Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques
Universitat de Barcelona

MODELOS GRÁFICOS PROBABILíSTICOS EN SISTEMAS DISTRIBUIDOS

Guillermo Blasco Jiménez

Director: Jesús Cerquides Bueno,
Marc Pujol González
Realitzat a: Departament de Matemàtica
Aplicada i Anàlisi. UB

Barcelona, 23 de gener de 2015

Abstract

Modern companies have recognized machine learning techniques as a key instrument to gain competitive edges in their businesses, from better client profiling to optimization and streamlining of their resources and operations. Among the many approaches and problems defined under the machine learning umbrella, we focus on Probabilistic Graphical Models (PGM). PGM is a generic framework that allows analysts to harness collected statistics and make predictions using them. Nonetheless, the computation of such predictions is a known NP-hard problem, and hence presents a significant challenge. Therefore, the PGM community has mainly focused on the development of either useful relaxations that can be solved with a lower computational complexity or approximate algorithms that still provide good solutions.

Meanwhile, the relentless technological advances of our era have brought us to an interesting situation. On the one hand, the sheer amount of collected data that can be analyzed and exploited is ever-increasing. On the other hand, the advent of cloud computing provides vast amounts of inexpensive computation capabilities. However, exploiting these resources is not an easy endeavor because they are distributed along many networked machines. Fortunately, the community response against these challenges has been the development of generic distributed computation platforms such as Hadoop or Spark. These platforms free the developers from dealing with the distribution challenges, allowing them to focus on the implementation of their algorithms instead.

Against this background, the main goal of this project is to build a bridge between these worlds by implementing a core PGM algorithm in one of such distributed systems. The selected algorithm is Belief Propagation (BP), an approximate inference algorithm that has good complexity characteristics and has proven to provide good solutions in practice. Additionally, BP is based on a message-passing model, and hence stands as a perfect candidate for a distributed implementation.

Furthermore, the implementation is done under Apache Spark, because it is designed specifically to support message-passing algorithms. Nonetheless, despite the invaluable foundation provided by the platform, Spark leaves many considerations left to the criteria of the developer. Therefore, in this project we present the challenges that arose during such implementation, and how they can be overcome.

The end result of this project is hence a BP implementation that can run in a distributed manner between as many systems as supported by Spark. Moreover, the implementation is both unit-tested and checked against a centralized open-source library. This opens up the possibility for anyone to make predictions based on large PGM models built spanning gigabytes of information.

Índice

1. Introducción

Este proyecto pretende poner en práctica los conocimientos de un márco teórico matemático usado en el campo del aprendizaje máquina llamado Modelos Gráficos Probabilísticos implementándolo en un sistema de computación distribuida moderno, Apache Spark, mediante el planteamiento de un problema realista. La motivación es el potencial que ambos elementos, PGM y Spark, tienen hoy en día, así como el auge del aprendizaje máquina en la indústria digital moderna. Para llegar a esta meta primero se debe plantear el problema, luego comentar los aspectos teóricos del modelo matemático, después los aspectos técnicos de la tecnología y finalmente los detalles de la solución, es decir, la implementación. El resultado esperado es un código capaz de resolver la problema planteado.

1.1. El problema

Supongamos que hay una empresa de la indústria digital que gestiona y mantiene diferentes aplicaciones web. Debido a que las aplicaciones web fueron debidamente diseñadas y programadas generan *logs*, es decir, información asociada a un evento que sucede en el sistema junto con la fecha del evento. Los logs tienen la propiedad de que una vez escritos no se deben modificar y su borrado siempre es problemático, pues en los logs está la historia del sistema y de él se puede extraer información como entender los errores de los sistemas o detectar el uso fraudulento de éstos. Como es una empresa de aplicaciones web, los logs contienen toda información referente a las conexiones, como la ip del usuario, del servidor, el tiempo del sistema, el identificador del usuario, el país de origen de la petición, el tiempo de respuesta del servidor, el formato de la respuesta, el tamaño en bytes de la respuesta y de la petición, el tipo de evento (compra de un producto, modificación del perfil, log-in,...) entre otros muchos datos. Al principio se interesaron por averiguar cierta información del sistema básica. Por ejemplo, cuál era la distribución del tiempo de respuesta del servidor según el país del cliente de la aplicación web. Pronto se hicieron preguntas más complejas, como por ejemplo, cuál es el tiempo de respuesta condicionado al tamaño en bytes de la respuesta y al formato de la respuesta. En definitiva, la cantidad de preguntas fue en aumento, a medida que entendían mejor su sistema. Sin embargo, a medida que el volumen de datos de los logs aumentaba su algoritmo de análisis dejaba de ser usable: no escalaba. Lo que hacían era por cada pregunta programar un script que leía todos los logs almacenados y según el tipo de pregunta efectuaba el recuento estadístico adecuado. Por ejemplo, para su priemra pregunta al leer cada log tenían que extraer el país de la ip, generar una lista de países y luego asociar el tiempo de respuesta que consta en un log al país de la ip que aparecee en el log. Luego hacer el recuento en las listas de cada país.

