijos o hermanos, que son muy frecuentes, pasan a tener complejidad $O(1)$, porque sólo hay que devolver como resultado un enlace a la lista pedida. La operación de buscar un nodo hijo, padre o hermano sigue teniendo complejidad ($O(n)$), pero el caso promedio es más rapido porque la búsqueda se restringe a una de las tres listas. En cuanto a complejidad espacial, suele disminuir, porque cada elemento de las listas no tiene que tener un indicador de si el nodo es padre, hijo o hermano.

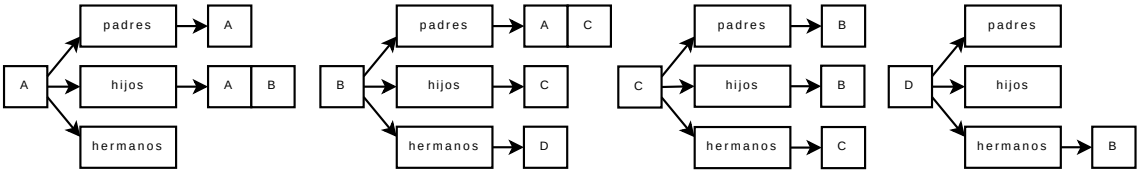


Figura 5.3: Ejemplo de la figura 5.2(c) usando multilistas de adyacencia.

En nuestro diseño hemos elegido las multilistas de adyacencia porque, cuando $k \ll n$, como suele ocurrir en los MGPs, éstas suponen un gran ahorro de memoria respecto a las matrices de adyacencia y porque las operaciones más habituales, como son acceder a todos los padres, hijos o hermanos de un nodo tienen complejidad $O(1)$, lo cual significa una eficiencia mucho mayor.

Representación de los enlaces

Dentro de las listas de adyacencia un enlace se puede representar de dos formas:

- *Implícita*: cada nodo tiene una colección de punteros a otros nodos. La ventaja de esta representación es que ocupa menos memoria y es más rápida porque se accede desde el nodo A al nodo B a través de una referencia en lugar de a través de un objeto intermedio.

- *Explícita*: cada nodo tiene una colección de objetos de tipo *enlace* que contienen referencias a los dos nodos enlazados. La ventaja de esta representación es la flexibilidad: el enlace puede tener atributos (etiquetas) y puede haber varios enlaces distintos entre dos nodos A y B. El inconveniente es el consumo de memoria asociado a un objeto por cada enlace y que esta estructura va a ser siempre más lenta que una asociación.

La decisión que hemos tomado en Carmen es que un grafo puede usar simultáneamente (de ser necesario) ambas representaciones. El criterio que se sigue es tener siempre la representación con enlaces implícitos y, si se da uno de los casos indicados anteriormente, crear además la representación con enlaces explícitos. Esta decisión es más compleja de programar y requiere dos tipos de actualizaciones distintas cuando se cambia la estructura del grafo, pero tiene como ventaja la flexibilidad de los enlaces explícitos y la eficiencia cuando no se necesitan enlaces explícitos.

En el caso de usar varios enlaces explícitos distintos entre dos nodos A y B, también existe uno implícito (sin información asociada). Ese enlace se borra cuando se borra el último de los enlaces explícitos.

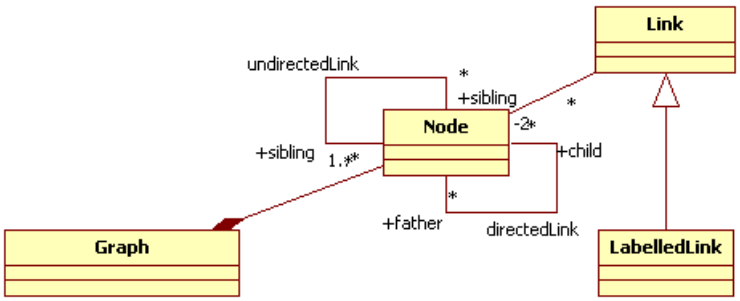


Figura 5.4: Enlaces implícitos y explícitos de un grafo.

Aspecto de clasificación

Este apartado ha sido sin duda el más complicado. Ha supesto muchas vueltas adelante y atrás en el diseño hasta que hemos encontrado la solución que nos parece más adecuada (lo más habitual en la documentación de un sistema es exponer las decisiones que se han

tomado y sus motivos, pero creemos que en este caso las modificaciones que se han ido introduciendo en el diseño son tan interesantes como la solución final).

Los modelos gráficos probabilistas usan diferentes tipos de grafos para representar sus estructuras de datos y todos esos tipos de grafos condicionaron nuestros diseños preliminares de esta librería.

Primera aproximación: herencia

Se hizo una recopilación de las características importantes en grafos, que básicamente eran propiedades estructurales. Se definieron algunas propiedades como variables booleanas y otras se utilizaron para derivar por herencia una jerarquía de clases. Los grafos se dividieron en dos subclases: etiquetados y no etiquetados.

Las variables booleanas que escogimos fueron:

- *admitsCycles*
- *admitsDirectedLinks*
- *admitsUndirectedLinks*
- *admitsLoops*
- *admitsSelfLoops*

Además de las clases de la figura 5.5, se diseñaron varias clases relativas a modelos gráficos probabilistas. Estas clases heredaban de *UnlabelledGraph*.

Por otra parte, cada vez que se realiza una operación sobre un grafo puede ocurrir que se intente hacer algo no permitido, como por ejemplo crear un grafo en un grafo acíclico. En esta primera versión, siempre se hacían comprobaciones cada vez que se modificaba el grafo, lo cual complicaba el código y empeoraba la eficiencia.

Inicialmente la clase *Graph* contenía una lista de nodos; posteriormente se convirtieron las clases relativas a grafos de la figura 5.5 en clases abstractas y se eliminó esa lista. El objetivo inicial era que fueran las clases que se derivaran a partir de ellas las que hicieran la implementación más eficiente en cada nodo de la lista de nodos.

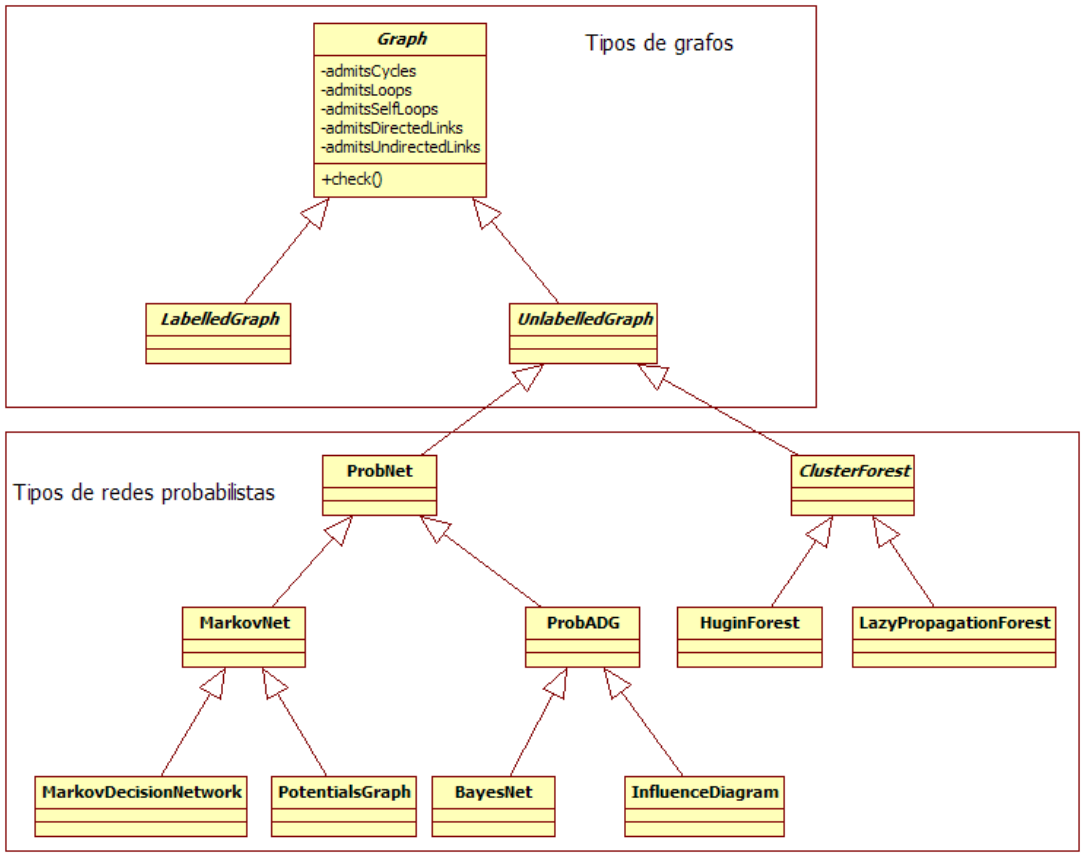


Figura 5.5: Primera aproximación para representar grafos y MGPs.

Problemas que nos planteó esta aproximación:

- Fuerza a manejar un modelo de gran tamaño y por eso costó mucho trabajo obtenerlo. Además, es muy difícil determinar cuál es la mejor forma de dividir las clases de la cima de la jerarquía en subclases.
- El mecanismo de herencia supone un acoplamiento muy fuerte entre las clases que heredan, y ese acoplamiento da serios problemas en el momento en el que se necesita hacer una modificación en las clases de la cima de la jerarquía.
- Los grafos se quedaban reducidos a un conjunto de clases abstractas sin ninguna funcionalidad real.

Segunda aproximación: asociación

La conclusión a la que llegamos es que era mejor desacoplar el aspecto estructural del aspecto probabilista, utilizando asociación en vez de herencia, tal como recomiendan algunos expertos en ingeniería del software (67). La idea se indica esquemáticamente en la figura 5.6, en la que se han omitido las clases que no son relevantes.

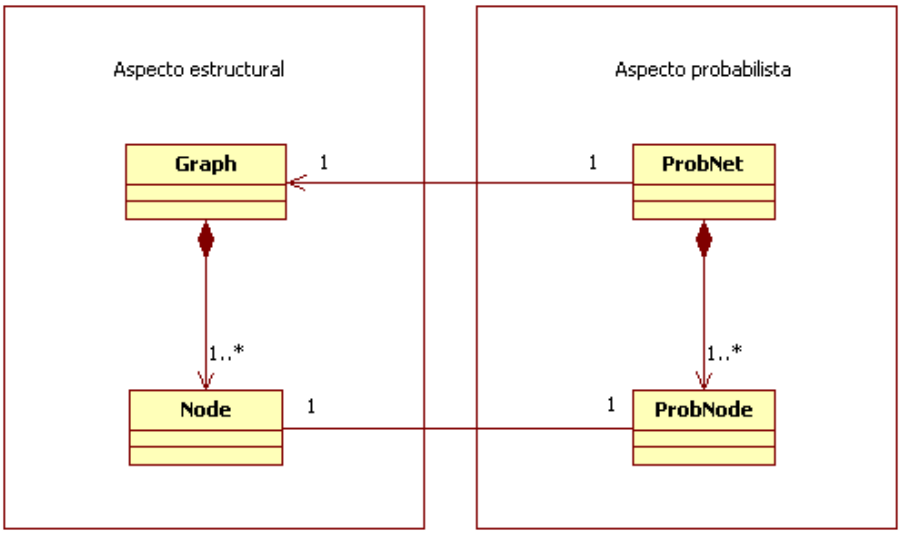


Figura 5.6: Segunda aproximación para representar grafos MGPs. Nótese que se utiliza asociación en vez de herencia.

ProbNet y *Graph* tienen una asociación uno a uno. *ProbNet* puede acceder a *Graph*, pero no al revés. *Node* y *ProbNode* tienen una asociación uno a uno y ambos pueden acceder al otro.

Esta solución tiene ventajas importantes sobre la primera aproximación:

- Desacopla el aspecto estructural y el probabilista. Como consecuencia, las posibles modificaciones en la librería de grafos no afectan a la otra mientras conserve su funcionalidad.
- Redistribuye las responsabilidades entre las clases *Graph* y *ProbNet* de un modo más coherente[1].
- No hace falta definir una subclase para cada tipo de red pues el propósito de éstas se consigue mediante el uso de restricciones como vamos a explicar a continuación.

Uso de restricciones explícitas

Existen muchos tipos de grafos y modelos probabilistas, por este motivo, usar la herecia y variables booleanas para seleccionar el tipo de modelo no nos parecía un diseño orientado a la extensión futura. Los diferentes tipos de grafos son en realidad distintas restricciones aplicadas sobre un modelo general simple. La decisión final fue incluir explícitamente estas restricciones en el diseño codificadas como clases.

La cuestión que se debe resolver es la forma de representar las restricciones en el diseño.

Las posibles alternativas son:

1. Las restricciones están imbuidas en la programación de los métodos que manipulan el grafo.
2. Las restricciones se representan explícitamente mediante clases. En el momento de la creación del modelo, se le añaden por asociación las restricciones oportunas.

[1] De acuerdo con (67), un objeto no debe verse principalmente como un conjunto de datos y métodos, sino como una entidad que asume una responsabilidad.

La primera alternativa es más eficiente porque no hay que manipular objetos para comprobar las restricciones; sin embargo, nosotros hemos escogido la segunda alternativa, porque según nuestro criterio, conviene aceptar una pequeña pérdida de eficiencia si con ello se consigue una ganancia importante en mantenibilidad, flexibilidad y claridad.

En la primera aproximación, el hecho de que una red bayesiana sólo tiene enlaces dirigidos se indicaba asignando el valor *true* al atributo *admitsDirectedLinks*, heredado de la clase *Graph*, y el valor *false* a *admitsDirectedLinks*. En cambio, en esta segunda aproximación, cuando queremos tener una red bayesiana creamos un objeto de la clase *ProbNet* y le asociamos la restricción compuesta *BNConstraint*, que incluye, entre otras, la restricción *OnlyDirectedLinks*.

Las restricciones tienen varias ventajas:

1. Partiendo de un modelo simple se pueden representar todos los tipos de modelos posibles y, cuando se desea añadir otro tipo de modelo, sólo hay que codificar las restricciones adecuadas. De este modo, la inclusión de un nuevo tipo no supone cambios en el código existente.

2. Existen situaciones en las que no se puede garantizar que no se intente hacer una operación inválida; por ejemplo: un usuario editando una red con la interfaz gráfica puede por error intentar crear un ciclo en una red acíclica. Las restricciones capturan el error y lanzan una excepción, que el manipulador de la red se encarga de tratar (en este ejemplo, la interfaz gráfica puede mostrar un mensaje de error al usuario). La ventaja es que no hay que codificar toda la lógica de control para tratar todos los tipos de errores que se puedan dar cuando se va a realizar una operación.

3. Se separa el tratamiento de los dos tipos de errores: los errores debidos al usuario se controlan con la comprobación de las restricciones asociadas a cada operación, mientras que los errores de programación se controlan con pruebas de unidad y de integración.

4. El usuario (programador) puede definir nuevas restricciones. Por ejemplo, para inplementar un nuevo MGP o un algoritmo que sólo sirve para cierto subtipo de los ha existentes.

5. Encapsula el aspecto del control de errores y el de la manipulación del grafo en lugares distintos, lo que implica que para cada tipo de grafo se deben programar clases que codifican las restricciones aplicables.

6. Cuando se sabe que las operaciones que se van a realizar son correctas no es necesario comprobar las restricciones; esto ocurre por ejemplo en la ejecución de un algoritmo o en la lectura de una red. En cambio, en nuestra primera aproximación, las restricciones asociadas a las propiedades booleanas se comprobaban siempre, lo cual implicaba una pérdida de eficiencia.

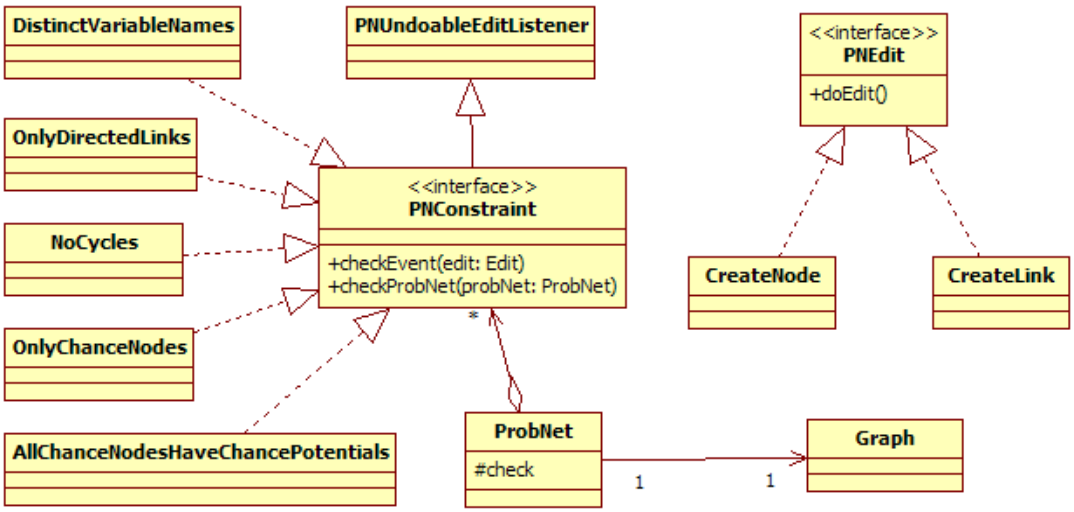


Figura 5.7: Diagrama de clases de restricciones asociadas a un grafo. Se representan también las ediciones.

5.3. Modelos Gráficos Probabilistas

5.3.1. Modelo de análisis de MGPs

Desde el punto de vista del análisis y del diseño, los MGPs son relativamente complejos porque introducen varios conceptos distintos interrelacionados. Los tipos de modelos gráficos que se desean representar son en primer lugar redes bayesianas y diagramas de influencia, aunque el diseño es lo suficientemente flexible para acomodar otros, como los RADS, los modelos temporales markovianos, etc.

Cada nodo de un modelo gráfico tiene asociada una variable y un número variable de potenciales de probabilidad o de utilidad. Los nodos representan las características estructurales de la red y, en consecuencia, tienen enlaces con otros nodos. Por otra parte, cada variable está asociada a tantos nodos como modelos probabilistas de los que forma parte.

En el caso de variables discretas, que es el único que hemos implementado en Carmen, cada potencial viene dado por una tabla de probabilidades o una tabla de utilidades, definida en función de un conjunto de variables.

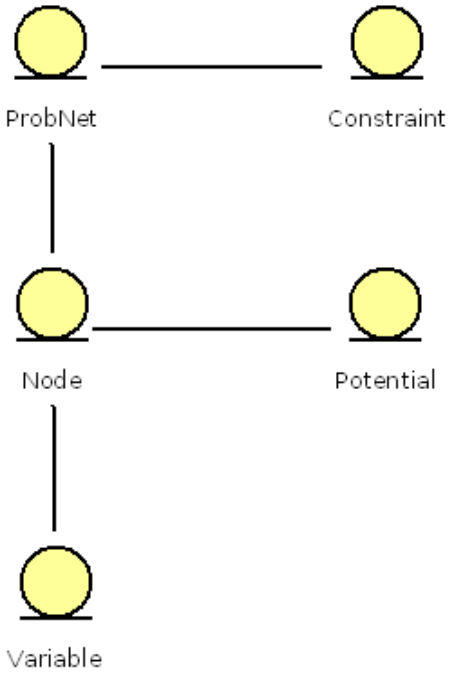


Figura 5.8: Modelo de análisis de modelos gráficos probabilistas.

La figura 5.8 es una versión simplificada del modelo real, en la cual todavía no hemos tenido en cuenta que pueden existir varios tipos de potenciales, de nodos o de variables, ni la estructura gráfica de la red, aunque si hemos incluido la existencia de restricciones explícitas porque son importantes en nuestro modelo.

5.3.2. Modelo de diseño de MGPs

Al igual que en el caso de la sección 5.2.3, introduciremos varios condicionantes necesarios para la implementación del modelo anterior, que complicarán bastante el modelo de diseño. Vamos a dividir este modelo en cinco partes bien diferenciadas:

1. *Redes probabilistas*. Los tipos que hemos considerado inicialmente son redes bayesianas, redes de Markov y diagramas de influencia.

2. *Variables* . Cada una representa un objeto matemático que puede tomar varios valores.

3. *Nodos probabilistas*. Son estructuras especializadas que almacenan la información necesaria para las redes anteriores. Cada nodo representa una variable[2].

4. *Potenciales*. Guardan las tablas de probabilidades (si las variables son discretas).

5. *Restricciones*. Son las condiciones que tienen que cumplir tanto los grafos como los modelos probabilistas.

Redes probabilistas

Existen varios tipos de redes probabilistas: redes bayesianas, redes de markov, diagramas de influencia, arboles de cliques, etc. Nuestra aproximación ha sido encontrar un mínimo común denominador que hemos codificado en una clase *red probabilista* (*ProbNet*).

Una red probabilista puede tener los tres tipos de nodos que se definen en los diagramas de influencia: de azar, de decisión y de utilidad.

Potenciales

La información que contiene un potencial, definidos en la sección 2.1, puede variar en función del tipo de variables que almacena, aunque en la versión actual de Carmen (*baseline*) sólo hemos tratado variables discreats.

[2]La distinción entre nodo y variable sirve para ahorrar tiempo y memoria, especialmente, en los algoritmos de inferencia. Así, al clonar una red para operar sobre ella (por ejemplo, borrando nodos y enlaces) no hace falta clonar todas las propiedades de cada variable; basta poner un puntero, de modo que ambas redes compartan las mismas variables.

Para dar una solución general, hemos creado una jerarquía de clases, cuya cima es la clase *Potential*, prácticamente vacía pues sólo contiene el conjunto de variables sobre el que está definido el potencial.

En la clase *FSPotential* y sus hijas están representados los potenciales que trabajan con variables de estados finitos (*FSVariables*). La clase *TablePotential* es un tipo de *FSPotential* con una tabla de probabilidades explícita y la clase *GTablePotential* es un *TablePotential* cuya tabla de probabilidades almacena objetos de cualquier tipo.

La clase *ICIPotential* (siglas en inglés de *Independence Causal Influence Potential*) y sus derivadas son los potenciales usados en los modelos canónicos, que en lugar de tener una tabla de probabilidades explícita, están formadas por un conjunto de subpotenciales del tipo *TablePotential*.

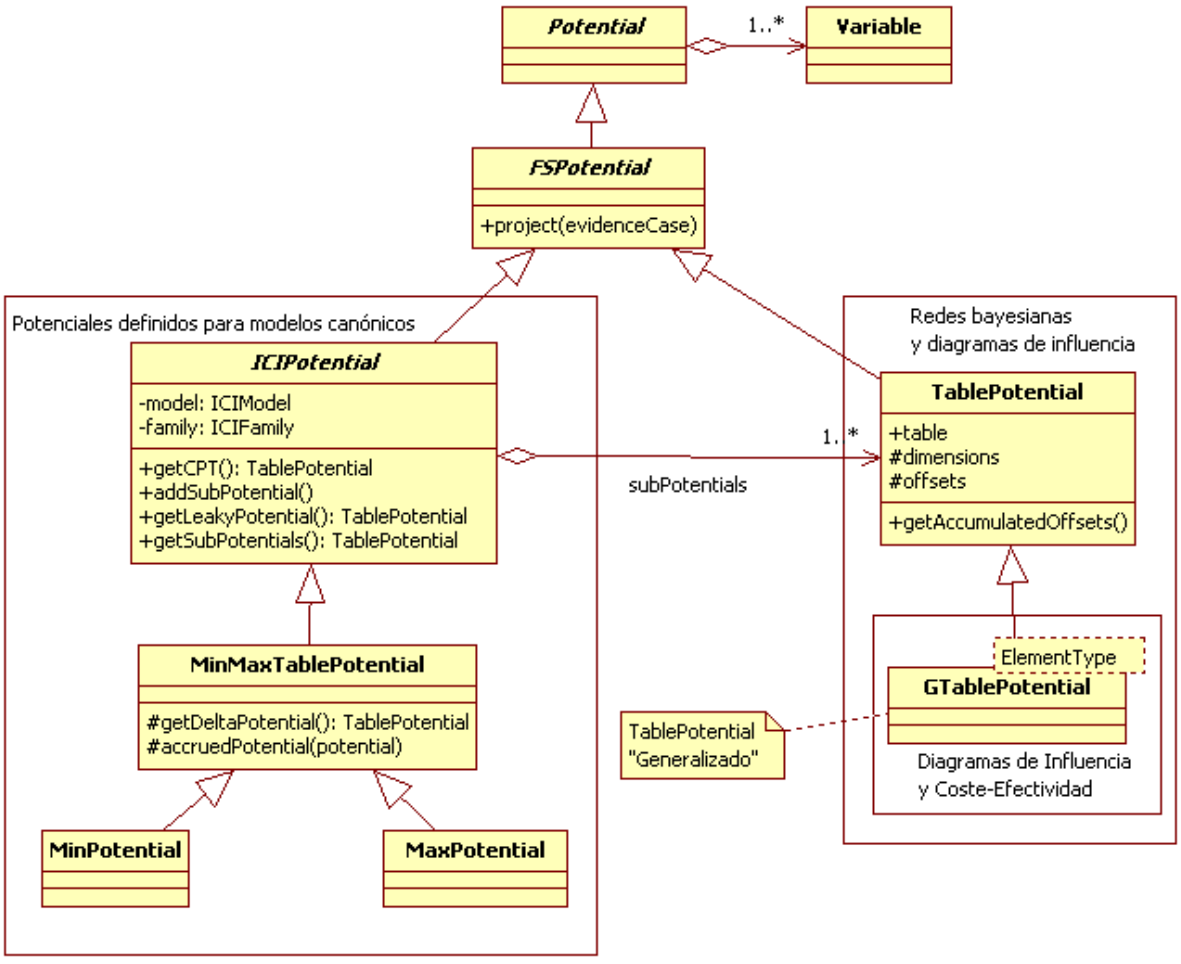


Figura 5.9: Modelo de diseño para los potenciales.

Nodos probabilistas

Un nodo es una estructura muy sencilla: por un lado hace referencia a una estructura que almacena la información relativa a la estructura gráfica (padres, hermanos e hijos) y por otro almacena la información relativa a la red probabilista (variable asociada y un conjunto de potenciales). Ver la figura 5.10.

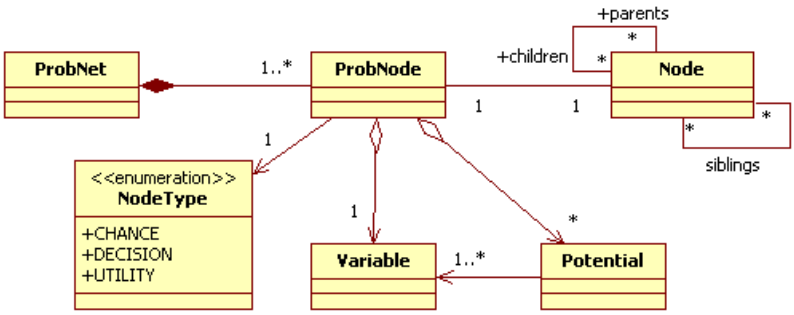


Figura 5.10: Modelo de diseño para los nodos probabilistas.