Naturalmente el proceso de creación y comprobación de los scripts era lento y tedioso. Además de que cada vez era más y más lento debido al creciente volumen de los logs. Por tanto, optaron por desarrollar otra herramienta más

eficiente que les permitiese seguir respondiendo a sus preguntas de una forma más barata y escalable. Por ello deciden usar machine learning en un sistema distribuido, para poder generar un modelo reutilizable de la fuente de datos masiva. Debido a que buscan respuestas rápidas, no es necesario un resultado exacto, basta con que sea aproximado pero que no tenga un coste computacional alto.

1.2. Objetivos

Este documento propone una solución para este tipo de problemas y los retos anteriormente expuestos. El algoritmo propuesto para responder a estas preguntas es Belief Propagation por dos motivos:

- Es un algoritmo de inferencia aproximado e iterativo con buenas características de complejidad
- Puede formularse mediante el patrón de paso de mensajes y por lo tanto implementable en un sistema de computación distribuida como Spark

Cabe decir, que el algoritmo es sólamente una parte de la aplicación completa que el caso de uso anteriormente expuesto necesitaría. En lugar de proponer una solución completa de principio a fin el objetivo es analizar en concreto el algoritmo y su implementación debido al interés explicado en el primer párrafo de la introducción. El resto de componentes, como la interfície de usuario, el almacenamiento de los datos, el despligue de la aplicación, entre otros, quedan fuera del propósito de este proyecto. Por lo que los objetivos serán los siguientes:

- Aprender y exponer los conocimientos necesarios para entender el algoritmo Belief Propagation
- Analizar las posibles soluciones tecnológicas en cuanto al sistema de computación
- Implementar y testear el algoritmo en una de estas soluciones tecnológicas

1.3. Planificación

Para llevar a cabo el proyecto se desarrolló una planificación como la que figura en la tabla 1. Aunque no ha habido desviaciones significativas la implementación y la elaboración de la memoria llevarn más tiempo del previsto. Además de las tareas se planificó la asistencia a los siguientes eventos relacionados con la comunidad de aprendizje máquina y sistemas de computación distribuidos en Barcelona con el objetivo de reforzar los conocimientos y conocer experiencias de estos sectores. Los eventos que figuran en tabla 2 pertenecen al *Grup d'estudi de machine learing de Barcelona*[1] y el *Barcelona Spark Meetup*[2] y las conferencias PAPIs.io[3] y NoSql Matters Bcn[4].

[1] http://www.meetup.com/Grup-destudi-de-machine-learning-de-Barcelona
[2] http://www.meetup.com/Spark-Barcelona/
[3] http://www.papis.io/
[4] http://2014.nosql-matters.org/bcn/homepage/

Número de tarea	Objeto de la tarea	Fecha de inicio	Duración (semanas)
1	Recolección de referencias de PGM	primera semana de junio de 2014	3
2	Recolección de referencias de Sistemas de computación distribuida	segunda semana de junio de 2014	2
3	Lectura de las referencias principales de PGM [1, 5]	tercera semana de junio de 2014	5
4	Lectura de las referencias de sistemas de computación distribuida [10, 2, 7, 18, 16]	primera semana de agosto de 2014	4
5	Aprendizaje y experimentos de Spark	primera semana de septiembre de 2014	4
6	Análisis del algoritmo Belief Propagation	primera semana de octubre de 2014	3
7	Primeros experimentos de la implementación	tercera semana de octubre de 2014	2
8	Desarrollo de la implementación definitiva	primera semana de noviembre de 2014	5
9	Elaboración de la memoria	segunda semana de noviembre de 2014	6
10	Preparación de la release de la implementación	tercera semana de diciembre de 2014	3