Variables

En modelos probabilistas básicamente existen dos tipos de variables: discretas y continuas. Las variables discretas podrían tener un número infinito de valores (por ejemplo, para representar los números naturales) o un número finito (*finite state variables*)

Las variables continuas pueden tomar cualquiera de los infinitos valores de un intervalo $[a,b]$. El valor que toma una variable continua es un número real. En algunos casos puede interesar indicar la unidad en que se mide (por ejemplo, kilogramos o milímetros de mercurio) y la precisión (0,01, 0,001, etc.).

Una variable continua puede discretizarse definiendo un conjunto de intervalos. Por ejemplo, $\{[0,0,25],[0,25,0,75],[0,75,1]\}$. En las aplicaciones de MGPs que hemos construido en nuestro grupo, la mayor parte de ellas en medicina, frecuentemente hemos utilizado variables discretizadas debido fundamentalmente a la dificultad de construir y hacer inferencia en modelos con variables continuas.

Hemos representado este tipo de variables de la siguiente forma: hemos creado la clase *PartitionedInterval* que contiene la información relativa a cada intervalo (el valor de los extremos y si están abiertos o cerrados). Todos los tipos de variables los representamos con una única clase *Variable* que indica el tipo de variable (discreta, continua, discretizada, etc). El motivo para esta representación es que así es muy fácil transformar una variable discreta en una discretizada si se le añade la información relativa a los intervalos.

Evidencia Llamamos *hallazgo* (*finding*) a una asignación de valor a una variable. Un *caso de evidencia* (*evidence case*) es un conjunto de hallazgos.

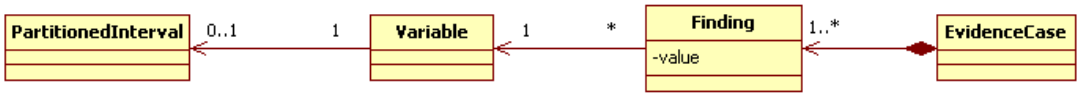


Figura 5.11: Modelo de diseño para variables y hallazgos.

Restricciones

En el apartado 5.2.3 se habló de la asociación de restricciones que definían las propiedades de la red. En este apartado vamos a definir más detalladamente cómo se han diseñado las restricciones.

Una restricción es una condición que se aplica a una clase, por ejemplo, una red bayesiana tiene la restricción de no tener ciclos.

Una restricción puede ser simple, si comprueba una condición específica, o compuesta, si está formada por varias restricciones simples, véase la figura 5.12 , por ejemplo, la restricción compuesta *BNConstraint* está formada por las restricciones simples: *NoCycles, AllChanceVariablesHaveChancePotentials, NoSelfLoops, OnlyChanceNodes* y *OnlyDirectedLinks*.

Antes de asociar una restricción a una red hay que comprobar que se cumple para toda la red. Cuando se sabe que una restricción se cumple para una red y se quiere hacer una pequeña modificación, se hace una comprobación incremental.

Puede ocurrir que un algoritmo necesite para aplicarse sobre una red determinada comprobar otras restricciones que no estén asociadas a la red. Esto es posible también sin necesidad de asociar esas restricciones.

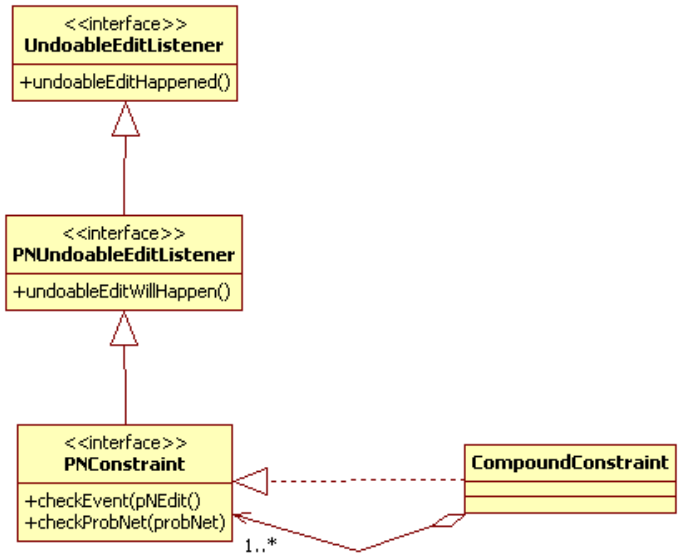


Figura 5.12: Modelo de diseño para restricciones.

5.3.3. Resumen

El resultado final del diseño tanto de grafos como de MGPs queda resumido en la figura 5.13.

Por un lado, el tipo de MGP está definido por sus restricciones, que son un conjunto de clases que implementan la interfaz *PNConstraint* y, por otra parte, el MGP se compone de: 1) un grafo asociado, que representa el aspecto estructural del modelo y, a su vez, formado por un conjunto de nodos relacionados entre ellos por enlaces dirigidos o no, y 2) un conjunto de nodos probabilistas, cada uno de ellos asociado a: un nodo del grafo, una variable y una colección de potenciales.

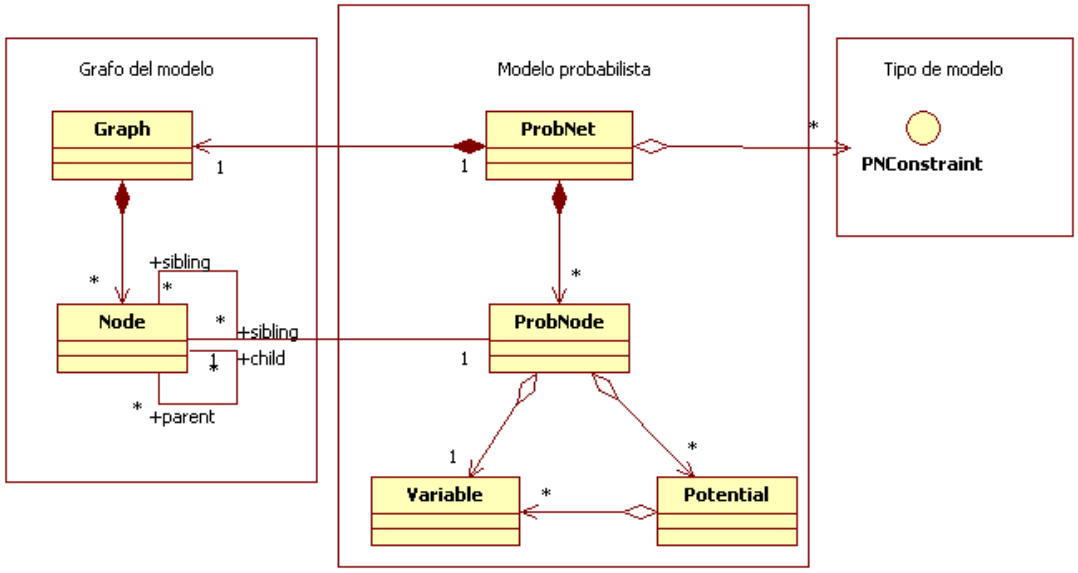


Figura 5.13: Resumen simplificado del diseño de modelos gráficos probabilistas.

Capítulo 6

Operaciones sobre MGPs

Este capítulo desarrolla el aspecto dinámico de Carmen. En la sección 6.1 explicamos el modo en el que se han diseñado los algoritmos y mediante un nuevo patrón de diseño relativo a la modificación de objetos sujetos a restricciones; la sección 6.2 detalla el diseño de los diferentes algoritmos implementados para redes bayesianas (eliminación de variables en la sección 6.2.1; describimos el método básico de agrupamiento y la propagación perezosa en la 6.2.2 y, los modelos canónicos en la sección 6.2.3. En la sección 6.3 describimos la implementación del algoritmo de eliminación de variables para diagramas de influencia y la forma de utilizarlos; la sección 6.4 detalla el módulo de aprendizaje y, por último, la sección 6.5 describe el estado actual de la interfaz gráfica de usuario.

6.1. Análisis y diseño de algoritmos

Un algoritmo es un conjunto ordenado de operaciones que resuelve un problema: recibe datos de entrada, sigue un proceso y proporciona datos de salida. En Carmen este esquema sencillo se ha alterarado para satisfacer algunos requisitos de control. El resultado ha sido un patrón de diseño de algoritmos que hemos aplicado a varios algoritmos de inferencia para modelos gráficos probabilistas.

6.1.1. Objetivos

Uno de los objetivos más importantes de la herramienta Carmen es que un nuevo programador que debe hacer alguna aportación no tenga dificultades debidas a la

complejidad del modelo de diseño.

Un requisito inicial era que Carmen permitiera construir y evaluar modelos probabilistas a través de una interfaz gráfica de usuario. Durante esta interacción la herramienta tiene que ser capaz de:

- Detectar los errores cometidos por el usuario.
- Permitir que el usuario deshaga y rehaga los últimos cambios.
- Permitir la ejecución paso a paso de los algoritmos, por motivos didácticos.

Además de los problemas de la interactividad es necesario integrar varios procesos que van a operar sobre una misma estructura de datos, lo cual requiere coordinar dichos procesos y de garantizar la consistencia de las estructuras de datos.

La última restricción sobre el diseño es que cada algoritmo debería poder dejar constancia en un registro (*log*) de las operaciones que se van realizando y el nivel de detalle de esa descripción debe ser configurable.

6.1.2. Análisis de un algoritmo genérico

Un algoritmo (por ejemplo, de inferencia o de aprendizaje) puede ser invocado por la interfaz gráfica de usuario o por un programa que manipule el algoritmo como si fuera una librería. Esta clase que invoca la modelamos como *Executor*.

Otra cuestión es decidir cómo gestionar la capacidad de rehacer y deshacer los pasos de un algoritmo. Para aislar la ejecución del algoritmo propiamente y la gestión (control) del algoritmo, hemos decidido que la clase *Algorithm* sea responsable de realizar la inferencia o el aprendizaje y que una clase *Undo/Redo* controle cuándo se realizan los pasos de los que consta el algoritmo.

Vamos a considerar también, que cada uno de los pasos en los que se divide un algoritmo se codifican como *ediciones*, representadas por la clase *Edit*. Para cada una de estas ediciones existirá la posibilidad de hacerse (modificando *ProbNet*), deshacerse (restaurando *ProbNet* a su estado previo) o de rehacerse (modificando de nuevo *ProbNet*).

La responsabilidad de crear las ediciones corresponde a la clase *Algorithm* y la de controlar su ejecución a la clase *Undo/Redo*.

Para dejar un registro de las actividades hemos creado la clase *Log*, que simplemente se encarga de escribir en un fichero de texto o en pantalla los pasos que se vayan ejecutando.

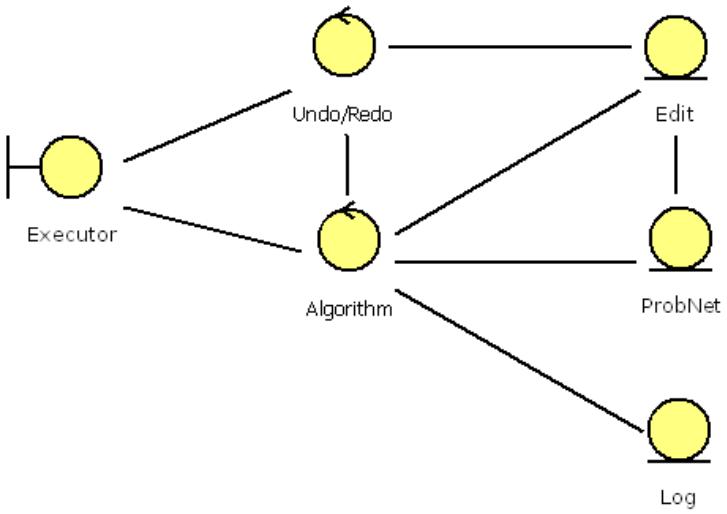


Figura 6.1: Modelo de clases de análisis para un algoritmo genérico.

6.1.3. Modelo de diseño

La inferencia/aprendizaje en Carmen es un componente con suficiente entidad como para describir a parte su arquitectura. Como dijimos en la sección 4.3, el diseño está dividido en dos partes: diseño arquitectónico y diseño detallado. Una arquitectura se describe con tres modelos o vistas:

a) *Vista estática*.

b) *Vista funcional*.

c) *Vista dinámica*.

Tambien dijimos que en el diseño lo mejor es encontrar un patrón (solución ya probada para un problema específico de diseño, véase (25; 17)) que sea aplicable. Aunque existen numerosos patrones arquitectónicos, ninguno de ellos es adecuado para describir la arquitectura de Carmen; por ellos vamos a describir cada una de estas tres vistas usando los diagramas habituales del UML.

a) Vista estática

Los paquetes relativos a los algoritmos son los reflejados en la figura 6.2.

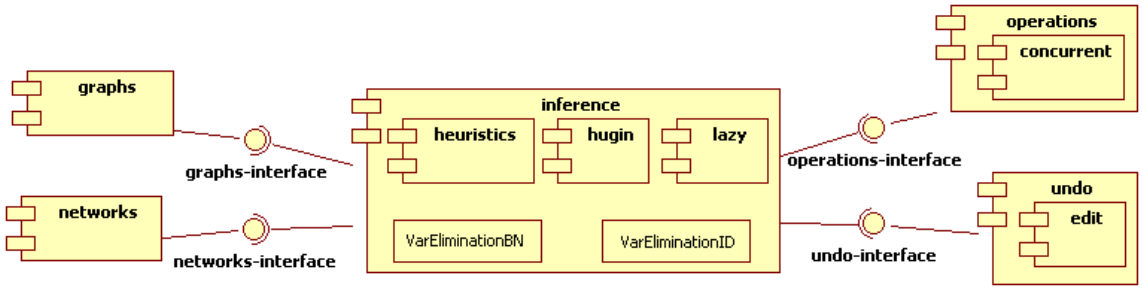


Figura 6.2: Modelo de diseño arquitectónico para los algoritmos de inferencia: vista estática.

b) Vista funcional

Los algoritmos están codificados en el paquete *inference*. Cada algoritmo se implementa como una clase cuando las estructuras de datos que utilizan y su lógica del algoritmo son lo suficientemente sencillos; éste es el caso de *VarEliminationBN* y *VarEliminationID*. Cuando son más complejas se componen de varias clases, como en el caso de los algoritmos *Clustering* y *LazyPropagation*, y se codifican en paquetes específicos.

Por otra parte, existe un conjunto de heurísticas que se emplean para obtener la secuencia de eliminación de variables, usadas por los algoritmos mencionados. Las heurísticas tienen una interfaz común, la cual permite que sean manejadas de una forma homogénea.

Las tres heurísticas que hemos implementado son las siguientes:

1. *Relleno mínimo* (en inglés, *minimal fill-in (79)*) que es una de las más simples y conocidas.

2. *Heurística de Cano-Moral* (11), que proporciona órdenes de eliminación eficientes con un coste computacional reducido.

3. *Heurística prefijada*: consiste en leer de un archvo el orden de eliminación; no es en realidad una heurística, pero realiza la misma función y proporciona al algoritmo de inferencia un orden de eliminación. Esta "heurística" es útil para comparar distintas herramientas. Por ejemplo, nosotros la hemos utilizado para forzar a Carmen a utilizar el mismo orden de eliminación que Hugin, véase (3).

El paquete *operations* codifica las operaciones elementales con potenciales (suma, multiplicación, marginalización y división) utilizando el método de los desplazamientos acumulados, que hemos descrito en el capítulo 2.

Un subpaquete de *operations* es *concurrent*, que permite operaciones básicas en paralelo. Este paquete no ha dado el rendimiento que esperábamos debido a varios problemas del modelo de memoria de Java y de la arquitectura hardware del sistema, por lo cual no lo vamos a describir con más detalle en esta memoria; lo mencionaremos en la sección 7.2.2 al hablar de las futuras líneas de investigación.

El paquete *undo* se encarga de gestionar las operaciones de hacer, deshacer y rehacer. Los algoritmos definen sus pasos como *ediciones*. Cada edición se codifica en una clase, que implementa la interfaz *PNEdit* y está definida en el paquete *edit*. Describiremos todo esto en detalle más adelante.

Los paquetes *graphs* y *networks* ya se han descrito en el capítulo anterior.

c) Vista dinámica

Las tres fases de un algoritmo genérico son: *inicio*, *operación* y *finalización*.

- En la fase de inicio se crean los objetos necesarios para la gestión de la inferencia.
- En la fase de operación el algoritmo se ejecuta siendo controlado por un *Executor*, por ejemplo, la interfaz gráfica y usando los servicios proporcionados por los paquetes *networks, operations* y *heuristics*.
- En la fase de finalización el objeto ejecutor recibe los resultados del algoritmo.

Control En la arquitectura, el control es la forma de decidir cuándo y cómo un módulo proporciona sus servicios. Existen dos estilos: centralizado y basado en eventos. El control centralizado supone una organización piramidal de los módulos tal que los que están en la cima controlan a los demás; este estilo es propio de la programación estructurada. En el control basado en eventos cada subsistema puede responder a eventos generados externamente; es propio de la programación orientada a objetos.

En el control basado en eventos cada componente posee varios fragmentos de código llamados *manejadores de eventos*. Los manejadores son llamados por un *despachador de eventos*, que se encarga de decidir qué componente maneja cada evento. Todo este esquema está basado en el concepto de invocación implícita, que consiste en que un componente genera un evento y aquellos componentes que han registrado su interés reciben dicho evento invocando el manejador asociado. El patrón de diseño más conocido de este tipo se denomina *Listener*.

6.1.4. Diseño detallado

Esta parte está fuertemente ligada al modelo de diseño de los MGPs, sobre todo por lo que respecta a las restricciones que pueden tener asociadas. En general, se entiende por *edición* cualquier cambio que se opere sobre una estructura de datos. Una edición se puede realizar de dos formas: cambiando directamente la estructura de datos mediante los métodos de las clases involucradas, o bien codificando los posibles cambios en unas clases que hacen de intermediarias. Las ventajas de dada forma son los inconvenientes de la otra. La primera aproximación tiene dos ventajas: es más eficiente, por ser más directa, y es más sencilla de codificar. En cambio, las ventajas de la segunda son estas: se abstrae en un conjunto de operaciones expresadas en un lenguaje de alto nivel orientado a las estructuras de datos; permite realizar las funciones de deshacer-rehacer, es capaz de comprobar si una determinada edición es consistente con las restricciones definidas en el modelo; permite la ejecución de los algoritmos paso a paso desde la interfaz gráfica, etc.

Nosotros hemos elegido la segunda opción porque la pérdida de eficiencia es mínima[1] y, aunque requiere una labor de diseño muy cuidadosa, una vez creado el conjunto inicial de funciones, se cuenta con un super-lenguaje de manipulación de redes probabilistas

[1] El cuello de botella de la eficiencia está en la realización de las operaciones básicas sobre potenciales, descritas en el capítulo 2.

muy potente. Este lenguaje permite implementar nuevos algoritmos fácilmente y se puede ampliar sin modificar las ediciones existentes.

Al intentar resolver este problema, hemos visto la conveniencia de desarrollar un nuevo patrón de diseño, que denominamos *Permiso-Ejecución*, basado en dos patrones ya conocidos: *Observer* y *Command*.

a) Patrón *Observer*. Dado un objeto que puede cambiar de estado y otros que están interesados en los posibles cambios, este patrón define una forma de notificar los cambios. El objetivo es desacoplar los observadores (clientes) y el objeto observado. La forma de capturar los eventos de cambio es que cada observador se suscriba como oyente (*listener*).

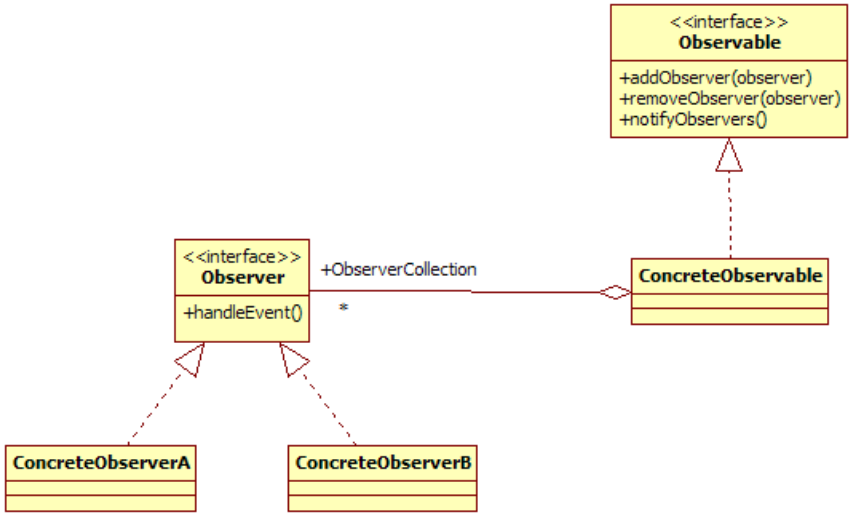


Figura 6.3: Diagrama de clases del patrón *Observer*.

Cada objeto *ConcreteObservable* contiene una colección de *Observer*s. Cuando ocurre un cambio sobre *ConcreteObservable*, este objeto se encarga de avisar de dicho cambio a todos sus *Observer*s invocando el método *notifyObservers()*.

La secuencia de eventos que ocurre con este patrón está indicada en la figura 6.4: el objeto observado (*ConcreteObservable*) se inicializa con los objetos interesados en sus cambios y cada vez que hace algo que modifica su estado envía un mensaje (mediante el método *notifyObservers*) para comunicar dicha modificación a los observadores que tiene almacenados. Cada observador actúa del modo que sea oportuno.

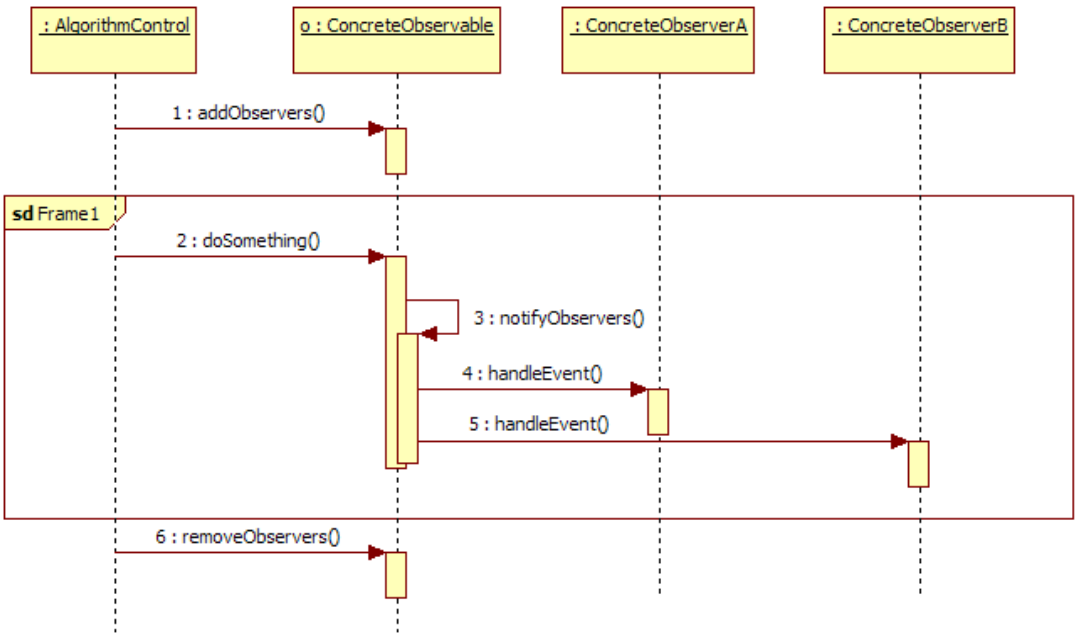


Figura 6.4: Diagrama de secuencias del patrón *Observer*. Se supone que el mensaje 2 (*doSomething()*) puede iterarse varias veces.

b) Patrón *Command*. Este patrón encapsula una instrucción en un objeto de forma que puede ser almacenado, pasado a métodos y devuelto al igual que cualquier otro objeto. Se utiliza generalmente para implementar las acciones de deshacer y rehacer, poner comandos en una cola y ejecutarlos en momentos distintos y para desacoplar la fuente de una petición del objeto que la ejecuta.

Lo más importante de este patrón es que separa el *cuándo* y el *cómo* en la ejecución de las acciones.

Existen dos variaciones del patrón *Command*. Una es para las acciones de deshacer y rehacer y otra para instrucciones compuestas de otras. En el diseño de algoritmos utilizaremos las dos.

b.1) Patrón *Commnad* para deshacer y rehacer. En este caso, es necesario que el objeto *Receiver* debe ser capaz de invertir la acción realizada, lo cual supone que tiene que guardar la información necesaria de tantas acciones como se desee poder deshacer. Lo más lógico es guardar las acciones en una pila, para poder deshacerlas en el orden inverso.

Por otra parte, puede ocurrir que un algoritmo se ejecute sin posibilidad de deshacer

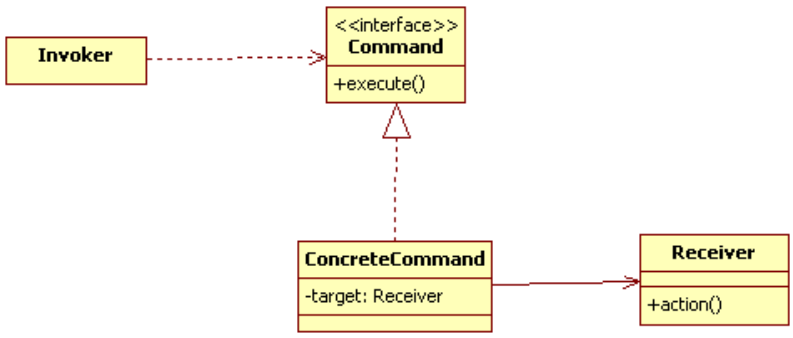


Figura 6.5: Diagrama de clases del patrón *Command*.

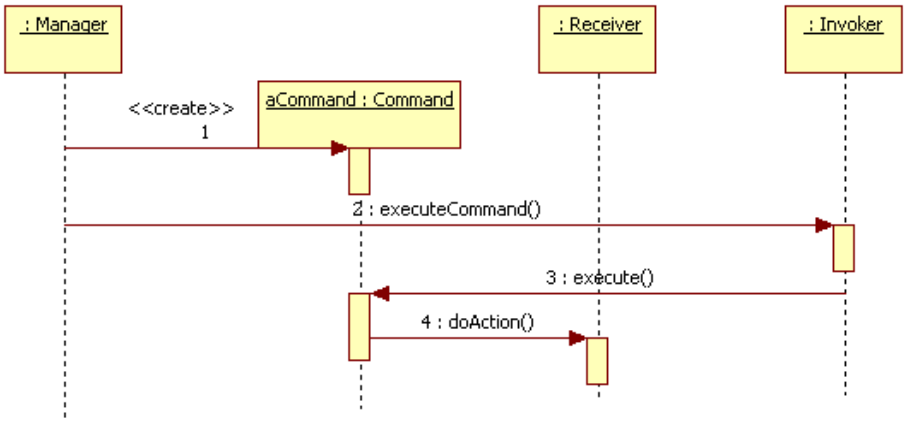


Figura 6.6: Diagrama de secuencias del patrón *Command*.

los pasos. Este modo de ejecución va a ser más eficiente, al menos en cuanto al uso de memoria porque necesita guardar información de las acciones realizadas.

b.2) Patrón *Command* para acciones compuestas de otras Esta modificación se conoce como *MacroCommand*. Se utiliza en el caso de acciones complejas que se tienen que hacer o todas o ninguna. Un objeto de este tipo contiene una lista de subcomandos; el método *execute()* se encarga de invocar a todos los subcomandos. Cuando se deshace el comando, la llamada a los hijos se hace en el orden inverso.

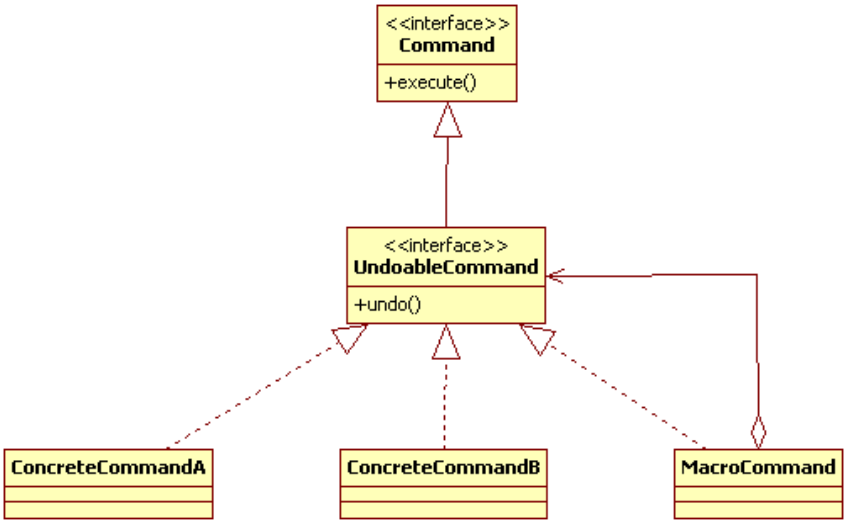


Figura 6.7: Patrón Command con las dos modificaciones para rehacer y deshacer, y acciones compuestas.

6.1.5. Patrón de diseño *Permiso-Ejecución*

Para desarrollar este patrón tenemos que hacer primero la siguiente modificación del patrón *Observer*:

Puede darse la circunstancia de que se intente hacer un cambio sobre el objeto observado que no esté permitido por cierta restricción; un ejemplo es lo que ocurre en la interfaz de usuario cuando un usuario intenta asignar el mismo nombre a dos variables distintas. Este caso se puede tratar dividiendo el patrón anterior en dos fases: en una primera fase el objeto observado anuncia su intención de hacer un determinado cambio. Los objetos oyentes pueden vetar ese cambio si no lo consideran correcto en cuyo caso

no se realiza. Si no hay veto por parte de ningún objeto oyente, el objeto observado actúa como en el caso anterior: realizando el cambio y anunciándolo a los objetos oyentes para que actúen como estaba previsto.

Este cambio que hemos introducido en el patrón *Observer* implica que en el diagrama de clases hay que sustituir el método *handleEvent* por los métodos *announceChange*, que se invoca antes de hacer el cambio (por ejemplo, para dar la posibilidad de que el experto vete el cambio) y *changeHappened*, que se invoca después del cambio (por ejemplo, para que el oyente actualice su información sobre el objeto observado).

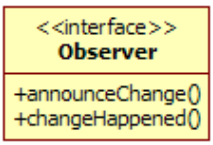


Figura 6.8: Cambios en la interfaz de Observer.

El diagrama de secuencias queda modificado como se indica en la figura 6.9. Siguiendo el esquema definido en el patrón *Command*, un algoritmo se divide en pasos elementales que llamamos *ediciones*. Una edición se codifica como una clase e implementa los métodos *doEdit()* y *undo()* (a los que únicamente hemos cambiado el nombre respecto al patrón Command) definidos en la interfaz *PNEdit*[2]. Una clase que implemente la interfaz *PNEdit* es responsable de hacer y deshacer la edición que codifique y guardar toda la información adicional. La consecuencia de este esquema desde el punto de vista del programador es que un algoritmo se va a dividir en varias clases, pero cada una de ellas no es más compleja que la parte del algoritmo que implementa.

Por otra parte, y siguiendo el esquema definido en el patrón *Observer*, en el caso de modelos gráficos probabilistas se trata con estructuras de datos bastante complejas (hay muchas clases de distinto tipo involucradas en la ejecución de un algoritmo). Esta complejidad se ha tratado aislando los diferentes aspectos en diferentes participantes. Los participantes son: el algoritmo, la red probabilista sobre la que opera, la interfaz

[2]**P**robabilistic **N**etwork Edit

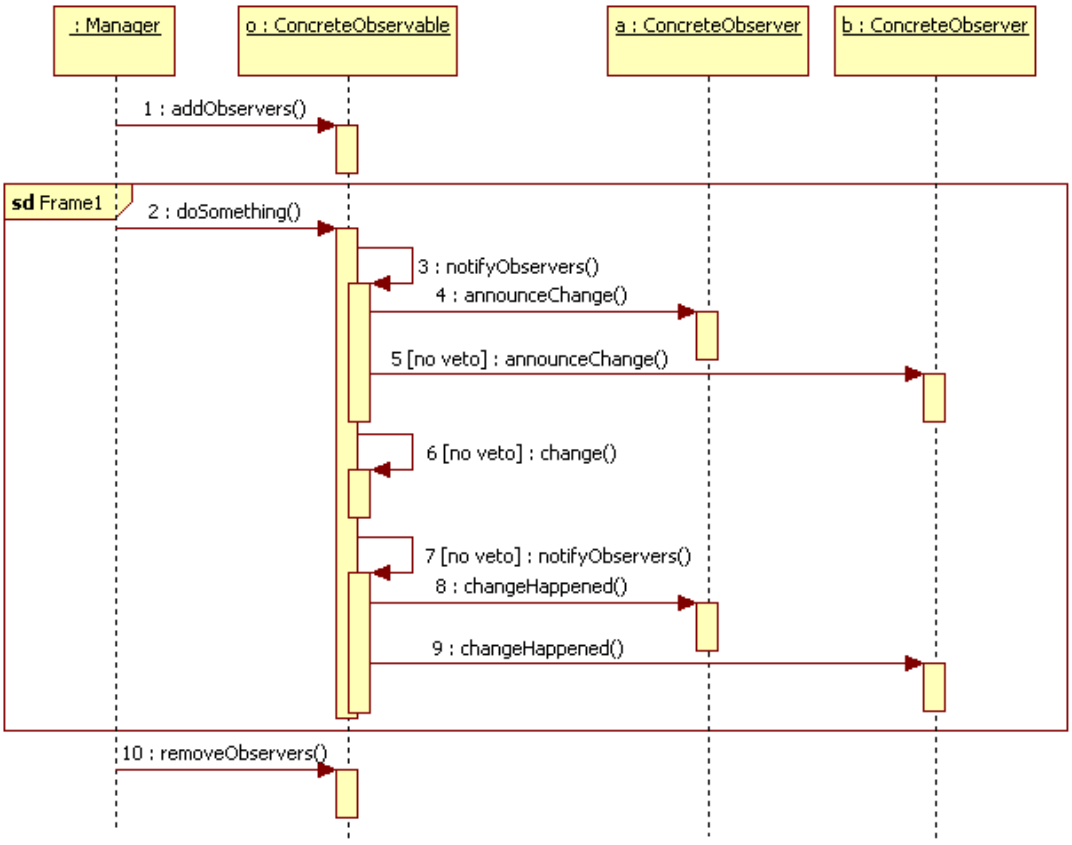


Figura 6.9: Patrón observer de permiso-ejecución.

gráfica que representa la red, la heurística que usa las características del grafo, etc. Cada participante hace una copia de la información que utiliza y se registra en un objeto de la clase *PNESupport*[3]. Cada vez que un participante desea modificar algo se lo indica al objeto de la clase *PNESupport*, que retransmite ese deseo al resto de participantes. Si ninguno se opone se realiza el cambio.

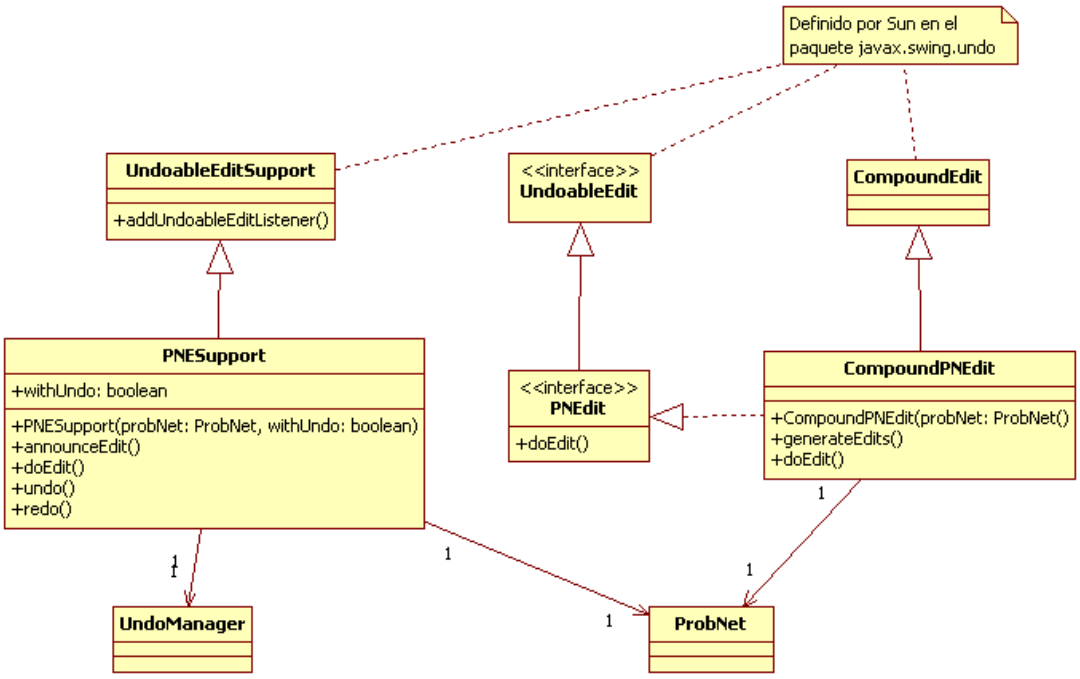


Figura 6.10: Clases e interfaces en el paquete *carmen.undo*.

Algunas aclaraciones sobre el párrafo anterior:

- Existe un único objeto del tipo *PNESupport* al que todos los participantes pueden acceder.
- El objeto *PNESupport* es el que se encarga de gestionar la realización de ediciones.
- Expresar el deseo de realizar una modificación significa que se crea el objeto asociado a la edición correspondiente a la acción y se pide al objeto de tipo *PNESupport* que se encargue de anunciarlo.

[3] **P**robabilistic **N**etwork **E**dit Support

El paquete *undo* tiene muchas más clases que las indicadas en la figura 6.10 y las clases tienen más métodos que los indicados. Las clases y los métodos que se han incluido son lo esencial para entender el funcionamiento de las ediciones.

a) Modelo de diseño de un algoritmo

Este diseño lo vamos a expresar desde dos puntos de vista: estático y dinámico. El punto de vista estático representa los componentes de los que está formado y sus relaciones; usaremos un diagrama de clases. El punto de vista dinámico representa la interacción de esos componentes para realizar una función; usaremos un diagrama de secuencias. Mostraremos un ejemplo sencillo pero no trivial del diseño de un algoritmo genérico.

a.1) Punto de vista estático En el caso de Carmen un algoritmo va a operar sobre un modelo probabilista. Podemos representar estos modelos por la clase *ProbNet* que es la clase antecesora de todas ellas. En un principio pensamos en crear una interfaz común para todos los algoritmos con un método llamado *run()*, pero como cada uno es diferente y a veces se puede invocar de distintos modos al final desistimos de esa idea. Para simplificar representaremos la interfaz de usuario como un paquete llamado *gui*.

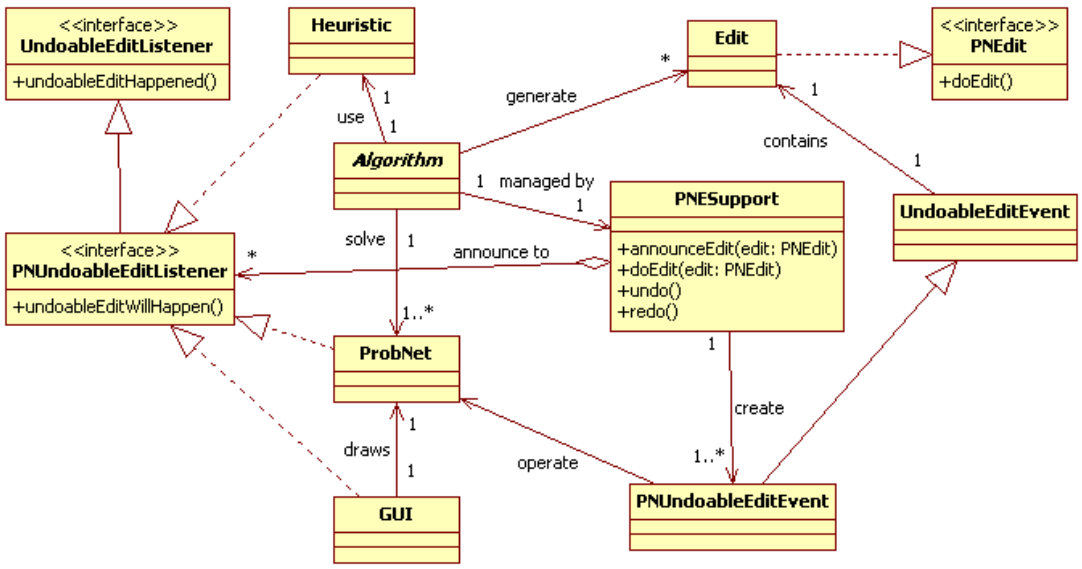


Figura 6.11: Diagrama de clases de un algoritmo genérico.

Cada uno de los diferentes participantes hace una copia de la red probabilista recibida (*ProbNet*). Cada copia está adaptada a las características del participante: *AlgorithmNet* para *Algoritmo*, *GraphicsNet* para la interfaz gráfica (*GUI*) y *HeuristicNet* para *Heuristic*. Cuando se reciben los eventos (ediciones) generados por el algoritmo las copias son actualizadas. En el diagrama de clases de la figura 6.11 es la clase *Algoritmo* la que va a actuar como objeto observado y las clases *GUI* y *Heuristica* como observadores. Por ese motivo la clase *Algoritmo* es gestionado por la clase *PNESupport* y las clases *GUI* y *Heuristica* implementan la interfaz *PNUndoableEditListener* para escuchar los eventos de cambio y actualizar sus copias locales.

El motivo de que cada participante tenga su propia copia es que cada participante puede haber adaptado la red a las características específicas del proceso que lleva a cabo; por ejemplo, una heurística de eliminación de variables puede ser que trabaje con grafos no dirigidos, la interfaz gráfica maneja datos tales como la posición en pantalla de cada nodo y el algoritmo crea también una copia adaptada a las características del proceso, por ejemplo, en el método de propagación de Hugin una red se convierte en un árbol de unión con una topología y unas tablas de probabilidades totalmente diferentes a las de la red original que puede ser que interese guardar en otro sitio.

Esta decisión de hacer copias de una misma red adaptadas a las necesidades de cada participante tiene la ventaja de ser más fácilmente modificable. El único inconveniente es el mayor consumo de memoria, que es despreciable porque lo que realmente ocupa espacio son las tablas de probabilidades que en ningún caso es necesario duplicar.

a.2) Punto de vista dinámico Ahora veremos cómo es el esquema anterior durante la ejecución del algoritmo con un diagrama de secuencias.

Un algoritmo es iniciado por algún objeto, por ejemplo la interfaz gráfica. La ejecución de un algoritmo tiene lugar en tres fases: lo primero que hace un objeto algoritmo es crear los objetos necesarios para su funcionamiento; por ejemplo: una heurística de eliminación, un objeto del tipo *PNESupport* si no existe y una copia de la red probabilista sobre la que opera. Además de crear dichos objetos, aquellos objetos interesados en las operaciones del algoritmo se registran en el objeto *PNESupport*. La segunda fase es la de operación: cada vez que el algoritmo va a realizar una acción crea el objeto del tipo *PNEdit* correspondiente a dicha acción. Después se lo comunica al objeto *PNESupport*, que se encarga de comprobar si hay algún veto. Si no lo hay se envía un mensaje a *PNESupport* para que realize la acción.

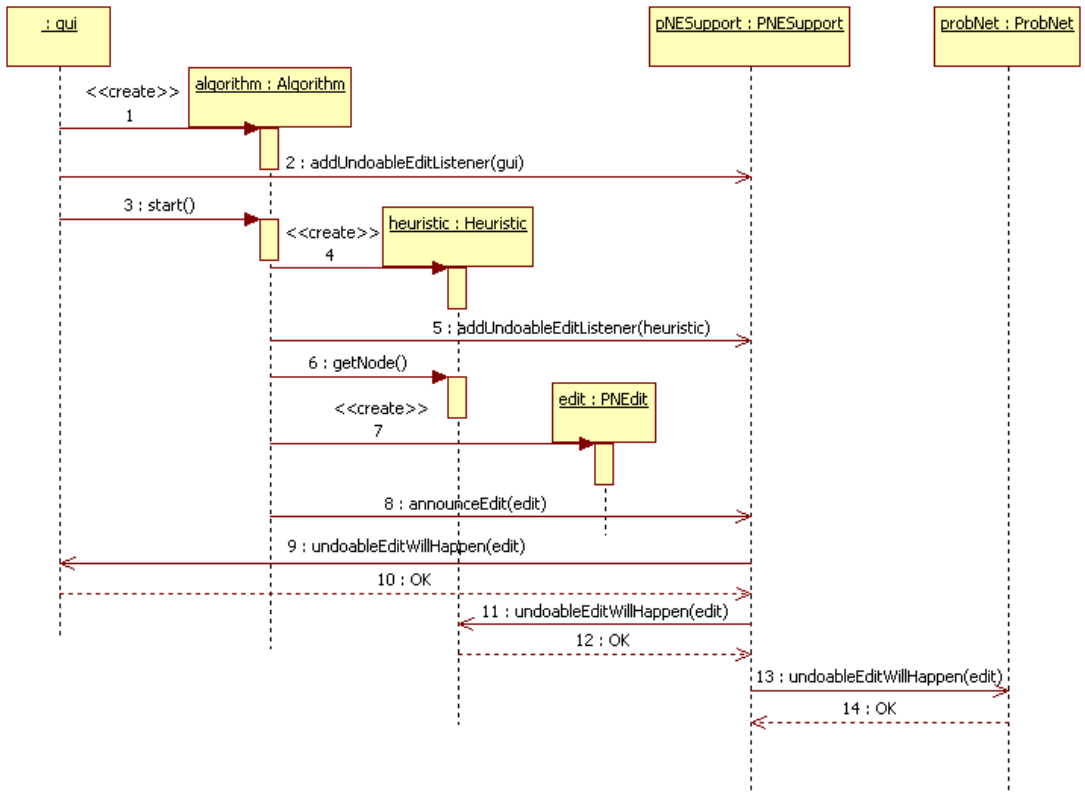


Figura 6.12: Diagrama de secuencias simplificado del funcionamiento de un algoritmo.

En la figura 6.2.1 no se han incluido los objetos del tipo red probabilista para no enmarañar el diagrama con mensajes innecesarios. Hay que tener en cuenta que en esta sección todavía no se ha visto ningún ejemplo real porque su finalidad es exponer una idea general.

El problema general de los diagramas de UML es que cuando algo adquiere una mínima complejidad aumentan mucho de tamaño. Los diagramas expuestos en las secciones siguientes tienen todo lo necesario para comprender los algoritmos, pero se han resumido para que sean manejables.

b) Heurísticas de eliminación de nodos

La mayor parte de los algoritmos de inferencia en MGPs utilizan una heurística que determina el orden de eliminación de las variables. Algunos de los algoritmos de inferencia añaden enlaces hasta que el grafo queda triangulado (véase 1.1.1). A estas heurísticas se las llama *heurísticas de triangulación*.

Cuando se elimina una variable se crean enlaces entre los vecinos de dicha variable (en caso de que no existieran previamente). Hablando *grosso modo*, mejor cuanto menores sean los conglomerados que se van creando. El problema de encontrar la mejor secuencia de eliminación de variables es un problema NP-completo para la mayor parte de los algoritmos, véase (4), por este motivo se utilizan heurísticas.

Las heurísticas proponen la eliminación de un nodo, pero se contempla la posibilidad de que el algotitmo pueda seleccionar otro nodo, en vez de o además del que ha propuesto la heurística. Por este motivo, cada heurística funciona como un *listener*:[4] primero la heurística es consultada sobre qué nodo propone eliminar y luego se le informa de qué nodo o nodos han sido realmente eliminados.

Las heurísticas que hemos implementado son:

1. *SimpleElimination*, que elige el nodo con menos vecinos. Hay dos versiones, para redes bayesianas y para diagramas de influencia (*SimpleEliminationID*).

2. *MinimalFillIn*, que escoge el nodo que crea menos enlaces en su eliminación.

[4]Un listener funciona de un modo parecido a una lista de distribución: un objeto genera un mensaje y los que están apuntados a la lista lo reciben.

3. *FileElimination*, que no es propiamente una heurística, pero se utiliza del mismo modo: escoge los nodos según el orden indicado en un fichero de texo. Esta heurística es útil para comparar diferentes herramientas forzando cierto orden de eliminación.

4. *CanoMoralElimination*, que implementa la heurística definida por Andrés Cano y Serafín Moral (11), que es una de las más eficientes que existen actualmente.

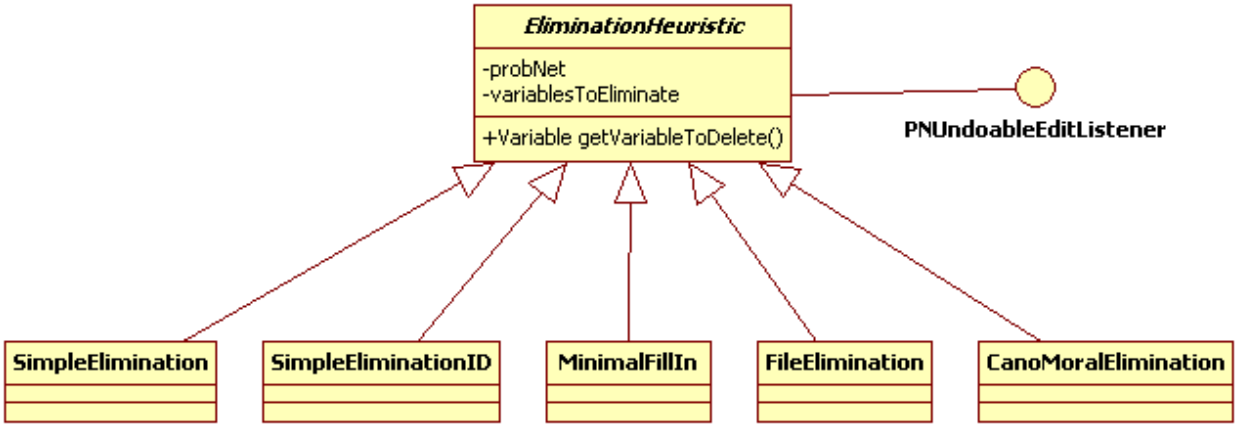


Figura 6.13: Heurísticas disponibles en Carmen.

6.2. Inferencia en redes bayesianas

6.2.1. Eliminación de variables para redes bayesianas

En este punto vamos a ver una realización concreta de todas las ideas desarrolladas en la sección 6.1 aplicada al algoritmo de eliminación de variables para redes bayesianas. Comentaremos el diseño detallado de la implementación del algoritmo.

Este algoritmo está diseñado para encontrar la distribución de probabilidad de *una* variable dada la evidencia. Se ha adaptado para operar sobre varias variables aplicando el algoritmo repetidas veces, una por cada variable.

Dado que el algoritmo opera podando los nodos de la red que no son relevantes dada la evidencia (usando las propiedades de *d*-separación), la fase de poda la hemos realizado en dos pasos: primero hemos podado los nodos *d*-separados de *todos* los nodos de interés

dada la evidencia y a la subred resultante le hemos aplicado el algoritmo de eliminación de variables original podando otra vez para cada nodo de interés.

Siguiendo este esquema, veremos los diagramas de clases y de secuencias que ilustran el aspecto estático y el dinámico del problema.

a) Punto de vista estático

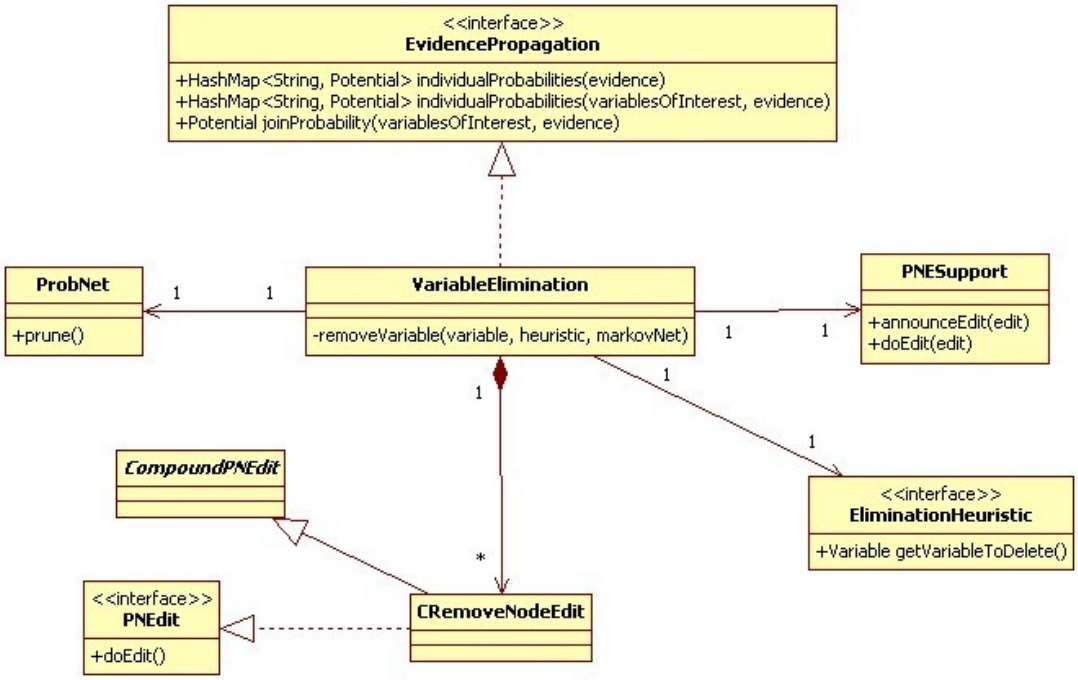


Figura 6.14: Clases e interfaces relativas a la eliminación de variables en redes bayesianas.

Los métodos principales que controlan el funcionamiento del algoritmo son los que permiten conocer las distribuciones de probabilidad conjunta o individual de las variables de interés, definidos en la interfaz *EvidencePropagation*:

- *individualProbabilities(evidence)*
- *individualProbabilities(variablesOfInterest, evidence)*
- *joinProbability(variablesOfInterest, evidence)*

La heurística es cualquier objeto que implemente la interfaz *EliminationHeuristic*, que proporciona un orden de eliminación de las variables de la red. Se manipula con el método *getVariableToDelete()*.