Cuadro 1: Planificación

Fecha	Nombre del evento
18 de septiembre de 2014	Asistencia a «Forecasting financial time series with machine learning models and Twitter data» del grupo Grup d'estudi de machine learning de Barcelona
22 de septiembre de 2014	Asistencia a «Third Spark Barcelona Meeting» del grupo de Barcelona Spark Meetup
9 de octubre de 2014	Asistencia a «Introduction to probabilistic graphical models» del grupo Grup d'estudi de machine learning de Barcelona
17 y 18 de noviembre de 2014	Asistencia a PAPIs.io conference
20 de noviembre de 2014	Asistencia a «Databricks comes to Barcelona» del grupo Barcelona Spark Meetup
21 y 22 de noviembre de 2014	Asistencia a NoSql Matters Bcn conference

Cuadro 2: Tabla de eventos

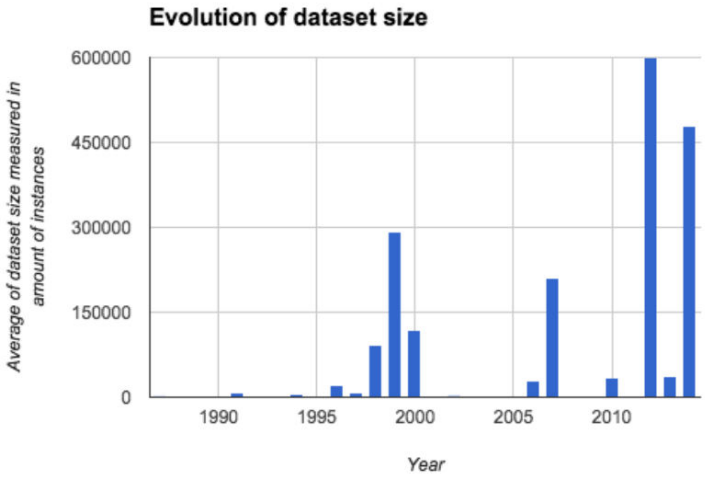

Figura 2.1: Evolution of dataset size. Elaboración propia. Fuente de los datos: UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets.html)

2. Estado del arte de sistemas de computación distribuida

El tamaño de los datasets en machine learning está creciendo como se puede observar en el gráfico 3.1 lo que hace necesario un forma escalable de almacenarlos. Por otro lado la complejidad de los modelos también aumentan con los años. Por ejemplo, en el gráfico 3.2 se puede observar como la cantidad de features que un dataset incorpora aumenta con los años, pero hay otras muchas evidencias como la creciente resolución de las imágenes que implican modelos más complicados (en el caso de las redes neuronales más capas de kernels para la extracción de características o en el caso de markov networks más factores y por ende más dependencias entre factores), la popularización del análisis de series temporales. Por lo que también será necesaria la escalabilidad del almacenamiento del modelo.

Ambos objetos -el dataset y el modelo- tienen necesidades diferentes en cuanto a lectura/escriura, magnitudes diferentes (el dataset siempre será ordenes de magnitud mayor que el modelo) y estructuras diferentes. Estas características de cada objeto serán analizadas en los apartados posteriores dando soluciones particulares para cada uno.

Aquí se propone lograr esa escalabilidad mediante los sistemas distribuidos usados en la última década para reducir los costes de infraestructura, aunque no se hace un estudio cuantitativo de esta reducción. En el gráfico 3.3. se muestra la

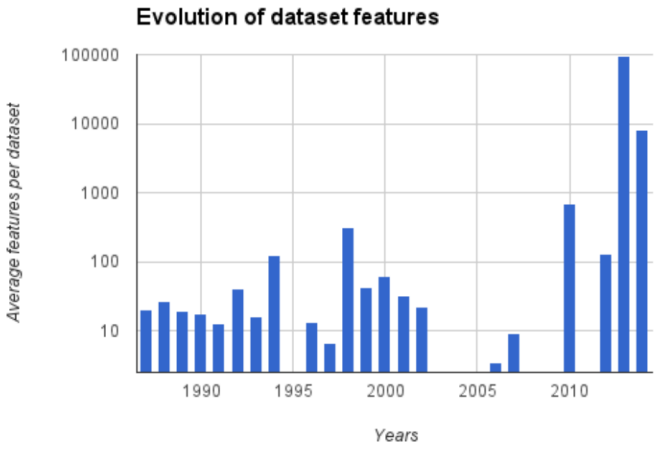

Figura 2.2: Evolution of dataset complexity. Elaboración propia. Fuente de los datos: UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets.html)

evolución de la popularidad de Hadoop, que es un framework de computación distribuida, como palabra de búsqueda en Google. Este software ha sido usado para escalar la capacidad de computación de un sistema respecto el escalado del tamaño de los datos. Por ejemplo, la empresa Yahoo! comenzó con un clúster de 4 nodos entre 2002 y 2004; al menos 20 nodos entre 2004 y 2006 [12]; 4000 nodos en 2008 [13] y 42000 nodos en 2013[14].