La evidencia está representada en un objeto del tipo *EvidenceCase*, que se compone de un conjunto de *Finding*s.

La función del objeto *PNESupport* es controlar las ediciones que operan sobre la red, que son siempre eliminaciones de variables. No se definen para este algoritmo objetos que veten la eliminación de una variable y, en consecuencia, después de anunciar cada edición, se lleva a cabo.

La red sobre la que opera el algoritmo *ProbNet* es una copia de la red que se recibe. De este modo no se pierde la red original.

La eliminación de una variable supone un proceso que consta de varias ediciones más simples:

- Combinar los potenciales que contienen la variable asociada al nodo que se va a borrar multiplicándolos y marginalizando dicha variable.
- Borrar los potenciales que se han utilizado.
- Crear enlaces, si no existen, entre los nodos asociados al que se va a borrar.
- Borrar los enlaces del nodo.
- Borrar el nodo.

Cada uno de los puntos anteriores se ha codificado como ediciones sencillas: *AddPotentialEdit*, *RemovePotentialEdit*, *AddLinkEdit*, *RemoveLinkEdit* y *RemoveNodeEdit*. Estas ediciones sencillas se tienen que realizar como un todo, porque, si el proceso se queda a medias, las estructuras de datos que representan la red quedarían en un estado inconsistente. La clase que codifica todo el proceso es *CRemoveNodeEdit*, que es una edición compuesta.

En el diagrama de clases de la figura 6.14 se han omitido la mayor parte de los atributos y de los métodos de las clases para no complicar demasiado el esquema.

b) Punto de vista dinámico

Vamos a suponer que el algoritmo es lanzado por un objeto llamado *executor*, que puede ser la interfaz gráfica, una clase de pruebas, etc. Al igual que en el diagrama anterior, obviaremos lo no esencial, que en este caso significa que no representaremos la mayor parte de los objetos del diagrama de la figura 6.14.

El algoritmo se puede manipular con tres métodos, de los cuales, el más completo, porque llama a los otros dos, es *individualProbabilities(evidence)*. En el diagrama de secuencias supondremos que se invoca a ese método. *individualProbabilities(evidence)* considera que todas las variables son variables de interés.

El algoritmo es sencillo pero se crean muchos objetos antes de empezar a operar. Hemos considerado más práctico dividir su presentación en dos fases: lanzamiento y operación.

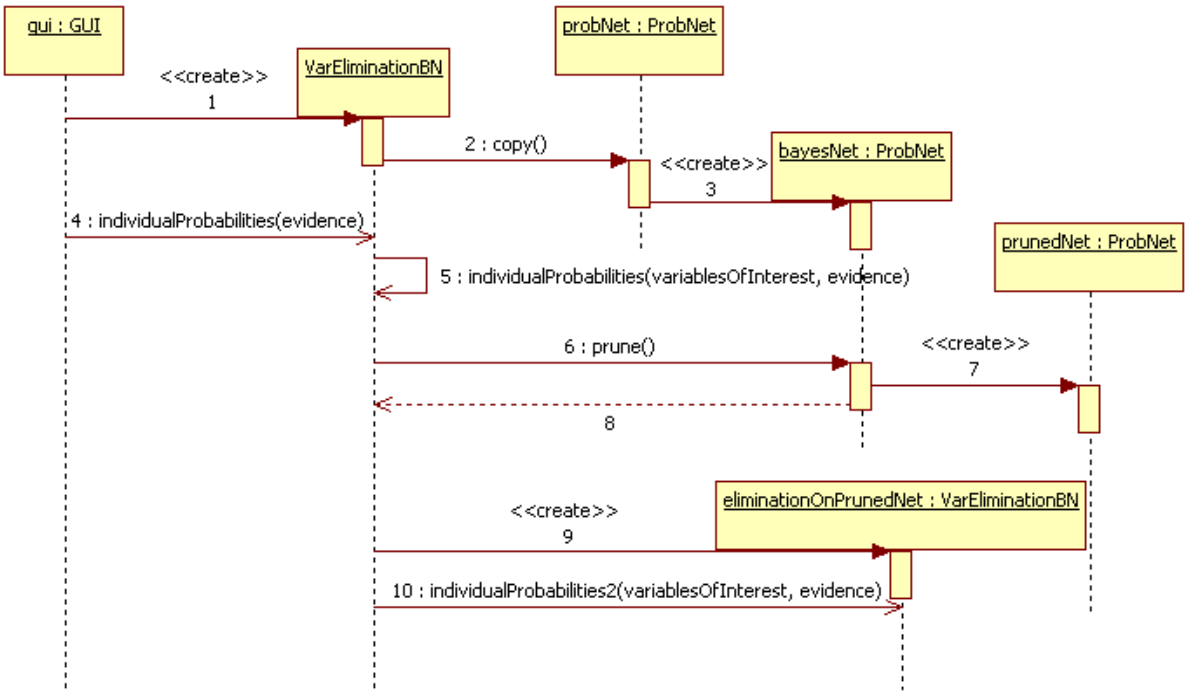


Figura 6.15: Lanzamiento del algoritmo de eliminación de variables para redes bayesianas.

b.1) Fase de lanzamiento En la figura 6.15 se describe esta situación: durante el lanzamiento se crea un objeto del tipo *VarEliminationBN* que copia la red recibida en otra para poder realizar cambios en su estructura.

Se invoca el método *individualProbabilities(variablesOfInterest, evidence)* poniendo como variables de interés todas las de la red excepto las variables que han recibido evidencia.

El método *individualProbabilities(variablesOfInterest, evidence)* poda la red copiada

en función de la evidencia y de las variables de interés y se crea un nuevo objeto del tipo *VarEliminationBN* que va a operar sobre la red podada.

Se invoca al método *individualProbabilities2(variablesOfInterest, evidence)*, que va a resolver el problema.

b.2) Fase de operación Partimos del momento en el que un objeto del tipo *VarEliminationBN* va a operar sobre una red podada para la evidencia recibida. Hay dos formas de invocar a este método, según se especifica en la interfaz *EvidencePropagation*: pedir la distribución de probabilidad de cada una de las variables de interés $\{P(x_1), P(x_2), \ldots, P(X_n)\}$ y pedir la distribución conjunta, $P(x_1, x_2, \ldots, x_n)$. En cada caso se procede de formas distintas.

En el primer caso se itera sobre cada variable de interés. La red podada se vuelve a copiar en cada iteración y se poda otra vez dicha copia en función de la única variable de interés de cada iteración. Esto significa que la poda de la red se realiza en dos fases: una poda general para todas las variables de interés y posteriormente otra poda sobre copias de la red anterior para cada variable. La primera poda es la mínima que siempre hay que hacer, y las siguientes son de menor tamaño, esto es más eficiente que hacer varias podas partiendo todas ellas de la red completa original.

En el segundo caso, es decir, al calcular la probabilidad conjunta, la red sólo se poda una vez. Cuando se tiene una red podada es cuando empieza realmente el método de eliminación de variables. El algoritmo es muy sencillo, véase la figura 6.16: se proyecta la evidencia sobre los potenciales que quedan en la red podada, se construye una red de Markov con dichos potenciales, se crea una heurística de eliminación y se eliminan una a una las variables que no son de interés.

6.2.2. Métodos de agrupamiento

Los métodos de agrupamiento obtienen la distribución de un conjunto de variables y son más rápidos que los anteriores. Existen varios tipos de algoritmos, véase (44), que trabajan sobre grupos (clusters), como el método de Hugin, el de Shafer-Shenoy o la propagación perezosa. Todos los métodos de agrupamiento tienen una parte en común. Se ha aíslado esa parte común utilizando el patrón de diseño *Template Method*. El resultado ha sido que una vez creada la parte común, la especialización en los métodos de Hugin (36), Shafer-Shenoy (68) y propagación perezosa (52) ha sido muy sencilla.

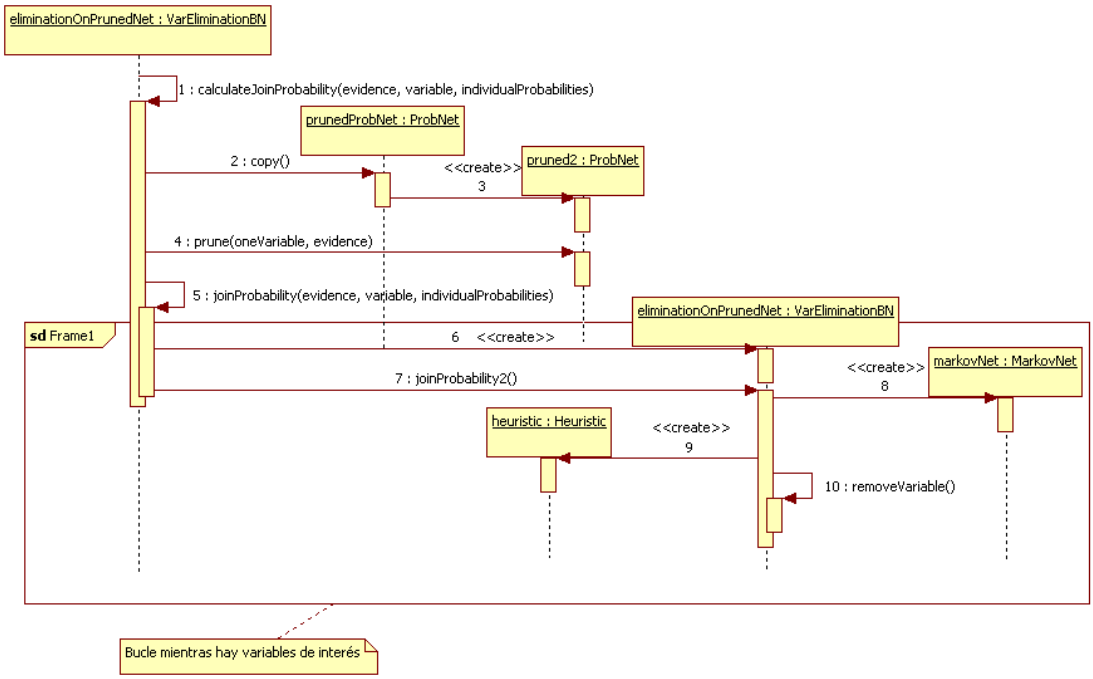


Figura 6.16: Fase de poda y operaciones del algoritmo de eliminación de variables para redes bayesianas. Se han eliminado los mensajes menos relevantes.

La estructura de datos que se utiliza en estos métodos es el bosque de grupos, que es un grafo dirigido acíclico formado por varios árboles. Cada nodo de este grafo representa un conjunto de variables. Un separador, es la intersección de los conjuntos de variables asociados a los nodos unidos por ese enlaceun conjunto de variables formado por las variables que tienen en común los dos nodos.

Algoritmo 6: Algoritmo general de agrupamiento

Entrada: Conjunto de potenciales (Φ) y heurística de eliminación
Resultado: Una distribución de probabilidad

1 **Mientras** *Quedan variables que no son de interés* **hacer**
2 La heurística selecciona una variable x_i.
3 Del conjunto total de potenciales (Φ_X) se seleccionan todos los potenciales Φ_{X_i} que contienen la variable X_i.
4 Se multiplican los potenciales de Φ_{X_i} y se elimina por suma la variable X_i obteniendo un potencial que no contiene x_i: $\Phi_{X \setminus X_i} = \sum_{x_i} \prod \Phi_{X_i}$.
5 Se eliminan los potenciales Φ_{X_i} de Φ.
6 Se añade el potencial $\Phi_{X \setminus x_i}$ al conjunto.
7 Se multiplican los potenciales que quedan en Φ para obtener un único potencial ϕ.
8 Se normaliza ϕ.

El potencial resultante del algoritmo 6 será de una variable o de varias. Nosotros hemos implementado una variante en la que construimos una red de Markov que guíe la heurística, por motivos de eficiencia.

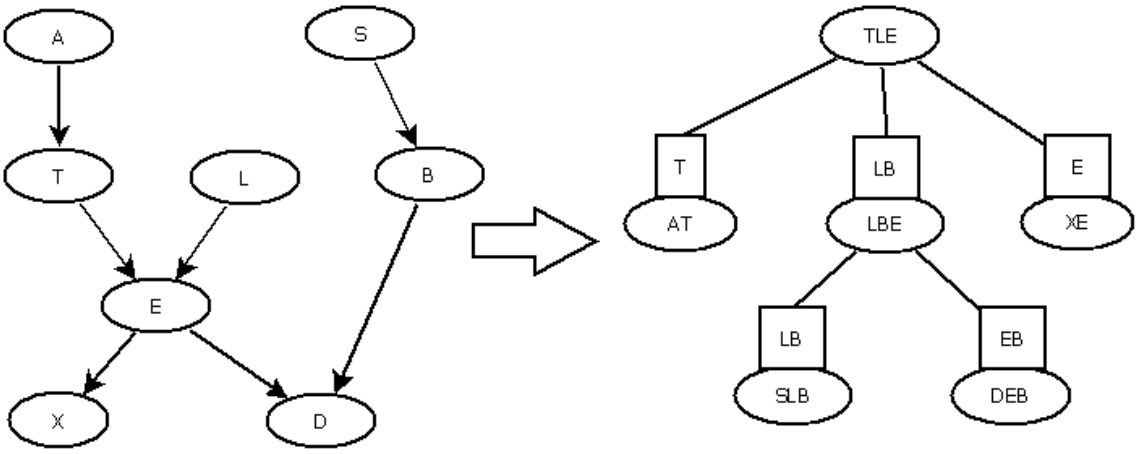


Figura 6.17: Conversión de una red bayesiana en un bosque de grupos.

La inferencia en los métodos de agrupamiento funciona intercambiando mensajes con los nodos vecinos. Un mensaje entre un nodo A y un nodo B es un potencial que pasa a través del separador S_{AB} y sólo puede contener las variables del separador. Un mensaje puede cruzar en la dirección $A \rightarrow B$ o $B \rightarrow A$. En los métodos de inferencia suele haber una fase ascendente (de las hojas al nodo raíz) y una fase descendente (del nodo raíz a las hojas).

Por lo que respecta a los nodos, una decisión importante que afecta al rendimiento es si se guardan o no los mensajes que pasan en las fases descendente y ascendente de la propagación de evidencia. Existen tres opciones: no almacenar nada, almacenar sólo los mensajes ascendentes y almacenarlos todos. Se ha comprobado que no existen diferencias notables de rendimiento entre las dos opciones más extremas: no almacenar nada y almacenar todos los mensajes, aunque el uso de memoria aumenta en el segundo caso.

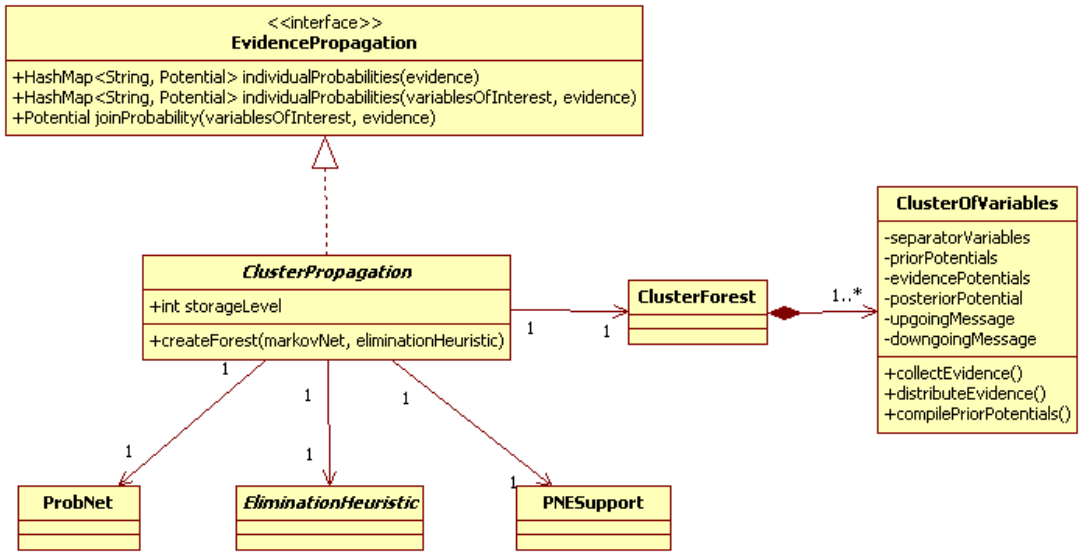


Figura 6.18: Vista estática: clases e interfaces necesarias para los métodos de agrupamiento.

En la clase *ClusterOfVariables* se almacenan los mensajes, el conjunto de variables de cada nodo y las variables de su separador. En la figura 6.17 hemos dibujado juntos los nodos y los separadores, aunque no es lo habitual, para reflejar la forma en la que han

sido implementados. El nodo raíz es el único que no tiene separador.

La parte algorítmica está aislada en la clase *ClusterPropagation*, es una clase abstracta que implementa la misma interfaz que el método de eliminación de variables (*EvidencePropagation*), y por lo tanto se puede manejar del mismo modo. Hemos utilizado el patrón de diseño *Template Method*, que define la estructura de un algoritmo en un método de una clase abstracta (*ClusterPropagation*), dejando la parte especializada de la implementación en las clases derivadas. Las subclases redefinen el algoritmo general (plantilla) sin cambiar su estructura.

El objetivo de este diseño es que se puedan implementar fácilmente nuevos métodos de agrupamiento con sólo definir las partes que cambian. Nosotros hemos implementado dos algoritmos: el método básico de agrupamiento y el de la propagación perezosa, que describimos en los siguientes párrafos.

Al igual que en el método de eliminación de variables en 6.2.1 también partimos de una heurística (*EliminationHeuristic*), un objeto que gestiona las ediciones (*PNESupport*) y una red probabilista (*ProbNet*) a partir de la cual se construye el *ClusterForest* (o un objeto de otra clase que herede de ella), que contiene varios *ClusterOfVariables* (o una instancia de otra clase que herede de ella).

a) Método básico de agrupamiento y método de Hugin

Desde el punto de vista de la implementación, el diseño es muy sencillo porque se utilizan las clases del apartado anterior y se crean las clases: *HuginForest, HuginClique* y *HuginPropagation*, que son especializaciones de las anteriores. La particularidad del método que hemos implementado es que no hay división de potenciales en la fase descendente.

a.1) Punto de vista estático En el diagrama de clases de la figura 6.21, la clase central es *HuginForest*, que sigue el mismo esquema básico definido en la sección 6.2.2. Comentaremos brevemente las diferencias.

La clase *HuginPropagation* lo único que hace es crear un objeto *HuginForest*. El constructor de *HuginForest*, partiendo de una red de Markov y de una heurística de

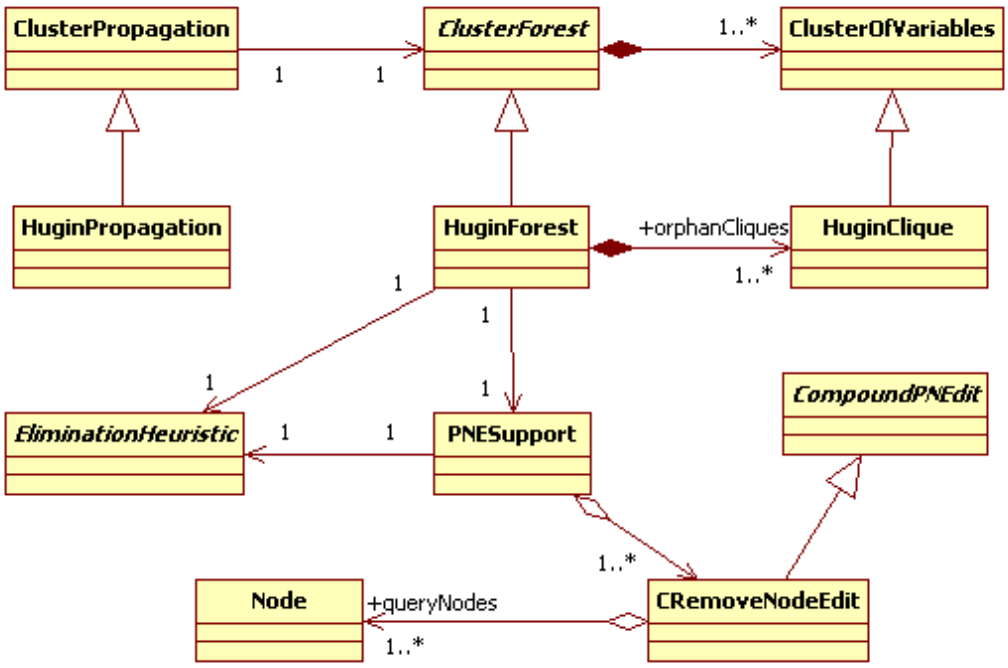


Figura 6.19: Vista estática: clases relativas al método de Hugin. Se han omitido los métodos para dar una visión más clara.

eliminación de variables, va borrando nodos de la red de Markov y creando el árbol de grupos, que en este caso son conglomerados (cliques). La diferencia entre grupo y conglomerado es que el conglomerado es maximal. Aunque únicamente el clique raíz no tiene padre, durante la construcción, y de un modo provisional, existe un conjunto de cliques huérfanos (*orphanCliques*) sin padre.

a.2) Punto de vista dinámico La figura 6.20 es el diagrama de secuencias resumido de la propagación de evidencia en nuestra implementación del método de Hugin. Después de crear un *HuginForest*, se introducen los potenciales de evidencia y se llama a los métodos *collectEvidence()* y *distributeEvidence()*.

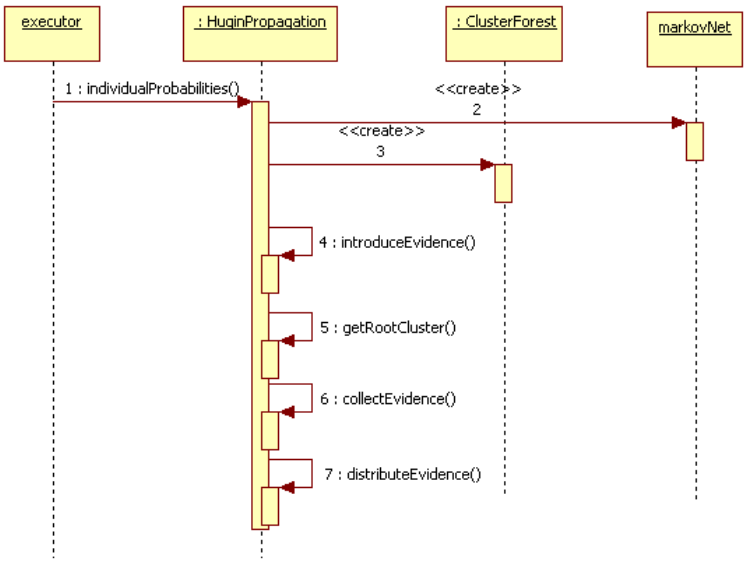


Figura 6.20: Vista dinámica resumida de nuestra implementación del método de Hugin.

b) Propagación perezosa

Una variante del método de Hugin es la propagación perezosa (52). Este método fue implementado por Carlos Baena, alumno de doctorado, bajo nuestra supervisión. El diseño es básicamente el mismo que para el método de Hugin y reutiliza por herencia las clases generales definidas para métodos de agrupamiento.

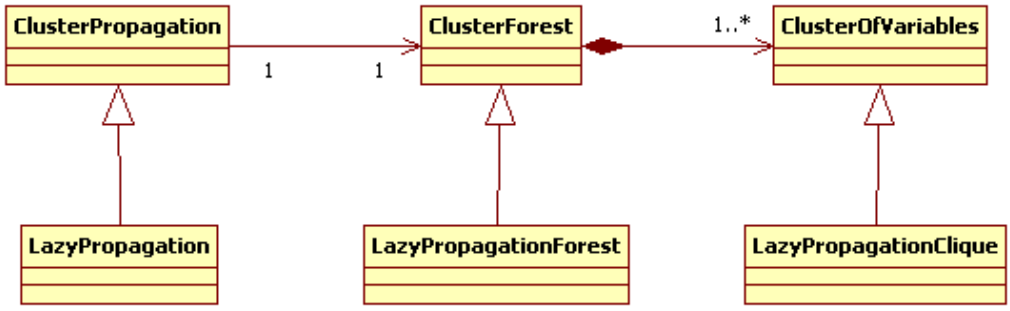


Figura 6.21: Clases utilizadas por la propagación perezosa.

6.2.3. Modelos canónicos

a) Representación de los modelos canónicos

Como se explicó en la sección 1.1.3, los modelos canónicos permiten reducir el tamaño de las tablas de probabilidades definiendo un potencial por cada uno de los padres en lugar de un único potencial que contenga todos los padres (22).

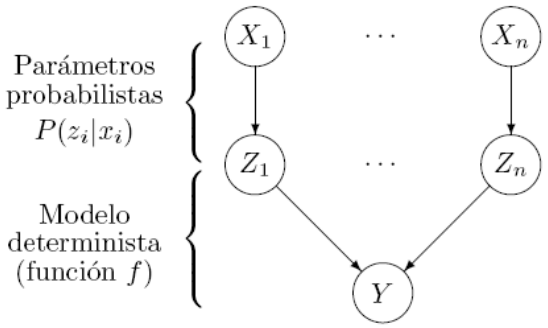


Figura 6.22: Estructura interna de un modelo canónico con ruido.

El potencial se construye añadiendo los subpotenciales necesarios. Cada enlace tiene asociados unos parámetros $P(z_i|x_i)$, donde las Z_i son unas variables auxiliares espcíficas de cada modelo (22). En el caso de un modelo residual (*leaky model*), hay además unos parámetros residuales, $P(z^L)$.