Evolution of popularity of "Hadoop" in Google

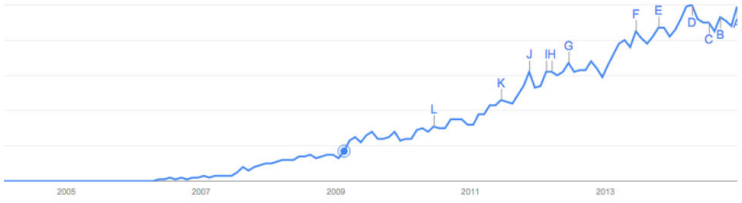

Figura 2.3: Evolution of popularity of Hadoop in Google. Source: Google Trends (http://www.google.com/trends/explore#q=hadoop)

2.1. Almacenamiento de datos

Los datasets en el campo de machine learning son evidencias producidas por ciertas fuentes que consisten en datos estructurados o semiestructurados. El dataset es usado para aprender un modelo y luego efectuar inferencia sobre éste. Estos son los dos componentes que han de ser almacenados: el dataset y el modelo. El dataset puede representar tipos de información muy diferentes: grafos de redes sociales, datos geoespaciales de vehículos, series temporales de sensores, El tipo de entidades que el dataset contiene determinará cuál es el mejor sistema para ser almacenados. No obstante, el aprendizaje de un modelo sobre un dataset implica efectuar una lectura intensiva de los datos.

En general, es conveniente almacenar el dataset en un sistema distribuido de ficheros, como HDFS, dado que ofrece un acceso de lectura rápido, escalable y distribuido. Almacenar datasets en bases de datos relacionales es una buena idea sólamente si hay un modelo relacional estricto y el volumen de datos es pequeño. Aún así, dado que el objetivo es que sea una solución escalable en cuanto a volumen de datos, la escalabilidad horizontal de las bases de datos relacionales implica difíciles decisiones arquitectónicas, como el *sharding*. Por lo que soluciones de bases de datos no SQL pueden ser mejores soluciones en cuanto a escalabilidad aún manteniendo cierta estructura declarada en los datos, como Cassandra o MongoDb. Hay bases de datos orientadas a las relaciones, ideales para almacenar grafos, por ejemplo Neo4J o ArangoDb.

Por otro lado el modelo aprendido, dado que es generado mediante un proceso automático y luego utilizado para extraer información de él, una buena solución es la serialización. Es decir, después del proceso de aprendizaje serializar el modelo en un fichero y disponerlo en un sistema distribuido de ficheros, por ejemplo. Después, para usar el modelo basta con deserializarlo. Hoy en día hay potentes librerías de serialización automática para múltiples lenguajes (como Kryo, que es la recomendada por Spark).

2.2. Procesamiento de datos

Un sistema de computación distribuida proporciona escalabilidad para el alojamiento en memoria de los datos así como de los recursos computacionales per se. En esencia, relaja las limitaciones que un único dispositivo pueda tener. Un sistema distribuido típicamente se instala en un cluster formado por un conjunto de nodos conectados por una red donde éstos se envían mensajes entre ellos. Los sistemas distribuidos suelen ofrecer diversos mecanismos para incrementar y decrementar la cantidad de nodos del cluster. Dado que los sistemas de computación están diseñados para ejecutar tareas batch la adecuación de la cantidad de nodos en el cluster para una tarea es configurada antes de ejecutar dicha tarea. Este tipo de soluciones dan una respuesta a la escalabilidad de memoria y recursos de computación aunque su naturaleza distribuida implica que la calidad de la red que aloja los nodos es una restricción. El origen de los sistemas distribuidos era poder usar un cluster de máquinas pequeñas en lugar de una sola máquina grande para reducir así los costes de la infraestructura.

No obstante, la densidad de conexiones también incrementa con el escalado del cluster. La estrategia primera es adecuar la calidad de la infraestructura de cada nodo y la cantidad de nodos al problema, algo habitualmente heurístico y basado en la experiencia. Otra solución es establecer políticas de particionamiento y planificación. Por ejemplo, Hadoop reduce la cantidad de tráfico en el cluster ejecutando la tarea map para un paquete de datos en el nodo donde están los datos alojados, de modo que no es necesario mover el paquete para procesarlo. Estos sistemas permiten configurar las estrategias de particionamiento de los datos[2].