Como se puede ver en la figura 6.23, la clase central de estos modelos es *ICIPotential (Independence of Causal Interactions Potential)*, que al igual que todos los potenciales

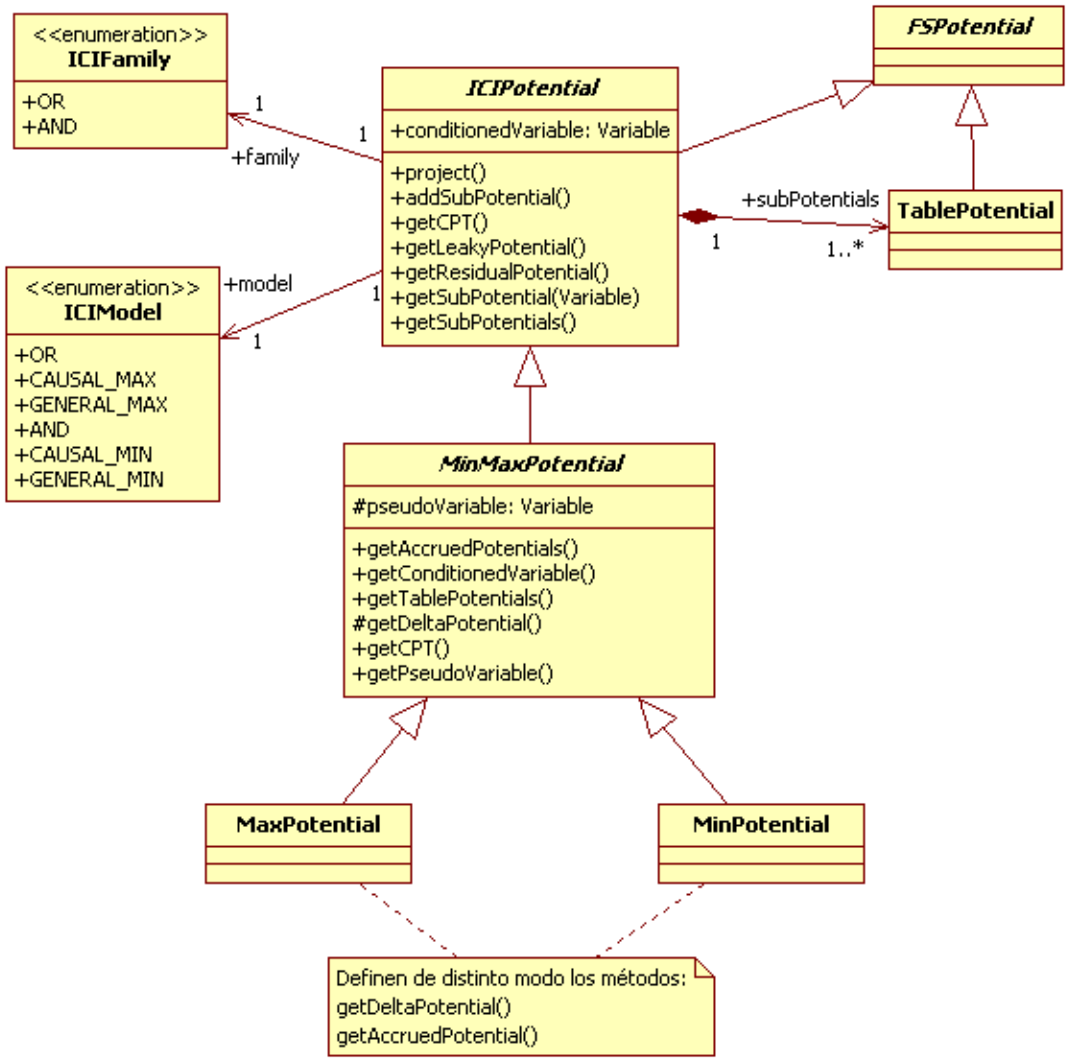


Figura 6.23: Nuevos tipos de potenciales usados en los modelos canónicos. Las clases de este diagrama están encapsuladas en el paquete *carmen.networks.canonical*.

de variables discretas, desciende de la clase *FSPotential (Finite States Potential)*; esta clase contiene los datos y métodos comunes a todos los modelos canónicos. Un *ICIPotential* está formado por varios potenciales de variables discretas ordinarios, del tipo *TablePotential*, uno por cada enlace. Tanto la familia a la que pertenece el potencial como el modelo que representa se indican con un enumerado, *ICIFamily* e *ICIModel* respectivamente; en el futuro se añadirán nuevas familias, como la *XOR*. La clase *MaxPotential* representa los modelos *OR* y varios tipos de *Max* y la clase *MinPotential* representa los modelos de los tipos *AND* y *MIN*; ambas definen de distinto modo los métodos *getDeltaPotential()* y *getAccruedPotential()*. Entre la clase *ICIPotential* y las clases *MaxPotential* y *MinPotential* hemos introducido la clase *MinMaxPotential*, que contiene el código que es común para ambas pero que no será común a otras clases que descenderán de *ICIPotential* en un futuro.

b) Computación de la probabilidad para modelos MAX/MIN

En el caso de los modelos canónicos de las familias MAX y MIN, los parámetros son las $c_y^{x_i}$ y c^L, dadas por las ecuaciones 1.16 y 1.17 (22).

Para computar la probabilidad de forma más eficiente (23), se definen unos nuevos parámetros,

$$C_y^{x_i} = \sum_{y'=0}^{y} c_y^{x_i} \tag{6.1}$$

$$C_y^{L} = \sum_{y'=0}^{y} c_y^{L} \tag{6.2}$$

. Se define también un potencial Δ_Y,

$$\Delta_Y(y,y') = \begin{cases} 1 & \text{si } y' = y \\ -1 & \text{si } y' = y-1 \\ 0 & \text{en otro caso} \end{cases} . \tag{6.3}$$

de modo que la tabla de probabilidad condicionada se puede calcular así:

$$P(y|\mathbf{x}) = \sum_{y'} \Delta_Y(y,y') \cdot \prod_i C_{y'}^{x_i} \tag{6.4}$$

Una factorización casi idéntica se puede aplicar a los modelos canónicos de la familia

MIN. Estas factorizaciones de $P(y|\mathbf{x})$ permiten, en muchos casos, una computación mucho más eficiente de la probabilidad de la red que contiene el nodo Y y sus padres: en los algoritmos de propagación de evidencia, en vez de incluir el potencial $P(y|\mathbf{x})$ explícitamente, hay que tomar n+2 potenciales (n de la forma $C_y^{x_i}$, uno por cada enlace; un potencial residual, C_y^L; y un potencial Δ_Y). A partir de ahí, las operaciones con modelos canónicos no difieren del resto de modelos probabilistas. Los algoritmos de inferencia implementados para redes bayesianas funcionan sin más que sustituir unos tipos de potenciales por otros.

6.3. Evaluación de diagramas de influencia

La evaluación de diagramas de influencia se realiza en Carmen mediante el método estándar de eliminación de variables, que calcula la política óptima para cada decisión y la máxima utilidad esperada. Hemos implementado sólo la variante con división de potenciales.

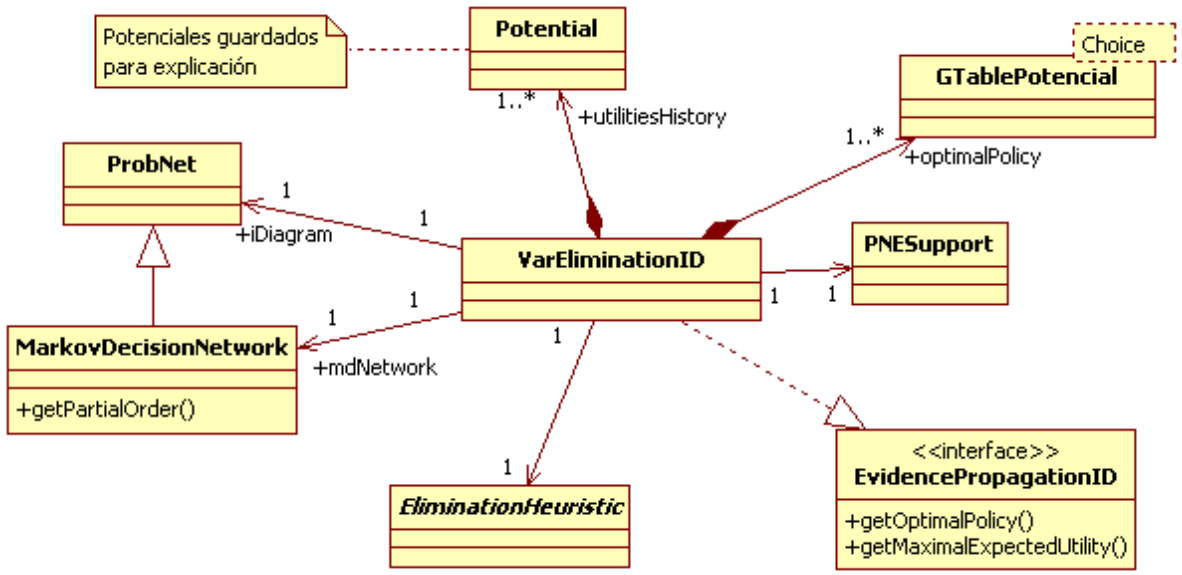


Figura 6.24: Vista estática: clases relativas a la eliminación de variables para diagramas de influencia.

La clase *MarkovDecisionNetwork* es la encargada de calcular el orden parcial del diagrama de influencia; un orden parcial representa una ordenación de variables de azar

y de decisión de modo tal que las variables de decisión siguen el camino dirigido que conecta todas las decisiones y cada una de ellas, en dicho orden parcial, está precedida por el conjunto de las variables de azar que son su padres; al final del orden parcial están todas las variables de azar que no son padres de ninguna decisión. Los conjuntos de variables de azar no tienen una ordenación definida dentro de dicho conjunto.

Posteriormente, se crea un objeto que implementa la interfaz definida por la clase abstracta *EliminationHeuristic*, y se utiliza dicha heurística para decidir el orden de eliminación de los nodos siguiendo el orden parcial. Las operaciones de eliminación de nodos se han codificado como ediciones compuestas que constan de varias ediciones más pequeñas. Existen dos ediciones distintas en función de qué tipo de nodo se esté borrando: azar (*CRemoveChanceNodeIDEdit*) o decisión (*CRemoveDecisionNodeIDEdit*); ambas del tipo *CompoundPNEdit*. La edición que elimina nodos de decisión se encarga de guardar un historial de las utilidad asociada a cada opción, lo que permitirá, en un futuro, implementar algoritmos de explicación similares a los del programa Elvira (41; 40). El objeto que se encarga de gestionar las ediciones es, como en los casos anteriores, del tipo *PNESupport*. La política óptima se guarda en una colección de objetos del tipo *GTablePotential*, que es un tipo de potencial similar a *TablePotential* pero que para cada posible configuración de las variables, en lugar de almacenar una número, almacena un objeto (en principio genérico, pero se puede parametrizar) del tipo *Choice*. Cada instancia de la clase *Choice* es una asignación de un valor, o más de uno en caso de empate, a una variable de decisión.

El algoritmo primero calcula el orden parcial del diagrama y luego, siguiendo ese orden, se eliminan las variables de azar y de decisión. En el caso de las variables de azar, cuando hay más de una en el conjunto que se está eliminando, se utiliza una heurística.

6.4. Aprendizaje de redes bayesianas

El componente de aprendizaje ha sido implementado por Jesús Oliva Gonzalo bajo nuestra supervisión. Dado que es bastante específico y relativamente independiente del resto, hemos decidido darle el mismo tratamiento que a otros componentes de mayor tamaño, describiendo sus objetivos, el modelo de análisis y el modelo de diseño, tanto arquitectónico como detallado.

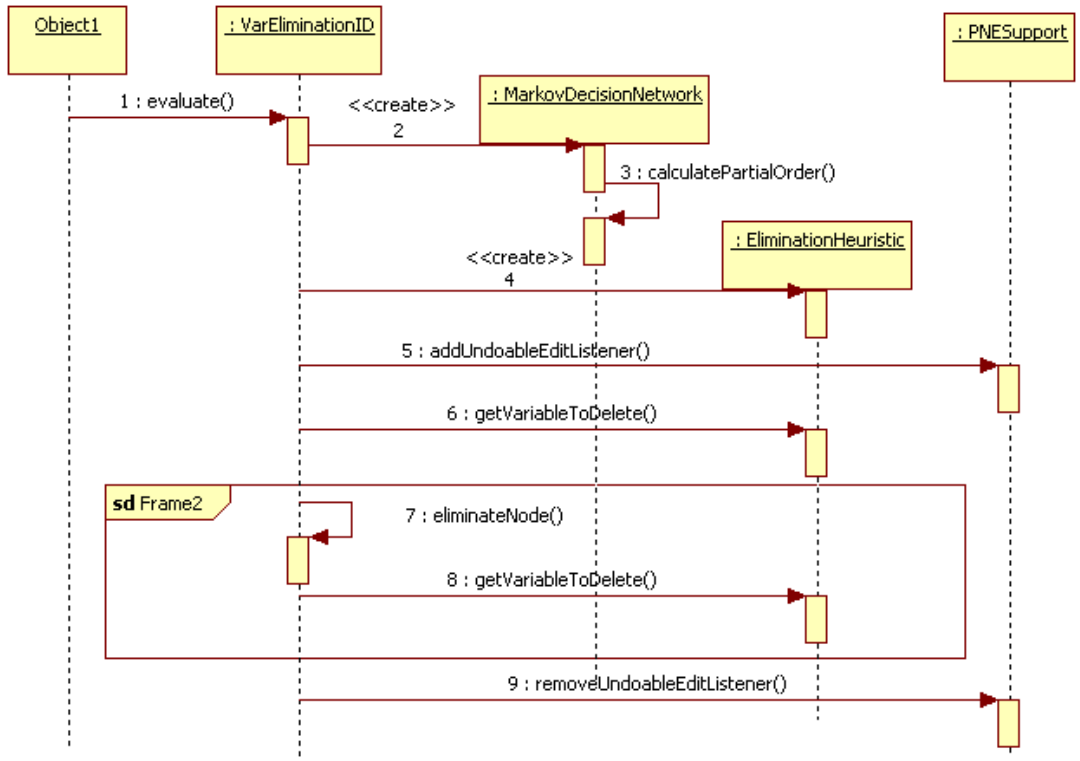


Figura 6.25: Vista dinámica: diagrama de secuencias del algoritmo de eliminación de variables para diagramas de influencia. Como en casos anteriores, se han omitido las llamadas relativas a la creación de ediciones para no complicar el diagrama.

6.4.1. Características del aprendizaje en Carmen

a) Algoritmos

En la versión actual de Carmen nos hemos centrado en los algoritmos de aprendizaje mediante búsqueda heurística (*search and score*), por ser los más utilizados en la actualidad.

Aunque sólo hemos implementado un algoritmo, el *algoritmo del gradiente*, la codificación de otros métodos es muy sencilla, debido a que el diseño de la herramienta Carmen en general y de este módulo de aprendizaje en particular se ha realizado teniendo entre sus objetivos prioritarios el de facilitar su ampliación.

b) Métricas

Las métricas más utilizadas pueden dividirse en tres grupos: métricas bayesianas, métricas de longitud mínima de descripción y medidas de información. Hemos hecho un diseño genérico y hemos implementado cuatro métricas: *Bayesian Dirichlet), K2, AIC* y *MDL.*

c) Red modelo

El aprendizaje en Carmen, además de tomar como entrada una base de datos, puede tomar una red modelo, que especifica algunas características de la red que se va a aprender:

- Nombres de variables. Esta opción permite tomar de la base de datos sólo aquellas variables incluidas en la red modelo[5]
- Las variables discretizadas. Como hemos dicho en la sección 5.3.2, en Carmen puede haber variables discretizadas, las cuales tienen asignado un intervalo para cada uno de sus estados. Cuando la base de datos tiene variables continuas, una forma de discretizarlas es mediante los intervalos de la variables del mismo nombre en la red modelo.
- Las posiciónes de los nodos en la pantalla. Esta opción permite ahorrar mucho tiempo cuando se hacen varias pruebas de aprendizaje para una misma base de

[5]Otra forma de seleccionar las *variables de interés*, es decir, el subconjunto de variables de la base de datos que se van a utilizar en el aprendizaje es declararlas explícitamente, por ejemplo, desde la interfaz gráfica.

datos: basta recolocar los nodos en patanlla una sola vez, y tomar esta red como modelo para la colocación de los nodos en las pruebas posteriores.

- Estructura de la red (enlaces entre variables). En este caso la estructura de la red aprendida es la misma que la de la red modelo, por lo que el aprendizaje es sólo paramétrico.

Una red modelo puede estar en formato Elvira (.elv) o en el formato propio de Carmen (.xml).

d) Base de datos

d.1) Formatos Los formatos de base de datos que puede leer Carmen son los siguientes:

- *Formato Excel (.xls)*: El formato de un fichero de casos en Excel aceptado por Carmen es el siguiente: La primera línea ha de contener los nombres de las variables implicadas, una en cada celda, sin que haya celdas en blanco entre ellas. En cada una de las líneas siguientes se ha de especificar un caso, incluyendo en cada celda el valor de una variable en el mismo orden en el que aparecen los nombres de las variables. Es decir, en cada columna ha de aparecer el nombre de la variable y todos los valores que toma dicha variable en los distintos casos registrados en el fichero. Cada uno de los valores de las variables puede especificarse como una cadena de caracteres o un entero. Para representar un valor ausente basta con dejar la celda correspondiente vacía.

- *Formato Weka*[6] *(.arff)*: En la implementación actual, las únicas restricciones que impone Carmen al formato de datos propio de Weka son las siguientes:

  - no se admiten atributos de tipo *numeric, string* ni *date*.
  - no se admiten ficheros de casos dispersos (*sparse ARFF files*).

- *Formato Elvira (.dbc)*: Es el formato de bases de datos utilizado por Elvira. Las propiedades generales son semejantes a las definidas para las redes y diagramas (formato .elv). Al comienzo del fichero se define el conjunto de nodos o variables de las que consta el problema, tipología, comentarios e información adicional que pueda resultar de interés. También debe incluirse el número de instancias o casos

[6] Weka es una herramienta para la construcción de diferentes clasificadores a partir de bases de datos (véase la sección 4.1.1).

que existen en el fichero, que tendrá que coincidir con el número de líneas que más tarde aparecen como información.

Una vez definidas todas las variables se incluyen los casos que conforman el cuerpo de conocimiento de la base de datos. Cada instancia se corresponderá con una línea, y, por cada línea, se incluirá el valor de esa instancia para cada una de las variables definidas en el preámbulo del fichero. Los valores que tome cada variable pueden ser separados bien por comas, o bien dejando un espacio en blanco.

La única restricción que impone Carmen es que no se pueden leer variables continuas. Para introducir una variable continua se ha de definir como variable discreta y especificar uno a uno todos sus posibles estados.

d.2) Tratamiento de las variables numéricas Carmen ofrece la posibilidad de tratar las variables de las siguientes formas:

- No discretizar: cuando un número representa un código, —por ejemplo, {0: ingresado, 1: ambulatorio}— no tiene sentido discretizar la variable por intervalos. En este caso, el usuario puede indicar que cada valor numérico debe tratarse como un estado, no como un número.

- Discretizar: hay tres posibilidades para la discretización de una variable numérica:

  a) Según la correspondiente variable discretizada de la red modelo, como hemos explicado en la sección 6.4.1.c.

  b) Partición en intervalos de igual frecuencia. Un caso particular es la partición de dos intervalos divididos por la mediana de las frecuencia en la base de datos. Esto se generaliza a $k+1$ intervalos mediante las k-medianas (*k-means*).

  c) Partición en intervalos de igual anchura: el intervalo que se particiona, que está delimitado por los elementos de valor mínimo y máximo, se divide en k intervalos de la misma anchura.

d.3) Tratamiento de valores ausentes Carmen ofrece dos posibilidades para el tratamiento de los valores ausentes:

1. Asignar un estado llamado “ausente” y tratarlo como si fuera un estado más.

2. Eliminar los registros en que alguna de las variables de interés tiene un valor ausente.

En el futuro implementaremos métodos específicos de imputación de valores ausentes.

e) Parámetro α

Para evitar problemas de sobreajuste o de configuraciones inexistentes, es posible especificar un parámetro $\alpha \geq 0$, de modo que si $\alpha = 1$ equivale a la corrección de Laplace, y si $\alpha = 0$ equivale a calcular el estimador de máxima verosimilitud para cada configuración[7].

6.4.2. Modelo de análisis

La representación de los algoritmos y de las redes probabilistas la vamos a realizar del mismo modo que en la sección 6.1.2. En la figura 6.26, la clase *LearningMain* coordina todo el proceso, recogiendo las opciones seleccionadas por el usuario y ejecutando con los parámetros correctos la clase *LearningAlgorithm*, que contiene el algoritmo de aprendizaje.

6.4.3. Diseño arquitectónico

El componente se descompone como se indica en la figura 6.27.

El paquete *algorithms* contiene el algoritmo que se ha implementado. Interactúa con *metrics*, que implementa varias métricas con una interfaz común, y *cache*, que guarda estructuras de datos que aceleran el algoritmo.

El paquete *preprocess* se encarga del preprocesamiento de los datos contenidos en la base de datos. Permite dos tipos de preprocesamiento: el tratamiento de valores ausentes y la discretización de variables numéricas.

El paquete *gui* es el encargado de mostrar la interfaz gráfica que se le presenta al usuario para que elija las distintas opciones parametrizables e inicie el proceso de aprendizaje.

El paquete *io* se encarga de la entrada / salida (lee y escribe archivos de modelos gráficos probabilistas). Se compone de otros dos paquetes: *configuration* que sirve para

[7] Si la variable X puede tomar m valores, $[x_1,\ldots,x_m]$ y para cierta configuración las frecuencias son $f_1,\ldots,f_m$, las correspondientes probabilidades se estiman como $\widehat{p}_i = \frac{f_i+\alpha}{m\cdot\alpha+\sum_{i'=1}^{m} f_{i'}}$. Si $\alpha = 0$, tenemos la estimación de máxima verosimilitud; si $\alpha = 1$, tenemos la corrección de Laplace; véase (57)

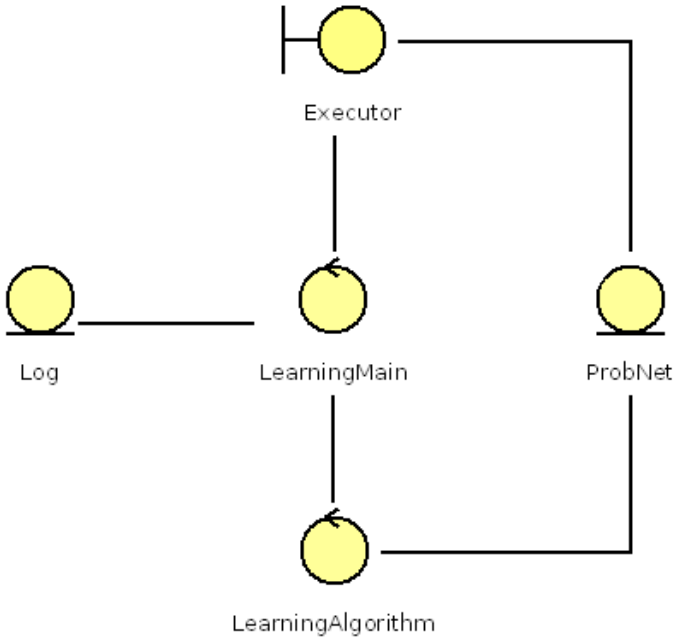


Figura 6.26: Modelo de análisis de un algoritmo de aprendizaje.

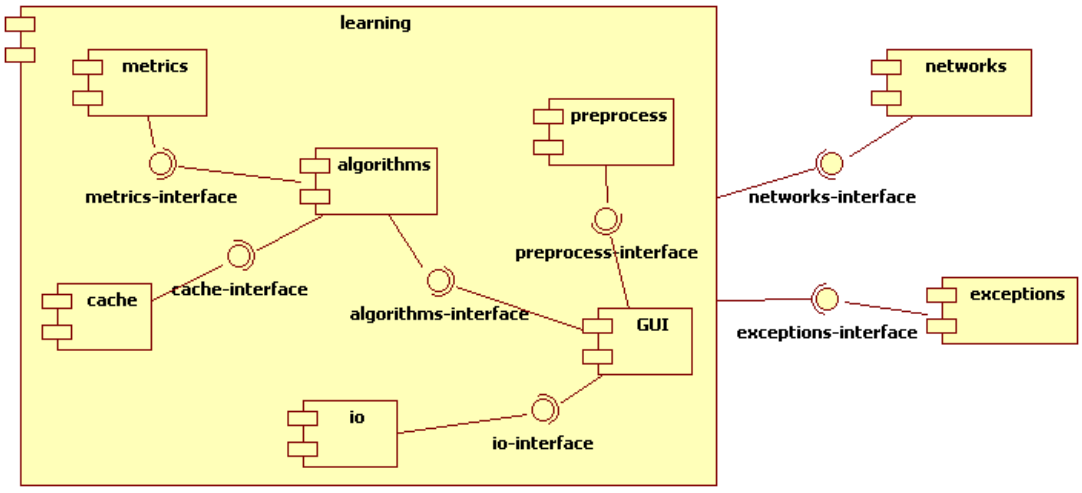


Figura 6.27: Componentes del aprendizaje.

leer de disco un conjunto de variables del sistema y *localize*, que sirve para mostrar las cadenas de caracteres del programa en varios idiomas.

6.4.4. Diseño detallado

Al igual que en otros componentes, este diseño lo vamos a expresar desde dos puntos de vista: *estático*, con diagramas de clases, y *dinámico*, con diagramas de secuencias.

a) Punto de vista estático

Para englobar todos los algoritmos de aprendizaje se ha creado una interfaz *LearningAlgorithm*, común a todos ellos, con un método *run()*, que va a devolver la red bayesiana aprendida. Se utilizará una métrica, representada por un objeto del tipo *Metric*, y una memoria caché representada por un objeto del tipo *Cache*.

En el diagrama de clases de la figura 6.28 es la clase *LearningAlgoritm* la que va a actuar como objeto observado y las clases *Métrica* y *Cache* como observadores (*listeners*). Por ese motivo la clase *LearningAlgorithm* es gestionada por la clase *PNESupport* y las clases *Métrica* y *Cache* implementan la interfaz *PNUndoableEditListener* para escuchar los eventos de cambio y actualizar sus copias locales.

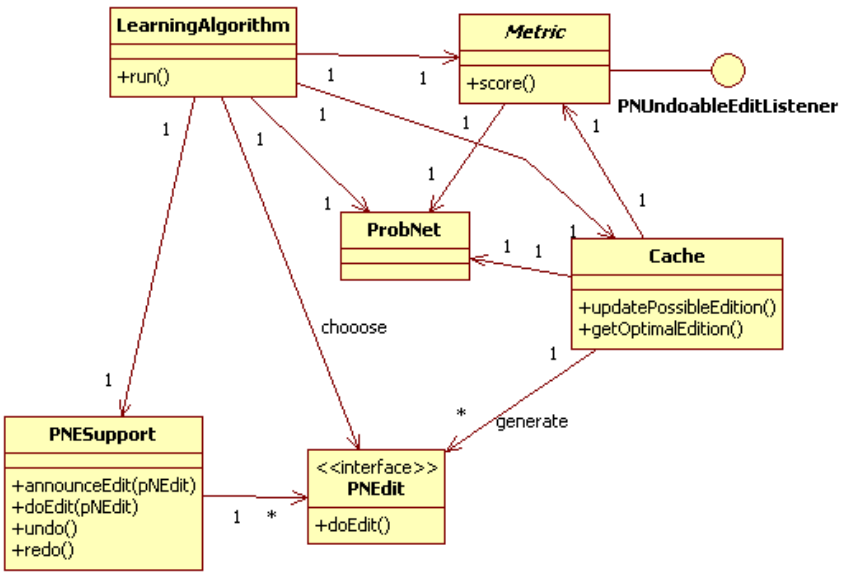


Figura 6.28: Vista estática del modelo de diseño. Clases más relevantes.

Las métricas implementadas hasta el momento, según la clasificación que hemos indicado en los objetivos (véase la figura 6.29) están localizadas en el paquete *metrics*.

Para englobar todas las métricas de calidad se ha creado una clase abstracta *Metric*, con un método llamado *score()*, que ha de ser implementado por todas sus subclases y que debe devolver la puntuación global de la red. Para cumplir con el requisito de eficiencia, se ha de evitar realizar todos los cálculos cada vez que se llama a este método. Las métricas que se emplean habitualemnte en el aprendizaje de redes bayesianas, entre las cuales están las que hemos implementado, cumplen la propiedad de *localidad*, es decir, que la puntación global se calcula mediante una combinación sencilla (por ejemplo, la suma) de las puntuaciones de cada una de las familias de la red. Como las sucesivas redes con las que se va a trabajar se generan añadiendo, eliminando o invirtiendo un único enlace, basta con realizar los cálculos a los que afecta ese nuevo enlace y obtener la calidad de la nueva red a partir del valor de calidad de la anterior.

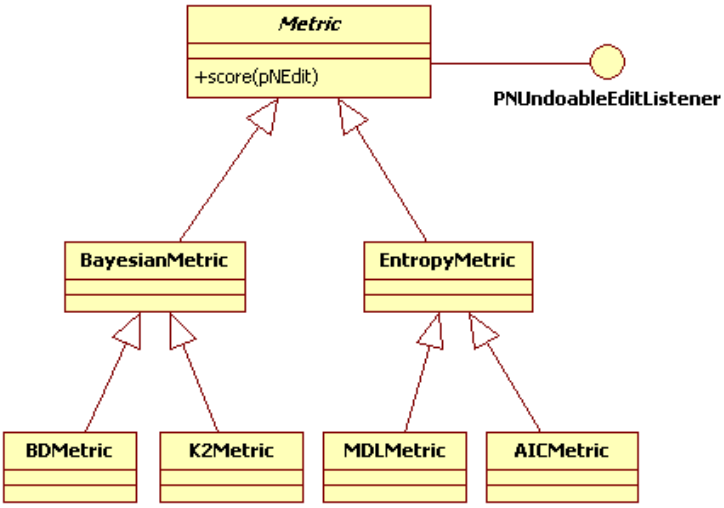


Figura 6.29: Diagrama de clases de las métricas implementadas en Carmen.

De la clase *Metric* heredan directamente dos clases principales: *BayesianMetric* y *EntropyMetric*, que se corresponden con las métricas de tipo bayesiano y las métricas de entropía. El hecho de que no se incluya directamente una subclase que agrupe a las métricas de longitud mínima de descripción se debe a que éstas están estrechamente relacionadas con las medidas de información (o de entropía) de modo que todas ellas pueden ser englobadas por la clase *EntropyMetric*. Cada uno de estos dos subtipos tiene

características en común que hacen aconsejable esta estructura de clases. Las métricas bayesianas comparten las funciones de cálculo global de la métrica, variando tan solo los cálculos de la puntuación de cada uno de sus nodos al añadirse o eliminarse un enlace. Por su parte, las métricas de longitud mínima de descripción y las de entropía tienen en común los cálculos de la entropía y la dimensión de un nodo (aunque este último no es utilizado por todas las métricas) cuando se le añade o elimina un enlace.

b) Punto de vista dinámico

Describimos ahora cómo funciona el esquema anterior durante la ejecución del algoritmo; véase el diagrama de secuencias de la figura 6.30.

Un algoritmo es iniciado por algún objeto, por ejemplo la interfaz gráfica. La ejecución de un algoritmo tiene lugar en tres fases: lo primero que hace un algoritmo es crear los objetos necesarios para su funcionamiento: entre ellos, la métrica, la caché y un objeto del tipo *PNESupport*. Aquellos objetos interesados en las operaciones del algoritmo se registran en *PNESupport*. Para terminar esta primera etapa, la caché es inicializada pidiendo a la métrica que puntúe cada una de las posibles ediciones que se pueden realizar sobre la red inicial.

La segunda fase es la de operación: en general, en cada iteración del bucle principal, el algoritmo de aprendizaje pide a la caché la mejor edición (en caso del algoritmo del gradiente, la "mejor edición" es la que tiene una mayor puntuación asociada) la realiza comunicando al objeto *PNESupport* la operación *PNEdit* realizada; éste se encarga de comprobar si hay algún veto. Si no lo hay, se envía un mensaje a *PNESupport* para que realice la acción. Cada uno de los objetos oyentes ha de realizar las operaciones pertinentes al recibir la notificación de que se ha realizado una edición. La caché ha de actualizar las puntuaciones asociadas a las ediciones afectadas por el cambio en la red.

Una vez terminado el aprendizaje estructural, ha de realizarse el aprendizaje paramétrico, que consiste en construir la table de probabilidad condicionada para cada uno de los nodos de la red. Por último, en la fase de finalización se devuelve la red obtenida a la interfaz gráfica para ser representada en pantalla.

c) Caché de posibles ediciones

En nuestra implementación actual, la caché está formada básicamente por dos matrices de dimensiones $n \times n$, donde n es el número de nodos de la red que se quiere aprender. Una de las matrices está destinada a almacenar las puntuaciones asociadas a

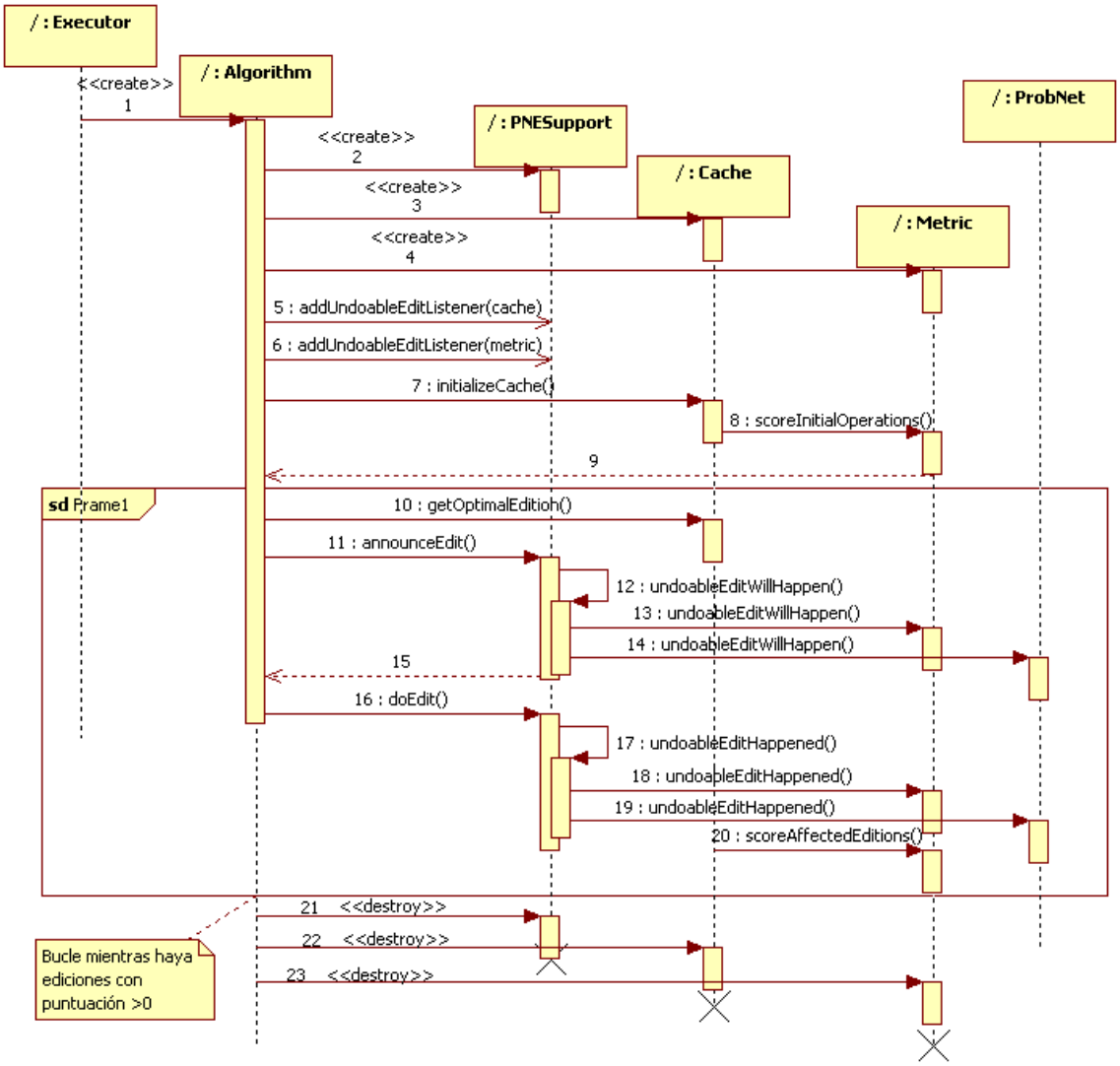


Figura 6.30: Diagrama de secuencias correspondiente a la ejecución de un algoritmo de aprendizaje.

la *adición* de un enlace, mientras que la segunda almacena las puntuaciones asociadas a la *eliminación* de un enlace. En cada una de las matrices, la casilla $[i, j]$ contiene la puntuación asociada a la inserción o elimiandción, respectivamente, del enlace que va del nodo *i*-ésimo al nodo *j*-ésimo. En cuanto a la inversión de enlaces, no es necesario guardar sus puntuaciones, puesto que una inversión del enlace $A \rightarrow B$ es similar a la eliminación del enlace $A \rightarrow B$ y la inserción del enlace $B \rightarrow A$.

El funcionamiento de la caché es muy simple. La inicialización consiste en puntuar todos los posibles enlaces bien de la red tal que no es necesariamente la red vacía, sino que puede ser otr que el usuario decida. Durante la ejecución del algoritmo, cada vez que se realiza una operación, los valores de la caché han de actualizarse. La inserción o elminación de un enlace $A \rightarrow B$ sólo afecta a las puntuaciones de los enlaces que llegan al nodo B. De este modo, basta con actualizar los valores de las columnas asociadas al nodo B en cada una de las dos matrices.

Naturalmente, en caso de que la edición consista en la inversión, es necesario actualizar las columnas asociadas tanto al nodo origen como al nodo destino.

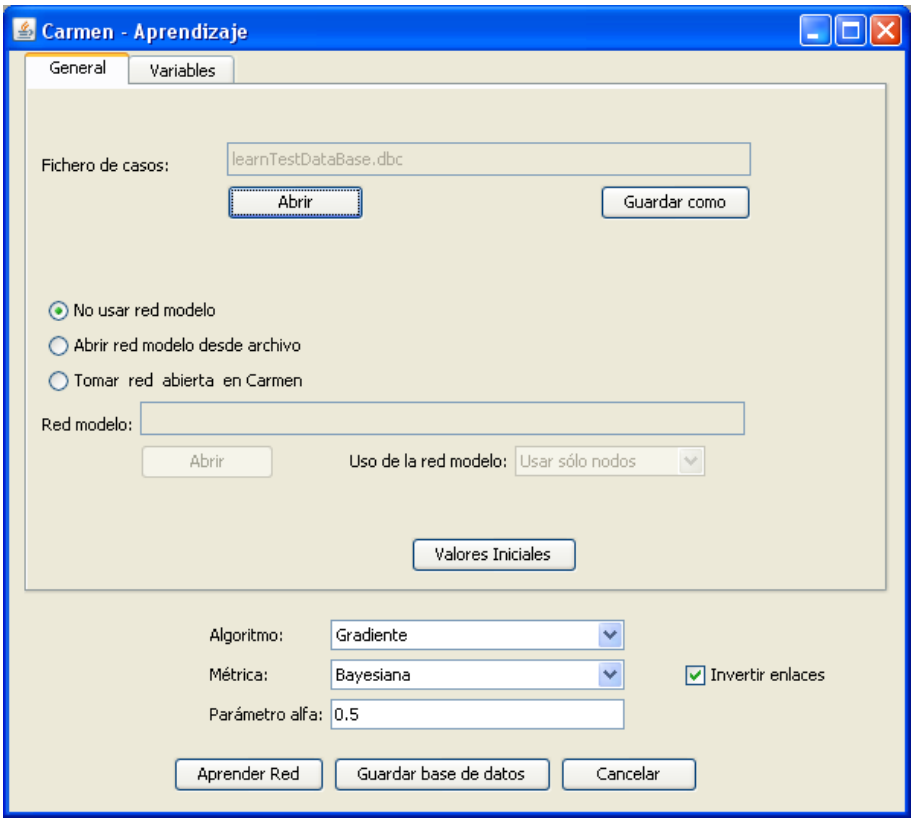
Resumen: Implementación de un algoritmo de aprendizaje

Si un usuario (programador) quisiera implementar un nuevo método de aprendizaje en Carmen, tendría que crear sustitutos para uno o varios de los componentes actuales. Los pasos que hay que seguir para implementar un nuevo algoritmo de aprendizaje en Carmen son los siguientes:

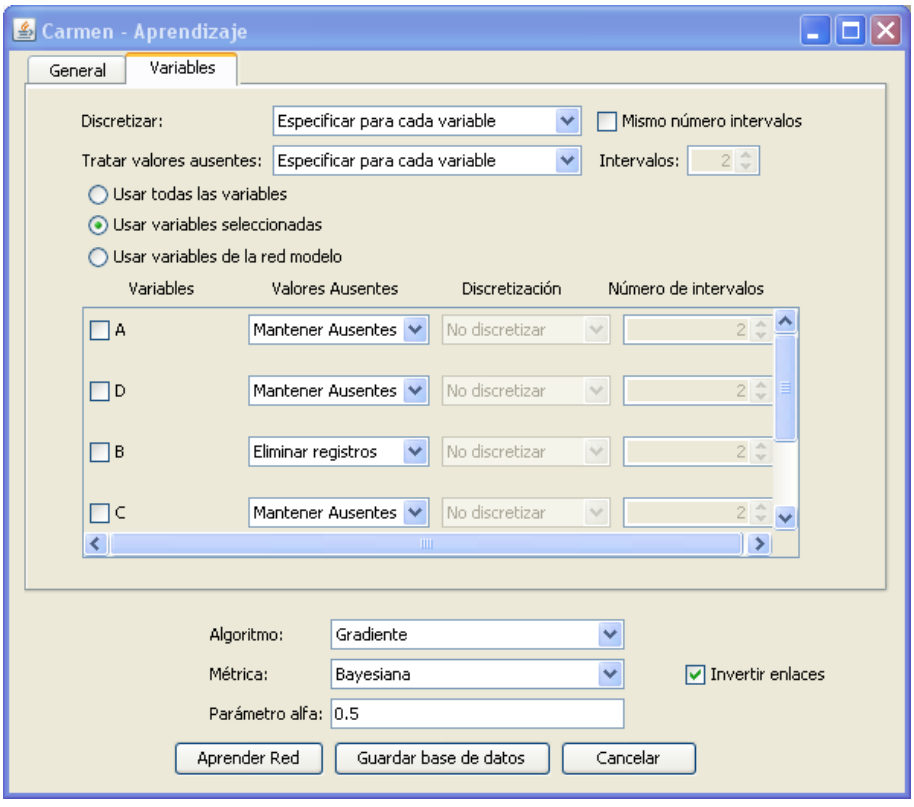
1. Crear una clase dentro del paquete *carmen.learning.algorithms* que herede de la clase abstracta *LearningAlgorithm*.

2. Implementar el método *run()* y el método *parametricLearning()*.

3. Modificar la clase *Cache* en función de las características del nuevo algoritmo. Por ejemplo, en vez de seleccionar lamejor edición de forma miope, en función de la puntuación inmediata (*one-step look-ahead*), se podrían utilizar métodos más sofisticados.



(a) Ventana general.



(b) Tratamiento de variables.

Figura 6.31: Captura de pantalla de las ventanas del módulo de aprendizaje.

6.5. Interfaz gráfica de usuario

La interfaz gráfica de usuario en Carmen ha sido el resultado de dos proyectos fin de carrera y de una colaboración. El primer PFC, realizado por José Enrique Mendoza, fue un prototipo preliminar que permitía leer redes, representarlas y modificar la estructura gráfica, aunque no podía editar las tablas de los nodos. Posteriormente, Alberto Manuel Ruiz implementó la edición de las características de los nodos (tipo de nodo, estados, etc) pero no la edición de las tablas de probabilidades. Por último, el proyecto de Juan Luis Gozalo, actualmente en fase de desarrollo, consiste en una tarea de reingeniería sobre el primer prototipo de la interfaz e integrar la parte de edición de nodos que se tiene implementada por el momento.

Como este trabajo no está todavía terminado, no consideramos oportuno exponer un diseño que seguramente tendrá cambios en su versión final.

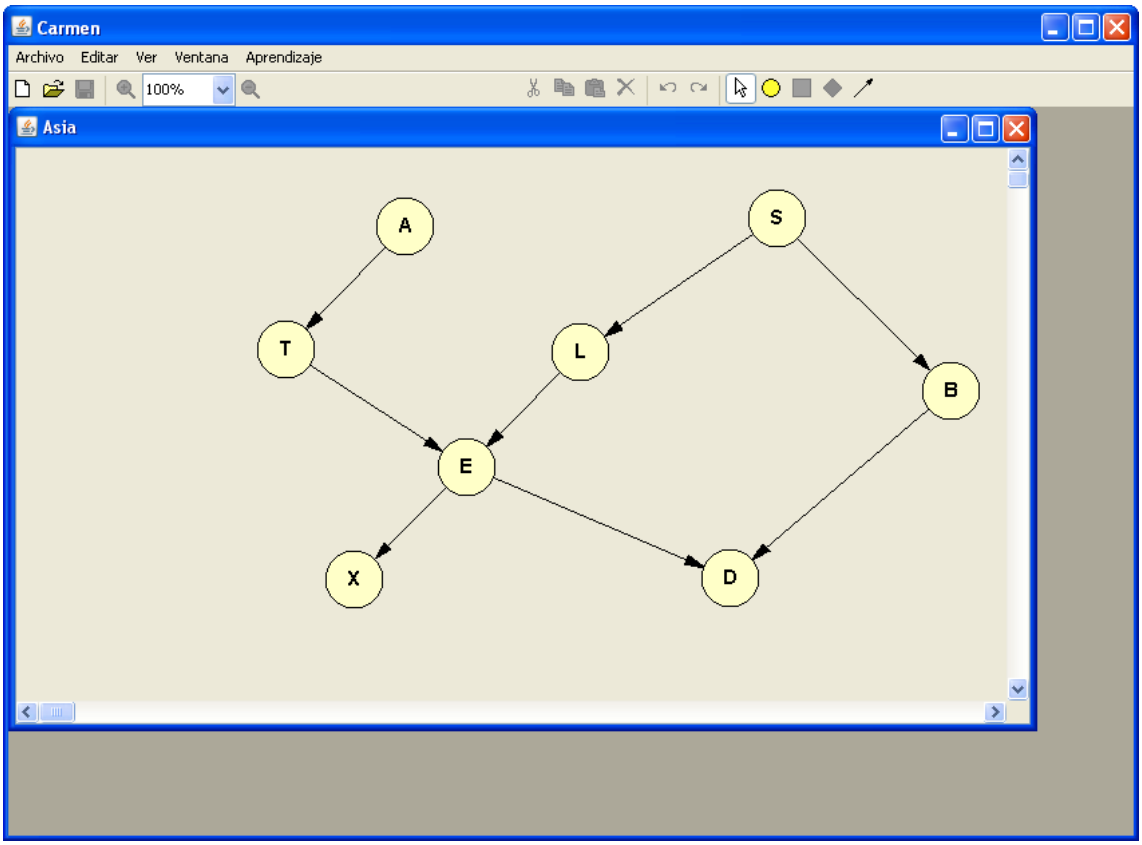


Figura 6.32: Aspecto de la interfaz gráfica de Carmen en el momento actual.

En términos generales, la interfaz gráfica de Carmen pretende tener las mismas funciones que las de Elvira. El diseño sin embargo, se ha realizado desde cero.

Parte IV

Conclusiones

Capítulo 7

Conclusiones

Finalizamos esta memoria presentado las principales aportaciones del trabajo realizado, en la sección 7.1, y exponiendo las posibles líneas de trabajo que quedan abiertas el futuro, en la sección 7.2.

7.1. Principales aportaciones

Tal como se indicó en la introducción, los objetivos iniciales eran dos: desarrollar una herramienta de software libre para editar y evaluar sobre modelos gráficos probabilistas (MGPs) y crear un algoritmo para el análisis de coste-efectividad que permita considerar varias variables de decisión. La herramienta CARMEN ha supuesto otras aportaciones secundarias que también vamos a comentar.

7.1.1. Aportaciones independientes de la herramienta CARMEN

Detallamos en primer lugar las aportaciones que, aunque se han utilizado en la construcción de CARMEN, son independientes de ella y, por tanto, se podrían utilizar en otras herramientas.

a) Algoritmos para operaciones básicas con potenciales

Hemos diseñado un método que mejora la eficiencia de las operaciones sobre potenciales con variables discretas. Como hemos dicho en la sección 2.6, es incorrecto analizar la complejidad computacional de los algoritmos de inferencia midiendo sólo el número de operaciones elementales, como la suma o la multiplicación de los valores de

los potenciales, porque también hay que tener en cuenta el tiempo que se tarda en acceder a dichos valores. El método que se ha desarrollado mejora sustancialmente la velocidad de las operaciones básicas de marginalización, maximización, suma, multiplicación y proyección. La eficiencia de estas operaciones se ha cuidado bastante porque este código es el que más tiempo se ejecuta con los algoritmos de inferencia.

b) Análisis de coste-efectividad

Hemos desarrollado un método para operaciones de coste-efectividad sobre árboles de decisión, cuyas ventajas principal es que permite incluir varias variables de decisión y de azar en el modelo mientras que los métodos anteriores sólo permitían una variable de decisión, que tenía que estar en la raíz del árbol (*TreeAge*, el programa comercial más utilizado en la actualidad, permite incluir varias decisiones en el árbol, pero de resultados incorrectos, como hemos mostrado con un ejemplo en la sección 3.2.2) y, presenta los resultados como un conjunto de intervalos del parámetro λ.

Nuestro método ha sido implementado en dos algoritmos, uno de ellos para árboles de decisión y otro para diagramas de influencia. Esperamos que este método sea utilizado ampliamente especialmente en medicina, donde hay un gran interés por este tipo de estudios.

c) Nuevo patrón de diseño

Hemos desarrollado un patrón de diseño, denominado *Permiso-Ejecución*, que permite editar objetos que están sujetos a restricciones. Las ventajas de este patrón son que permite que un objeto pueda tener asociadas un conjunto de restricciones que definen las operaciones que se pueda hacer sobre él, y que estas restricciones se pueden añadir o quitar de un modo dinámico. Este patrón es una aportación que puede ser utilizada en otros tipos de problemas distintos de los MGPs.

7.1.2. Herramienta de software libre para desarrollo de MGP

a) CARMEN como herramienta de software libre

La construcciòn de la herramienta CARMEN ha seguido las fases clásicas de desarrollo en ingeniería del software: especificación, análisis, diseño, codificación y pruebas. Las caracteríticas más importantes de Carmen desde el punto de vista de la ingeniería del software son:

1. **Robustez**: la reducida densidad de errores se ha conseguido mediante la creación de un conjunto sistemático de pruebas con la herramienta *jUnit*. Estas pruebas se incluyen junto con el código fuente en el repositorio de CARMEN.

2. **Eficiencia**, se ha conseguido optimizando todo lo posible las estructuras de datos y los algoritmos, en particular el funcionamiento de la librería de operaciones básicas sobre potenciales, que es la parte del código que consume la mayor parte del tiempo en la evaluación de los MGPs (capítulo 2), como ya hemos comentado.

   Como hemos visto en la sección 4.1, la propagación de evidencia en Carmen es muy superior a la de Elvira y comparable a la de las herramientas comerciales más avanzadas de la actualiadad: aunque CARMEN tarda más en hacer la inferencia cuando la red se compila independientemente de la evidencia, es muy rápida en la compilación de la red y esto permite realizar una propagación específica para la evidencia disponible, lo cual supone en muchos casos un importante ahorro de tiempo.

3. **Mantenibilidad**, como consecuencia de haber sido diseñada desde un principio pensando en la extensión de sus componentes y con una arquitectura que permite acomodar fácilmente los que se desarrollen en el futuro.

4. La mantenibilidad se ve favorecida por una extensa **documentación**, tanto interna como externa del proyecto que actualmente consta de:

   - Un conjunto de páginas HTML generadas automáticamente por la herramienta Javadoc[1] a partir de los comentarios incluidos en la documentación del código.
   - El artículo (3), presentado en el *European Workshop on Probabilistic Graphical Models (PGM-08)*.
   - Los capítulos 4 a 6 de esta memoria; en ellos se incluyen varios tipos de diagramas UML que muestran gráficamente la estructura y la dinámica de los componentes de CARMEN.

5. También contribuye a la mantenibilidad la **normativa de codificación**, la cual ha servido para dar un estilo homogéneo que facilite el mantenimiento del código. Esta normativa está basada en la publicada por la empresa Sun Microsystems, creadora

[1] Es una herramienta para documentar el código desarrollada por Sun, véase http://java.sun.com/j2se/javadoc.

del lenguaje de programación Java (véase java.sun.com/docs/codeconv/CodeConventions.pdf), con modificaciones menores.

Hemos puesto en marcha un repositorio con el código fuente del proyecto, gestionado por el servidor de versiones *Subversion*.

Licencia de CARMEN CARMEN se distribuye con licencia LGPL (véase http://www.gnu.org/copyleft/lesser.html), lo cual permite que cualquier persona, grupo de investigación o empresa puede utilizar libremente el software y modificarlo como estime oportuno. También permite que se desarrollen componentes (*plugins*) de pago.

b) Componentes de CARMEN

1. Librería para grafos Hemos desarrollado una librería genérica sobre grafos, que es independiente del resto de la aplicación. Esta librería es un refinamiento del modelo de listas de adyacencia, la diferencia principal es que separa los diferentes tipos de enlaces en listas distintas, lo que permite en general una eficiencia mayor, fundamentalmente cuando estos grafos representan MGPs.

2. Librería para operaciones básicas con potenciales de variables discretas Hemos hecho un análisis de las operaciones más comunes con potenciales de variables discretas y hemos implementado una librería que utiliza el método expuesto en el capítulo 2.

3. Librería para creación y edición de MGPs Hemos desarrollado una librería para MGPs donde se propone un nuevo tipo de diseño que desacopla en varios tipos de objetos las diferentes características que describen un MGP: grafo, restricciones (que son las que describen el tipo de modelo) y potenciales. Este diseño es generalizable y permitirá en un futuro añadir variables continuas, modelos temporales, etc. En general, el mantenimiento es sencillo porque las posibles modificaciones que se hagan en un futuro estarán localizadas en clases concretas. En el momento actual, nuestra librería permite editar y evaluar redes bayesianas y diagramas de influencia.

4. Algoritmos de inferencia En primer lugar hemos conceptualizado los algoritmos de modo que operen teniendo en cuenta las restricciones que pueden tener asociados los diferentes tipos de modelos y hemos creado un nuevo patrón para la manipulación de estructuras de datos de este tipo. En la actualidad, están implementados en Carmen

para redes bayesianas los algoritmos de eliminación de variables, propagación perezosa y el método de Hugin. Para diagramas de influencia está implementado el método de eliminación de variables.

5. Algoritmos de aprendizaje Jesús Oliva, que trabajó bajo la supervisión cercana del director de esta tesis y del doctorando, ha implementado un algoritmo de aprendizaje de redes bayesianas a partir de bases de datos, del tipo búsqueda-y-puntuación (en inglés, *search and score*), que utiliza el método del gradiente, con cinco métricas diferentes. A través de la interfaz gráfica se pueden seleccionar varias opciones para el preprocesado de datos y la métrica que se va a utilizar.

5. Interfaz gráfica Ha sido el resultado de dos proyectos fin de carrera. En su aspecto externo está inspirado en la interfaz de Elvira, pero el diseño ha sido completamente nuevo.

c) Utilización de CARMEN

Hay varios grupos que están utilizando CARMEN para sus tareas de investigación o para la implantación de sistemas expertos basados en MGPs.

Así, en nuestro grupo hemos desarrollado un sistema de ayuda a la decisión para cirugía de cataratas, por encargo de la Agencia Laín Entralgo, que ya está funcionando en modo de pruebas en el Hospital de Fuenlabrada (Madrid), y más tarde se implantará en varios de los grandes hospitales de la Comunidad de Madrid. Aunque la red bayesiana CATARNET se ha construido con Elvira, porque la interfaz gráfica de CARMEN aún no estaba completa, el sistema de ayuda a la decisión utiliza las librerías de inferencia de CARMEN porque son más eficientes que las de Elvira: la evaluación de CATARNET con Elvira puede tardar casi un minuto, dependiendo de la evidencia introducida —lo cual es inaceptable para un médico que está pasando consulta— mientras que con CARMEN nunca tarda más de unas décimas de segundo.

Análogamente, en 2010 la Empresa Pública de Emergencias Sanitarias de Andalucía va a utilizar CARMEN para la implantación de un sistema para ayudar a los operarios que atienden las llamadas telefónicas a tomar la mejor decisión en cada caso. Este sistema se basará en un conjunto de redes bayesianas, desarrolladas por D. Juan Gayubo dentro de una tesis doctoral de la que es co-director el Prof. Javier Díez.

Manuel Luque, profesor del Dpto. de Inteligencia Artificial de la UNED, se ha servido de CARMEN para realizar el análisis de sensibilidad de MEDIASTINET, un diagrama de influencia para el diagnóstico de cáncer de pulmón. Aunque el Prof. Luque ha utilizado Elvira para la mayor parte de su investigación, el análisis de sensibilidad con esa herramienta requería muchas horas, en unos casos, y en otros no podía realizar la computación por falta de memoria. En cambio, CARMEN es capaz de realizar todo el análisis de sensibilidad de MEDIASTINET (46, sec.~analisis-de-sensibilidad-de-Mediastinet) en unos 20 minutos.

Jesús Oliva, alumno del alumno del *Máster en Inteligencia Artificial Avanzada* de la UNED, ha utilizado los algoritmos de aprendizaje —que él mismo ha implementado en CARMEN— para la construcción de redes bayesianas a partir de tres bases de datos:

- pacientes de cáncer de pulmón, facilitada por el Dr. Agustín Gómez de la Cámara, del Hospital 12 de Octubre (Getafe, Madrid),

- pacientes de UCI, aportada por el Dr. José Javier Trujillano, Hospital Universitario Arnau de Vilanova (Lérida), y

- registro de hipotecas impagadas, que pusieron a nuestra disposición D. Antonio Ríos y D. Manuel Padial, del Área de Control Integral del Riesgo de CajaMadrid.

Carmen va a ser utilizada dentro del proyecto *Modelos gráficos probabilistas dinámicos y sus aplicaciones* para la construcción de modelos probabilistas relacionales y de modelos de decisión de Markov parcialmente observables (POMDPs). Se trata de un proyecto internacional, cofinanciado por el 7^{o} Programa Marco de la Unión Europea y el Consejo Nacional de Ciencia y Tecnología (CONACYT) mexicano, en el cual participan empresas y universidades de México, Holanda, Reino Unido, Francia y España. En la reunión de inicio del proyecto, celebrada en Puebla (México) en septiembre de 2009 se decidió utilizar esta herramientapor ser software libre, por su eficiencia y porque los autores de Carmen estamos involucrados en dicho proyecto.

7.2. Trabajo futuro

Exponemos algunas de las líneas de trabajo que quedan abiertas para el futuro.

7.2.1. Extensión de la herramienta CARMEN

a) Aspectos generales

Las mejoras que se pueden realizar sobre CARMEN en el futuro son varias:

- Escribir un *manual del programador*, en inglés, que incluya y amplíe los capítulos 4, 5 y 6 de esta memoria, así como la normativa de programación.
- Poner el código fuente en un repositorio público, como Sourceforge (http://sourceforge.net) o JavaSource (http://java-source.net).
- Extender y documentar el formato CarmenXML.
- Mejorar la página web de CARMEN (www.cisiad.uned.es/carmen).

b) Aspectos específicos

Los algoritmos que vamos a implementar en un futuro son:

- Otros métodos exactos para diagramas de influencia. Como hemos dicho en la introducción, actualmente hay una alumna que está implementado el método de Luque y Diez (47) para diagramas de influencia con nodos super-valor. Más adelante implementaremos otros como la propagación perezosa (en la actualidad está implementado sólo para redes bayesianas) y la inversión de arcos (59; 65).
- Algoritmos aproximados (simulación estocástica) para redes bayesianas y diagramas de influencia, especialmente, los de muestreo por importancia (56), que al parecer, son los que mejores resultados están dando en la actualidad.
- Otros tipos de MGPs, como los procesos de decisión de Markov, los LIMDS (42), los DLIMIDS (76) y redes de análisis de decisiones (48).
- Facilidades para análisis de sensibilidad. En primer lugar, es necesario poder asignar intervalos y distribuciones de probabilidad a los parámetros de los modelos, y en segundo lugar, integrar en la interfaz los métodos de análisis de sensibilidad implementados en CARMEN por Manuel Luque en la sección 3.3 de (46).
- Algoritmos de aprendizaje basados en la detección de independencias condicionales, como los que se pueden ver en (57).

- Mejoras en el preprocesado de datos, como la imputación de valores ausentes.
- Mejoras en la interfaz, que aun no está concluida: opciones de explicación del modelo y del razonamiento, semejante a los de Elvira (40).

7.2.2. Líneas de investigación abiertas

- En primer lugar, tenemos previsto extender nuestro método de coste-efectividad a otros tipos de modelos, como las redes de análisis de decisiones (48) y los procesos de decisión de Markov (62; 5).
- Una vez implementados en CARMEN los difererentes modelos temporales, sería interesante compararlos entre sí, —en general los DLIMIDS (76) frente a los POMPDs (5)— en cuanto a la eficiencia computacional y en cuanto a la calidad de los resultados, es decir, el valor esperado que cada uno de estos métodos puede lograr para un problema concreto.
- Los algoritmos y operaciones básicas que operan sobre MGPs tienen potencial para ser implementados en paralelo (50; 63). Se ha realizado una implementación paralela de las operaciones básicas, aunque no hemos encontrado diferencias significativas de rendimiento entre la versión secuencial y la paralela. Pensamos que para aprovechar las ventajas del paralelismo es necesario tener en cuenta las características de la arquitectura hardware (procesador, memoria cache, buses, etc.) y, en nuestro caso, el funcionamiento de la máquina virtual de Java.

Apéndices

Apéndice A

UML

El UML es la "lingua franca" de la ingeniería del software. Este apéndice sólo tiene lo mínimo necesario para entender todos los diagramas que hay en la tesis. La organización es como sigue: la sección A.1 describe brevemente los elementos del lenguaje y los tipos de diagramas; la sección A.2 describe en detalle los tipos de diagramas que se han utilizado en la tesis.

A.1. Introducción

El UML (Unified Modeling Language) es el lenguaje gráfico estándar para visualizar, especificar, construir y documentar sistemas software. El lenguaje prescribe un conjunto de notaciones y diagramas estándar para modelar sistemas orientados a objetos y describe la semántica esencial de estos diagramas y los símbolos utilizados en ellos. Cada tipo de diagrama se usa en situaciones distintas y tiene su propio subconjunto de símbolos y elementos del lenguaje.

En la actualidad UML es un estándar abierto del grupo OMG (Object Modeling Group, http://www.omg.org), que es un consorcio de empresas dedicado al establecimiento de estándares. Los miembros del OMG son decenas de empresas, gobiernos y universidades.

A.1.1. Elementos del lenguaje UML

A continuación se presenta una clasificación general de los elementos del lenguaje UML. Esos elementos se combinan para construir diferentes tipos de modelos, y se

definen algunos elementos adicionales que intervienen específicamente en estos modelos.

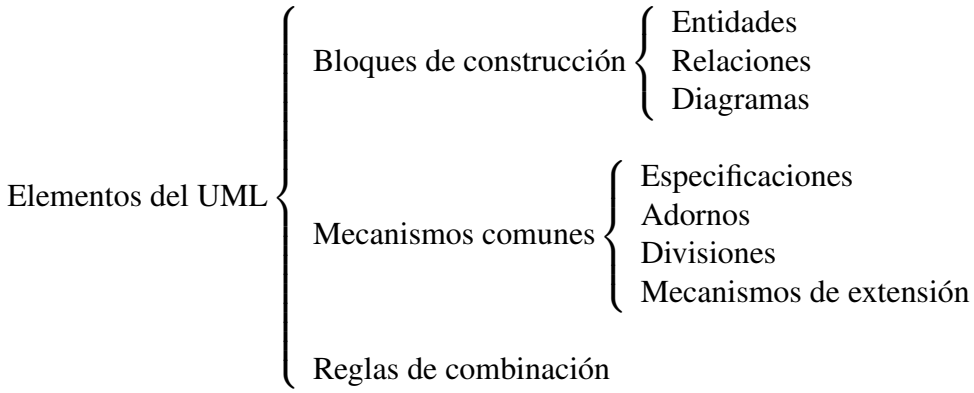


Figura A.1: Estructura general del UML.

Bloques de construcción

Los bloques de construcción se definen como abstracciones y tienen manifestaciones concretas denominadas *Instancias*. Las instancias más comunes son las instancias de clase (*Objetos*) y las instancias de asociación (*Enlaces*). Las instancias pueden identificarse mediante un nombre o bien ser anónimas.

- Las entidades son los bloques básicos para construir modelos orientados a objetos (véase A.2).
- Las relaciones unen las entidades. Hay tres tipos: asociaciones, generalizaciones y dependencias.
- Los diagramas agrupan colecciones de entidades.

Mecanismos comunes

Son características que dan consistencia al lenguaje:

- *Especificaciones*: descripciones textuales detalladas de la sintasix y semántica de un bloque de construcción.
- *Adornos*: elementos gráficos y textuales que pueden añadirse a la notación gráfica básica para proporcionar detalles adicionales de la especificación: símbolos de visibilidad, compartimentos adicionales, notas, roles, representación de clases y características especiales.

- *Divisiones*: existe una dicotomía entre ciertos elementos del lenguaje: 1) Del tipo clases y objetos. Una clase es una abstracción y un objeto una manifestación de esa abstracción. Ocurre lo mismo entre casos de uso e instancias de casos de uso, componentes e instancias de componentes, etc. 2) Del tipo interface e implementación. Una interfaz es una declaración y una implementación es una realización de esa declaración. Otros ejemplos de este tipo de dicotomía son casos de uso y las colaboraciones que los realizan, operaciones y los métodos que los implementan, etc.

- *Mecanismos de extensión*: posibilidades de extensión controlada del lenguaje paar adecuarse a aplicaciones, procesos o dominios de aplicación concretos.

Reglas de combinación

Los bloques de construcción del UML no se pueden poner juntos aleatoriamente; las reglas especifican como se construye un modelo bien formado. En UML hay reglas semánticas para los nombres, el alcance, la visibilidad, la integridad y la ejecución que indican cómo se dará nombre a los componentes, la forma en la que serán referenciados y vistos desde fuera y cómo se podrán utilizar. Estas reglase están implícitas en la descripción de los diagramas.

Las *Entidades* del UML son las abstracciones que se pueden identificar en un modelo. Hay varios tipos como se ve en el diagrama A.2.

A.2. Diagramas de UML

En las secciones siguientes describimos los diferentes tipos de modelos que pueden definirse en UML, indicando también los elementos que intervienen en los correspondientes diagramas.

En UML 2 existen trece diagramas agrupados en tres categorías, véase la figura A.3.

A.2.1. Modelado del comportamiento

Un modelo de comportamiento describe la evolución del estado del sistema y las interacciones que tienen lugar entre sus elementos. Se expresa mediante diagramas de interacción, de los que existen dos tipos: diagramas de colaboración y diagramas

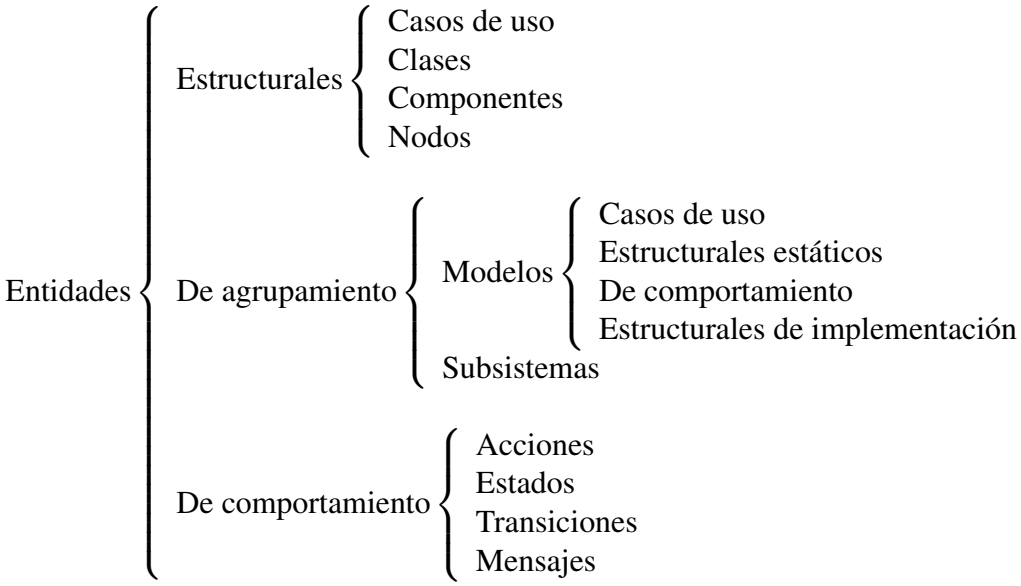


Figura A.2: Entidades del UML.

de secuencias; diagramas de estado y diagramas de actividad. En el modelado del comportamiento, las principales entidades implicadas en los diagramas de interacción son los mensajes, mientras que en los diagramas de estado y actividad las entidades protagonistas son los estados, las transiciones y las acciones. Los cuatro tipos de diagramas de comportamiento pueden implicar asimismo diferentes entidades estructurales, y los diagramas de colaboración exhiben, adicionalmente, asociaciones.

Casos de uso

Los casos de uso especifican los requisitos funcionales del sistema e identifican los escenarios de prueba. Generalmente se utilizan en las primeras etapas de desarrollo. Se definen mediante diagramas de casos de uso y de secuencia, y descripciones textuales (especificaciones) que resultan de cumplimentar unas plantillas estándar. Representan la visión que tendría del sistema un usuario externo y son desarrollados por los analistas con la colaboración de los expertos del dominio de aplicación. Un caso de uso expresa lo que comunmente se denomina una *transacción*, o lo que es lo mismo, una interacción entre el sistema y usuario u otro sistema, que es visto como un "actor" externo.

Los casos de uso pueden tener relaciones de herencia (*«extend»*) y uso (*«include»*) entre ellos.

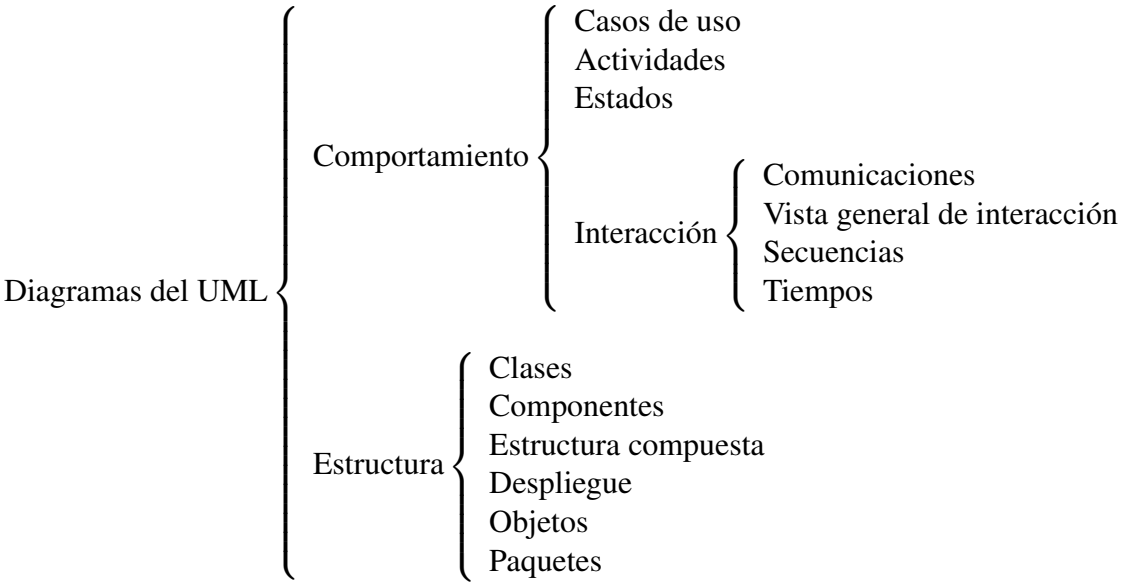


Figura A.3: Clasificación de los tipos de diagramas en el UML.

Diagrama de actividades

Representa procesos de trabajo de alto nivel, incluyendo flujos de datos. Las actividades son realizadas por uno o más casos de uso; la actividad describe el proceso, o se utiliza también para modelar la lógica de un sistema.

Diagrama de estados

Los diagramas de estados muestran el comportamiento de los objetos, es decir, el conjunto de estados por los cuales pasa un objeto durante su vida, junto con los cambios que permiten pasar de un estado a otro. Se usan para modelar el comportamiento de un interfaz, una clase o una colaboración y, por lo tanto, es de un nivel más bajo que el diagrama de actividades A.2.1.

Existen dos estados especiales: inicio (sólo uno) y parada (pueden ser varios). Cada estado es una condición que viene expresada por el valor de ciertos atributos o la presencia de determinadas asociaciones. Mientras la condición de un estado se satisface, o bien se realiza una actividad, o bien se espera la llegada de un evento. Si llega un evento, la actividad que se realiza en un estado puede quedar interrumpida. Las transiciones pueden tener asociadas acciones, que no se consideran interrumpibles.

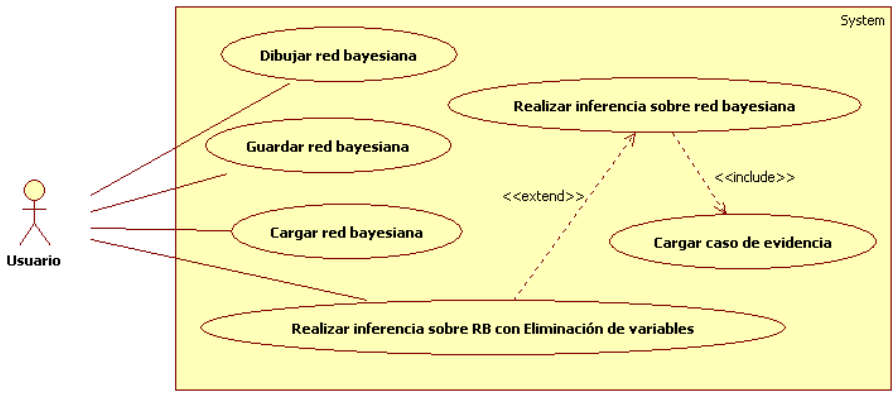


Figura A.4: Ejemplo de casos de uso: el caso de uso *Realizar inferencia sobre red bayesiana* es genérico y de él derivan (*«extend»*) casos de uso que usan determinados algoritmos, por ejemplo, el de eliminación de variables. A su vez, un caso de uso puede usar (*«include»*) las funciones suministradas por otros casos de uso.

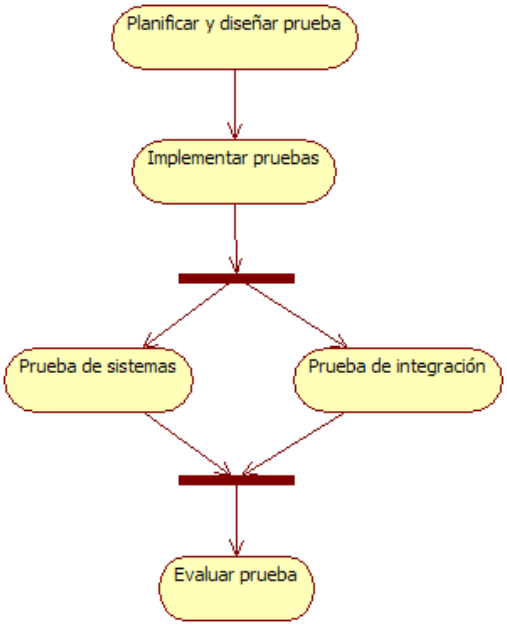


Figura A.5: Diagrama de actividades para realizar pruebas a un sistema.

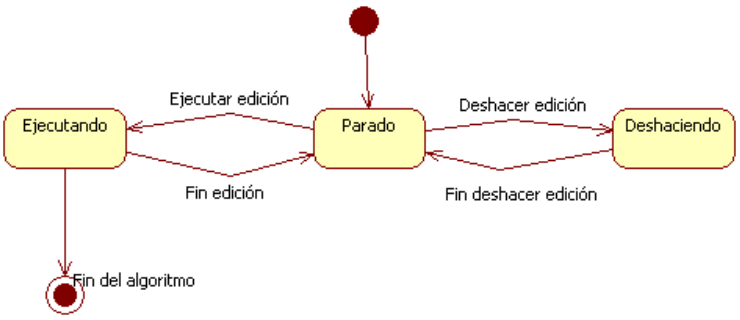


Figura A.6: Diagrama de estados de un algoritmo genérico. Un algoritmo se descompone en una serie de pasos llamados ediciones. Cada edición se ejecuta en función de un evento o se deshace en función de otro evento.

Diagramas de interacción

Los diagramas de interacción son un subconjunto de los diagramas de comportamiento. Expresan la forma en la que las instancias de objetos realizan tareas en el tiempo y cómo intercambian información. Un *mensaje* es la especificación de un conjunto de *estímulos* comunicados entre instancias de clase, de componente o de caso de uso, junto con la especificación de las correspondientes clases, componentes o casos de uso emisores y receptores del mensaje previo que los induce y de los susecuentes procesos desencadenados (nuevos mensajes y acciones).

Los estímulos pueden ser *señales*, entendidas como informaciones de datos o de control comunicadas entre instancias, o invocaciones de operaciones. Una *interacción* es un conjunto de estímulos, intercambiados entre un grupo de instancias (de clases, componentes o casos de uso), en un contexto dado, y con un determinado propósito. Se entiende que dos instancias sólo podrán intercambiar estímulos cuando estén ligadas por algún enlace.

Un Evento es una ocurrencia signicativa para una instancia; en el contexto de los diagramas de interacción son, principalmente, *recepciones de estímulos* y *satisfacción de condiciones*. En un diagrama de interacción figuran:

- *Instancias* (de clases, de componentes o de casos de uso). Entidades con identidad única a las que se puede aplicar un conjunto de operaciones, que tienen un estado y almacenan los efectos de estas operaciones.

- *Acciones*. Existen acciones predefinidas (p.e.: CreateAction, CallAction,

DestroyAction, UninterpretedAction...), y acciones definidas por el desarrollador.

- *Estímulos*. Las invocaciones de operaciones pueden ser de dos tipos:
  - *Síncronas*: las instancias en comunicación tienen que estar sincronizadas, esto es, la instancia que invoca se queda bloqueada hasta recibir una respuesta.
  - *Asíncronas*: las instancias en comunicación no tienen que estar sincronizadas.

  Las señales siempre son asíncronas.
- *Operaciones*. Servicios que pueden solicitarse de las instancias

Como ya hemos mencionado, existen dos tipos de diagramas de interacción: los diagramas de secuencias y los diagramas de colaboración. Además de los elementos antes listados, en los diagramas de colaboración pueden figurar explícitamente enlaces que conectan las diferentes instancias, y atributos de estos enlaces.

Vista general de interacción

Comunicaciones

Muestran las interacciones que tiene lugar entre los objetos, organizadas en términos de los objetos y de los enlaces que los conectan. Proporcionan una visión de la secuencia de interacciones alternativa de la proporcionada por los diagramas de secuencia. Para cada diagrama de comunicaciones existe un diagrama de secuencias equivalente (se puede generar de un modo automático un tipo de diagrama a partir del otro). Es un diagrama más compacto que el de secuencias pero muestra peor la evolución temporal.

Secuencias

Muestra instancias y mensajes intercambiados entre ellas teniendo en cuenta la temporalidad con la que ocurren (lo que comúnmente se denomina escenario en el contexto del software de comunicaciones). Las instancias pueden ser de clases y también de componentes y de casos de uso. Los tres tipos de instancias se ven como objetos únicos, a efectos de estos modelos. Sirven para documentar el diseño, y especificar y validar los casos de uso. Gracias a estos diagramas se pueden identificar los cuellos de botella del sistema, reflexionando sobre los tiempos que consumen los métodos invocados.

En un diagrama de secuencias figuran, además de los elementos básicos de un diagrama de interacción antes presentados.

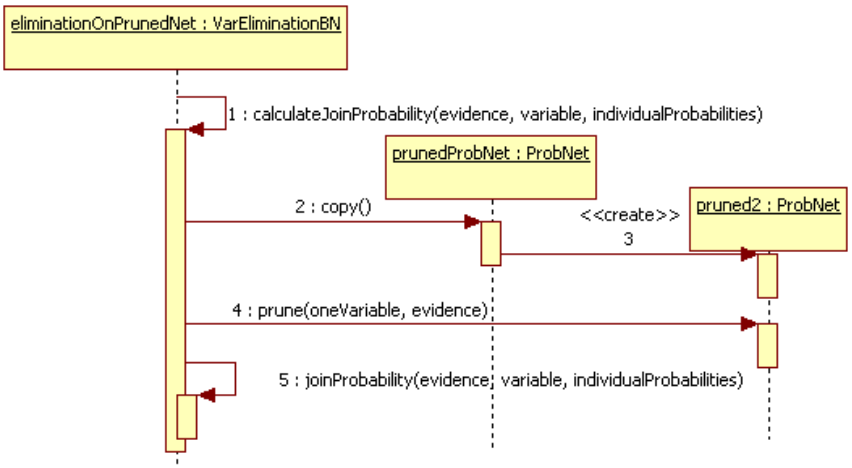


Figura A.7: Ejemplo de un diagrama de secuencias.

1. *Línea de vida de un objeto*: es una representación de la actividad del objeto durante el tiempo en que se desarrolla el escenario. Se entiende que la dimensión temporal crece de arriba a abajo de la línea. Se representa con una cabecera en forma de rectángulo, que incluye el nombre del objeto y una línea vertical de puntos por debajo. Durante el tiempo en que la instancia está ejecutando un método, la línea de puntos se convierte en un rectángulo.

2. *Activación*: punto en que un objeto pasa a tener un método en ejecución, bien por autoinvocación de la correspondiente operación, bien por invocación procedente de otro objeto.

3. *Tiempo de transición*: es el tiempo que transcurre entre dos mensajes.

4. *Condicional*: representa alternativas de ejecución o hilos de ejecución (procesos ligeros) paralelos. Indica que un mensaje sólo se envía si una condición se satisface. La condición se escribe entre corchetes y puede referenciar a otro objeto.

Tiempos

El diagrama de tiempos muestra los cambios en las líneas de vida de los objetos en función del tiempo o la interacción entre los eventos de tiempos, restricciones temporales y duración. Se utilizan en sistemas de tiempo real.

A.2.2. Modelo estructural estático

Las entidades estructurales implicadas en un modelo estructural estático son de dos tipos:

- *Clases*: descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica.
- *Interfaces*: nombre asignado a un conjunto de operaciones que caracterizan el comportamiento de un elemento.

Diagramas de clases

Las clases son abstracciones del dominio de aplicación o bien de la tecnología utilizada en su desarrollo. Constituyen el elemento básico de modelado en el paradigma de orientación a objetos. Una clase se instancia en objetos excepto cuando se define como *abstracta*.

Las clases abstractas se utilizan en niveles altos de las jerarquías de clases para definir abstracciones muy generales de las que se van a derivar otras clases. En una jerarquía de clases existe una clase raíz o base sin super clases, que da origen a la jerarquía, y clases hojas sin subclases. El significado de estas jerarquías está ligado al concepto de herencia.

En la figura A.8 se ilustran las partes de una clase, que se disponen de arriba a abajo en el orden siguiente:

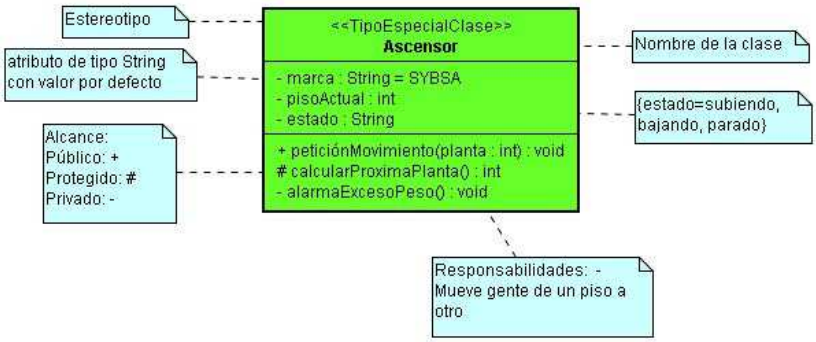


Figura A.8: Plantilla para una clase en UML.

1. *Nombre*: distingue a la clase del resto de clases en el ámbito del modelo. Consiste en una cadena de cualquier longitud que puede contener letras, números y signos

de puntuación (exceptuando . . y .::.). Aparece en cursiva en el caso de las clases abstractas. El nombre de clase puede aparecer opcionalmente precedido del nombre del paquete en que la clase se utiliza; su sintaxis se define pues, en notación BNF: [paquete::]nombre-simple.

2. *Atributos*: representan propiedades características compartidas por todos los objetos de la clase. Una clase puede tener cualquier número de atributos. Un nombre de atributo consiste en una cadena de cualquier longitud que puede contener letras y números. Sobre ellos pueden proporcionarse diferentes informaciones:

   a) *Tipo*: atributo:Tipo

   b) *Valor inicial*: atributo:Tipo=Valor ó Atributo=Valor

   c) *Restricciones*: condiciones semánticas que pueden expresarse utilizando cualquier lenguaje, incluido el lenguaje natural, si bien UML tiene un lenguaje semi-formal asociado para expresar restricciones: el lenguaje OCL. Se encierran siempre entre corchetes.

   d) *Visibilidad o Alcance*: determina en qué medida otras clases pueden hacer uso del atributo. Un atributo público (+) es visible para cualquier clase externa; un atributo protegido (#) es visible para cualquier descendiente en la jerarquía de herencia; y un atributo privado (-) es sólo visible para la propia clase. El valor por defecto de la visibilidad es siempre público.

   e) *Ámbito*: puede ser de dos tipos: de instancia: cuando cada objeto tiene su propio valor, de clase: cuando todas las instancias de una clase comparten un valor. (p.e., las variables *static* en Java). Este tipo de ámbito se indica subrayando el nombre del atributo.

   Toda esta información es opcional, de hecho, es común que en las especificaciones de las clases no se detallen más que los nombres de sus atributos. Su sintaxis se resumen en: [visibilidad]nombre[:tipo][=valorInicial][restricciones]. Atributos derivados son aquellos cuyo valor puede computarse a partir de valores de otros atributos; aunque no añaden información semántica al modelo, se hacen explícitos por claridad o propósitos de diseño.

3. *Operaciones*: Son servicios básicos ofrecidos por los objetos de una clase, que con frecuencia provocan cambios en sus estados (se adornan con query

aquellas operaciones que no tienen ningún efecto secundario, esto es, que no modifican el estado del sistema). Al igual que en el caso de los atributos, pueden especificarse su visibilidad, ámbito y propiedades. Se pueden indicar también el tipo de los argumentos que reciben y el tipo del valor que retornan. Tanto si se especifican sus argumentos como si no, han de añadirse un paréntesis abierto y otro cerrado al final de su nombre, de acuerdo a la sintaxis: [visibilidad]nombre([parámetros])[:tipoRetorno][propiedades]. La sintaxis completa de los parámetros es: [dirección]nombre:tipo[= valorPorDefecto]. La implementación concreta que una clase hace de una operación se denomina Método. La descripción de un método puede realizarse mediante una especificación textual descrita en cualquier lenguaje elegido (p.e., el OCL). Al igual que existen clases abstractas existen Operaciones abstractas, esto es, operaciones que se definen exclusivamente como una signatura. Las clases que contienen este tipo de operaciones requieren subclases que proporcionen métodos para su implementación. Las operaciones de las clases hoja de una jerarquía de clases obligadamente habrán de tener un método asociado. Responsabilidades: Son descripciones de lo que hace la clase en su conjunto. Esta información casi nunca se incluye en los diagramas.

4. *Otras*: Una clase puede incluir partes adicionales predefinidas (p.e., excepciones) o definidas por el usuario.

Interfaces Las interfaces son conjuntos de operaciones que especifican parte de la funcionalidad (un servicio) proporcionada por una clase o componente, con independencia de su implementación. Pueden verse como contratos entre los clientes de la interfaz y sus implementaciones; el programador responsable de la interfaz puede optar por implementar una nueva o bien adquirir o reutilizar otra implementación, con tal de que el estipulado contrato se cumpla.

Una interfaz y la clase que la implementa están relacionadas mediante una realización (un tipo particular de relación de dependencia), donde una clase puede realizar varias interfaces y una interfaz puede realizarse por más de una clase. Por otro lado, una clase .cliente. puede usar varias interfaces con las que estará relacionada mediante una relación de uso (otro tipo de relación de dependencia).

Las interfaces establecen conexiones entre los componentes de una aplicación, y representan los principios de modularidad y ocultación necesarios en diseños orientados

a objetos.

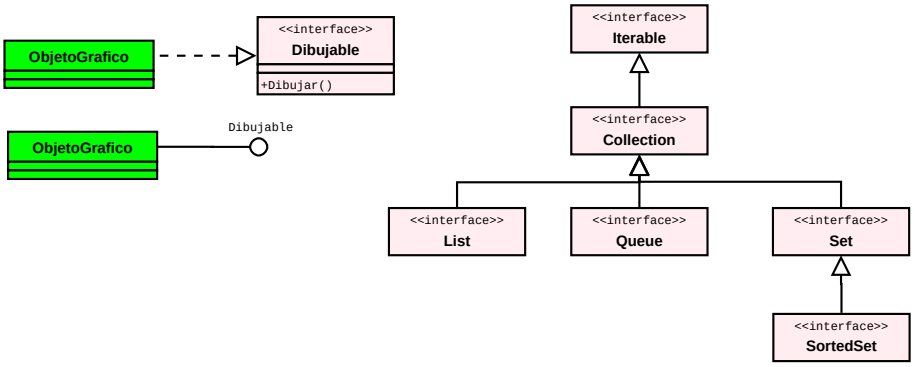


Figura A.9: Interfaces

En cuanto a su representación diagramática, las interfaces se muestran como clases sin atributos mediante dos representaciones alternativas que se muestran en la figura 1.14: como clases con estereotipo*«interfaz»* y, en forma abreviada, sin mostrar las operaciones. Al tratarse de un tipo particular de clases, entre las interfaces pueden darse relaciones de herencia. Todas las operaciones de una interfaz son públicas.

Clases de análisis y clases de diseño

En función de que la fase de desarrollo del proyecto sea el análisis o el diseño, la información se representa de distinto modo.

Modelo de análisis: diagramas de robustez El análisis especifica *qué* se debe hacer y omite intencionadamente todos los detalles sobre *cómo* hacerlo buscando la máxima simplicidad posible. En el modelo de análisis de Jacobson, los diagramas de clases del análisis se denominan diagramas de robustez. Estos diagramas se centran en los requisitos funcionales y aunque tiene relaciones, son de tipo conceptual. Existen tres estereotipos básicos: frontera, entidad y control.

- *Frontera*: en un caso de uso, son los objetos que interactúan con el actor.
- *Control*: gestionan la interacción entre los objetos que realizan un caso de uso.
- *Entidad*: representan el dominio del problema.

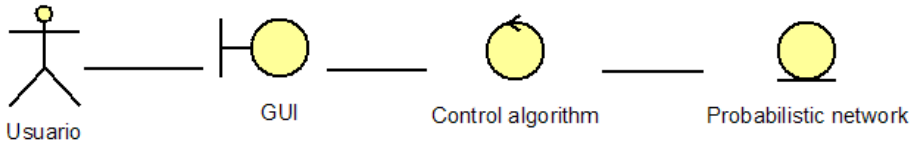


Figura A.10: Diagrama de robustez

Modelo de diseño: diagramas de clases Describen tipos de objetos en un sistema y sus relaciones. Una clase del análisis se corresponde con una o varias clases del diseño.

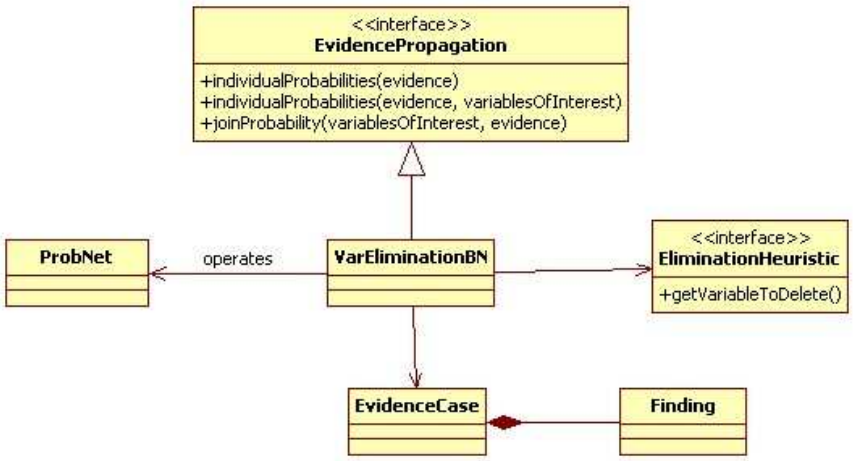


Figura A.11: Ejemplo de diagrama de clases en UML

Asociaciones Una asociación es una relación entre dos o más clases. Existen varios tipos de asociaciones:

- *Herencia*: una clase A_1 es un subconjunto de otra A redefiniendo su comportamiento.
- *Realización*: una clase implementa un determinado interfaz.
- *Agregación*: una clase está relacionada con una o varias instancias de otra.
- *Composición*: los objetos de una clase forman parte de otra (o son de su propiedad).

- *Dependencia*[1]: las características de una clase *A* influyen en el funcionamiento de otra *B* que la utiliza.

Navegabilidad En una asociación entre dos clases, por defecto cada una puede acceder a la otra; si el acceso es unidireccional se indica con una flecha.

Cardinalidad Una asociación puede indicar el número de elementos que están involucrados en cada lado de la relación, esto se conoce como *cardinalidad*. La cardinalidad en cada uno de los extremos se indica con uno o dos números. En este segundo caso, el primer número indica el mínimo número de elementos y el segundo el un máximo. El símbolo * significa "cualquier número".

Nombre En general, una asociación puede tener un nombre que indique la forma en la que se relacionan las clases involucradas.

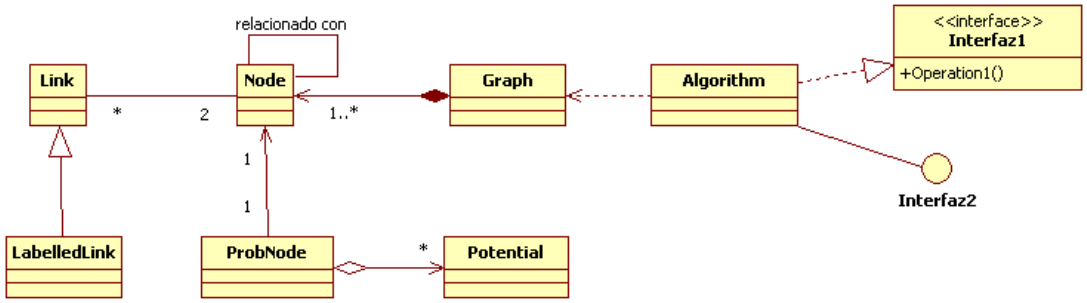


Figura A.12: Clases con varios tipos de asociaciones y cardinalidades.

En la figura A.12 la clase *LabelledLink* es un tipo especial de *Link*, de la que hereda las características generales. La clase *Algorithm* implementa dos interfaces, uno de ellos tiene la notación abreviada (*Interfaz2*) y el otro la textual (*Interfaz1*). Un *ProbNode* está relacionado con un número indeterminado (de cero a muchos) de *Potential* y un *Graph* posee varios *Node*. El funcionamiento de la clase *Algorithm* depende de cómo esté implementada la clase *Graph*. Un *Link* relaciona entre si dos *Node*, y cada *Node* puede tener un número indeterminado de *Link*. En el enlace que hay entre *ProbNode* y *Node*, un *ProbNode* puede acceder a un *Node* y a sus funciones pero no al revés. En el enlace

[1]Las relaciones de dependencia la mayor parte de las veces se pueden deducir de los otros tipos de relaciones y en general no se incluyen en los diagramas para no hacerlos demasiado farragosos.

entre *Link* y *Node*, un *Node* puede tener un número indeterminado de *Link*s pero un enlace consta exactamente de dos *Node*. En el enlace entre *Graph* y *Node*, un *Graph* posee como mínimo un nodo y como máximo muchos.

Bibliografía

[1] H. Akaike. A new look at statistical model identification. *IEEE Transactions on Automatic Control*, **19**:716–723, 1974. 22

[2] C. Alexander, S. Ishikawa y M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. OUP USA, 1977. 36

[3] M. Arias y F. J. Díez. Carmen: An open source project for probabilistic graphical models. En: *Proceedings of the Fourth European Workshop on Probabilistic Graphical Models (PGM'08)*, págs. 25–32, Hirtshals, Denmark, 2008. XII, 147, 193

[4] S. Arnborg, D. Corneil y A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, **8**(2):277–284, 1987. 159

[5] K. J. Åström. Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications*, **10**:174–205, 1965. 31, 198

[6] L. Bass, P. Clements y R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, Boston, MA, 1997. 115

[7] K. Beck y M. Fowler. *Planning Extreme Programming*. Addison-Wesley Professional, Boston, MA, 2000. 38

[8] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957. 31

[9] R. R. Bouckaert. Bayesian networks in Weka. Technical Report 14/2004, Computer Science Department, University of Waikato, New Zealand, 2004. 109

[10] W. Buntine. Theory refinement on Bayesian networks. En: *Proceedings of the 7th Conference on Uncertainty in Artificial Intelligence (UAI'91)*, págs. 52–60, Los Angeles, CA, 1991. Morgan Kaufmann, San Mateo, CA. 22

[11] A. Cano y S. Moral. Heuristic algorithms for the triangulation of graphs. En: B. Bouchon-Meunie, R. R. Yager y I. A. Zadeh (eds.), *Advances in Intelligent Computing (IPMU-94)*, págs. 98–107. Springer-Verlag, Berlin, 1995. 146, 160

[12] A. Cano, S. Moral y A. Salmerón. Penniless propagation in join trees. *International Journal of Intelligent Systems*, **15**:1027–1059, 2000. 67

[13] E. Castillo, J. M. Gutiérrez y A. S. Hadi. *Expert Systems and Probabilistic Network Models*. Springer-Verlag, New York, 1997. Versión española: *Sistemas Expertos y Modelos de Redes Probabilísticas*, Academia de Ingeniería, Madrid, 1997. 11, 44

[14] D. M. Chickering. Search operators for learning equivalence classes of Bayesian network structures. Technical Report, R231, UCLA Cognitive Systems Laboratory, Los Angeles, 1995. 22

[15] G. F. Cooper. A method for using belief networks as influence diagrams. En: *Proceedings of the 4th Workshop on Uncertainty in AI*, págs. 55–63, University of Minnesota, Minneapolis, MN, 1988. 29

[16] G. F. Cooper y E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, **9**:309–348, 1992. 20, 22

[17] J. Coplien. *Pattern Language of Program Design*. Addison Wesley, Boston, MA, 1995. 37, 145

[18] R. G. Cowell, A. P. Dawid, S. L. Lauritzen y D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer-Verlag, New York, 1999. 44

[19] A. Darwiche. Any-space probabilistic inference. En: *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI-2000)*, págs. 133–142, Madison, WI, 2000. Morgan Kaufmann, San Francisco, CA. 58

[20] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. En: *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI'96)*, págs. 211–219, Portland, OR, 1996. Morgan Kaufmann, San Francisco, CA. 66

[21] F. J. Díez. *Introducción a los Modelos Gráficos Probabilistas*. UNED, Madrid, 2007. 100

[22] F. J. Díez y M. J. Druzdzel. Canonical probabilistic models for knowledge engineering. Technical Report CISIAD-06-01, UNED, Madrid, Spain, 2006. 171, 173

[23] F. J. Díez y S. F. Galán. Efficient computation for the noisy MAX. *International Journal of Approximate Reasoning*, **18**:165–177, 2003. 173

[24] Michael F. Drummond, Mark J. Sculpher, George W. Torrance, Bernie J. O'Brien y Greg L. Stoddart. *Methods for the Economic Evaluation of Health Care Programmes*. Oxford University Press, USA, July 2005. 74, 75

[25] B. Eckel. Thinking in patterns: Problem-solving techniques using Java, 2003. 3, 145

[26] The Elvira Consortium. Elvira: An environment for creating and using probabilistic graphical models. En: *Proceedings of the First European Workshop on Probabilistic Graphical Models (PGM'02)*, págs. 1–11, Cuenca, Spain, 2002. 108

[27] R. A. Finkel y J. L. Bentley. Quad Trees: A data structure for retrieval on composite keys. *Acta Informatica*, **4**:1–9, 1974. 67

[28] M. Fowler. *Analysis patterns: reusable objects models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. 36

[29] E. Gamma, R. Helm, R. Johnson y J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 2005. 3, 36

[30] D. Geiger y D. Heckerman. A characterization of the Dirichlet distribution with application to learning Bayesian networks. En: *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, págs. 196–207. Morgan Kaufmann Publishers, San Francisco, CA, 1995. 20

[31] G. H. Golub y C. F. Van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, MD, tercera edición, 1996. 66

[32] R. A. Howard y J. E. Matheson. Influence diagrams. En: R. A. Howard y J. E. Matheson (eds.), *Readings on the Principles and Applications of Decision Analysis*, págs. 719–762. Strategic Decisions Group, Menlo Park, CA, 1984. 27, 29

[33] C. Huang y A. Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, **15**:225–263, 1996. 67

[34] Ivar Jacobson, Grady Booch y James Rumbaugh. *El proceso unificado de desarrollo de software*. Addison-Wesley, 2000. 33

[35] F. V. Jensen y T. D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer-Verlag, New York, segunda edición, 2007. 44

[36] F. V. Jensen, K. G. Olesen y S. K. Andersen. An algebra of Bayesian belief universes for knowledge-based systems. *Networks*, **20**:637–660, 1990. 66, 67, 164

[37] J. Kerievsky. *Refactoring To Patterns*. Addison Wesley, Boston, MA, 2004. 4

[38] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier y J. Irwin. Aspect oriented programming. volume 1241, págs. 220–242, 1997. 32

[39] R. Korf. Linear-space best-first search. *Artificial Intelligence*, **62**, 1993. 25

[40] C. Lacave, M. Luque y F. J. Díez. Explanation of Bayesian networks and influence diagrams in Elvira. *IEEE Transactions on Systems, Man and Cybernetics—Part B: Cybernetics*, **37**:952–965, 2007. 175, 198

[41] C. Lacave, A. Oniśko y F. J. Díez. Use of Elvira's explanation facilities for debugging probabilistic expert systems. *Knowledge-Based Systems*, **19**:730–738, 2006. 175

[42] S. Lauritzen y D.Ñilsson. Representing and solving decision problems with limited information. *Management Science*, **47**:1238–1251, 2001. 31, 197

[43] S. L. Lauritzen y D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B*, **50**:157–224, 1988. 66

[44] V. Lepar y P. P. Shenoy. A comparison of Lauritzen-Spiegelhalter, HUGIN, and Shenoy-Shafer architectures for computing marginals of probability distributions. En: *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI'98)*, págs. 328–337, Madison, WI, 1998. Morgan Kaufmann, San Francisco, CA. 164

[45] Z. Li y B. DÁmbrosio. Efficient inference in Bayes nets as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, **11**:55–81, 1994. 66

[46] M. Luque. *Probabilistic Graphical Models for Decision Making in Medicine.* Tesis doctoral, Universidad Nacional de Educación a Distancia, Madrid, 2009. 196, 197

[47] M. Luque y F. J. Díez. Variable elimination for influence diagrams with super-value nodes. En: P. Lucas (ed.), *Proceedings of the Second European Workshop on Probabilistic Graphical Models*, págs. 145–152, 2004. 5, 103, 197

[48] M. Luque y F. J. Díez. Representación de problemas de decisión con asimetrías de orden. En: *Actas del II Simposio de Inteligencia Computacional*, 2007. 197, 198

[49] M. Luque y F. J. Díez. Variable elimination for influence diagrams with super-value nodes. Technical Report DIA-08-01, Dpto. Inteligencia Artificial, UNED, Madrid, Spain, 2008. 5

[50] A. Madsen y F. V. Jensen. Parallelization of inference in Bayesian Networks. Technical Report R–90–09, University of Aalborg, Denmark, 1999. 198

[51] A. L. Madsen y F. V. Jensen. Lazy propagation in junction trees. En: *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI'98)*, págs. 362–369, Madison, WI, 1998. Morgan Kaufmann, San Francisco, CA. 66

[52] A. L. Madsen y F. V. Jensen. Lazy propagation: A junction tree inference algorithm based on lazy evaluation. *Artificial Intelligence*, **113**:203–245, 1999. 4, 164, 170

[53] A. C. McKellar y E. G. Coffman. Organizing matrices and matrix operations for paged-memory systems. *Communications of the ACM*, **12**:153–165, 1969. 67

[54] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller y E. Teller. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, **21**, 1953. 25

[55] Ministerio de Administraciones Públicas. Métrica versión 3: Metodología de planificación, desarrollo y mantenimiento de sistemas de información. 33

[56] S. Moral y A. Salmerón. Dynamic importance sampling in Bayesian networks based on probability trees. *International Journal of Approximate Reasoning*, **38**:245–261, 2005. 197

[57] R. E. Neapolitan. *Learning Bayesian Networks*. Prentice-Hall, Upper Saddle River, NJ, 2004. 20, 180, 197

[58] S. H. Nielsen, T. D. Nielsen y F. V. Jensen. Multi-currency influence diagrams. En: A. Salmerón y J. A. Gámez (eds.), *Advances in Probabilistic Graphical Models*, págs. 275–294. Springer, Berlin, Germany, 2007. 103

[59] S. M. Olmsted. *On Representing and Solving Decision Problems*. Tesis doctoral, Dept. Engineering-Economic Systems, Stanford University, CA, 1983. 29, 101, 197

[60] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988. 44

[61] Roger S. Pressman. *Ingeniería de Software: un Enfoque Práctico*. McGraw-Hill, 2005. 34

[62] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, New York, 1994. 198

[63] D. E. Rumelhart y J. L. McClelland (eds.). *Parallel Distributed Processing*. MIT Press, Cambridge, MA, 1986. 198

[64] G. Schwarz. Estimating the dimension of a model. *Annals of Statistics*, **17(2)**:461–464, 1978. 22

[65] R. D. Shachter. Evaluating influence diagrams. *Operations Research*, **34**:871–882, 1986. 29, 101, 197

[66] R. D. Shachter, B. D'Ambrosio y B. A. Del Favero. Symbolic probabilistic inference in belief networks. En: *Proceedings of the 8th National Conference on AI (AAAI–90)*, págs. 126–131, Boston, MA, 1990. 66

[67] A. Shalloway y J. R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, Boston, MA, segunda edición, 2004. 3, 132, 133

[68] P. P. Shenoy y G. Shafer. Axioms for probability and belief-function propagation. En: R. D. Shachter, T. S. Levitt, L.Ñ. Kanal y J. F. Lemmer (eds.), *Uncertainty in Artificial Intelligence 4*, págs. 169–198. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990. 66, 164

[69] Ian Sommerville. *Ingeniería del Software*. Prentice Hall, 2005. 32, 34

[70] Bert Spilker y Joyce Cramer. *Quality of life and Pharmaeconomics: An introduction*. Lippincott-Raven, 1998. 76

[71] P. Spirtes y C. Glymour. An algorithm for fast recovery of sparse causal graphs. *Social Science Computer Review*, **9**:62–72, 1991. 20

[72] P. Spirtes, C. Glymour y R. Scheines. *Causation, Prediction and Search*. The MIT Press, Cambridge, MA, segunda edición, 2000. 20

[73] P. Spirtes y C. Meek. Learning Bayesian networks with discrete variables from data. En: *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, págs. 294–300, Menlo Park, CA, 1995. AAAI Press. 22

[74] J. A. Tatman y R. D. Shachter. Dynamic programming and influence diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, **20**:365–379, 1990. 28

[75] G. W. Torrance. Utility approach to measuring health-related quality of life. *Journal of Chronic Diseases*, **40**:593–600, 1987. 75

[76] M. A. J. van Gerven, F. J. Díez, B. G. Taal y P. J. F. Lucas. Selecting treatment strategies with dynamic limited-memory influence diagrams. *Artificial Intelligence in Medicine*, **40**:171–186, 2007. 31, 197, 198

[77] W. X. Wen. Optimal decomposition of belief networks. En: P. P. Bonissone, M. Henrion, L.Ñ. Kanal y J. F. Lemmer (eds.), *Uncertainty in Artificial Intelligence 6*, págs. 209–224. Elsevier Science Publishers, Amsterdam, The Netherlands, 1991. 64

[78] I. H. Witten y E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. San Francisco, 2005. 108

[79] M. Yannakakis. Computing the minimal fill-in is NP-complete. *SIAM Journal of Algebraic Discrete Methods*, **2**:77–79, 1981. 146