Siempre hay que considerar el equilibrio de la tríada computación, memoria, I/O. No se debe escoger un sistema distribuido por ser tendencia, si no por una decisión objetiva. Los sistemas distribuidos añaden una sobrecarga en el I/O que debe tenerse en cuenta a la hora de tomar la decisión de su uso en un entorno de producción. Debe justificarse que el volumen de datos excede la memoria, la computación es compleja y por tanto el coste de I/O de un sistema distribuido merece la pena [6] [8].

La decisión de qué sistema de procesamiento de datos distribuido escoger depende fundamentalmente de dos factores: del tipo de datos y del conocimiento del equipo que debe desarrollar el sistema. Si omitimos el factor del tipo de datos hemos de buscar sistemas de procesamiento distribuido generalistas como Hadoop o Spark. Son sistemas flexibles, pero requiere una parte importante de desarrollo para implementar la solución específica. Sin embargo, podemos hacer una pequeña guía de tecnologías según el tipo de datos que se deben procesar.

Un tipo de datos habitual es el dato relacional, es decir, que sigue el modelo de entidad-relación como el de las bases de datos relacionales. Las operaciones sobre este tipo de datos suelen ser SQL-like, es decir, que siguen las semánticas del lenguaje SQL. Tecnologías afines a estas premisas son PIG, Apache Hive, Spark SQL, Stinger o Apache Drill. Otro tipo de dato habitual es el *streaming,* es decir, el dato que fluye constantemente de una fuente y debe procesarse en tiempo real (o casi-real). Las tecnologías afines son Apache Storm, Spark Streaming o Apache Flume. Luego las búsquedas textuales, es decir, buscar patrones en textos (lo cual incluye el caso de analizar logs) se tienen las tecnologías Elasticsearch, Solr o Lucene.

3. El algoritmo

Modelos Gráficos Probabilisticos [5, 1] (*Probabilistic Graphical Models* en inglés, y PGM a partir de ahora) es un marco teórico matemático para tratar con probabilidades estructuradas. Una probabilidad se dice que es estructurada cuando hay información sobre su factorización. PGM usa esta información estructurada para reducir la complejidad de la inferencia. PGM define un conjunto de modelos de grafos que se ajustan a diferentes hipótesis sobre las probabilidades estructuradas (Bayes Networks, Markov Networks, Factor Graphs, Cluster Graphs,) y sobre éstos un conjunto de algoritmos para efectuar la inferencia. Por lo que básicamente se estructuran las probabilidades mediante

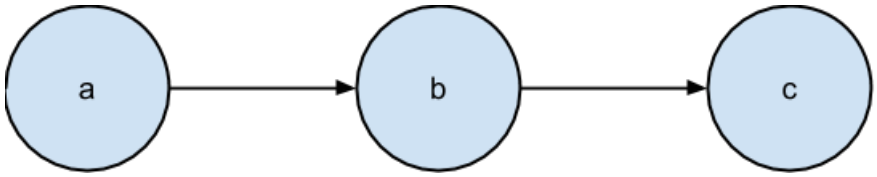

Figura 3.1: Ejemplo de modelo gráfico probabilístico

grafos cuyas propiedades se explotan para tener una ventaja en la complejidad de la inferencia respecto la inferencia clásica. Para entender el algoritmo Belief Propagation antes hay que introducirse en los modelos gráficos probabilísticos, explicando los tipos de modelos básicos como representación y las tareas habituales de inferencia y aprendizaje. En este sentido PGM es un framework que sigue el patrón de los modelos de aprendizaje máquina, donde dados unos datos se construye un modelo en un proceso llamado de aprendizaje y luego se infiere el modelo aprendido para extraer información adicional. Esta exposición del framework corresponde a la primera parte de esta sección. Luego para continuar cabe introducir el patrón de paso de mensajes que consiste en un esqueleto de algoritmo que muchos algoritmos iterativos del framework usan, y en particular Belief Propagation. En general muchos algoritmos iterativos sobre grafos usan este patrón. Después para poder introducir el algoritmo Belief Propagation hay que presentar la familia de algoritmos Sum-Product perteneciente a PGM y sus dos vertientes: los algoritmos de inferencia exacta y los algoritmos de inferencia aproximada. Finalmente Belief Propagation es estudiado como un algoritmo de inferencia aproximada la familia Sum-Product.

3.1. Modelos Gráficos Probabilisticos

Modelos gráficos probabilisíticos usa una representación basada en grafos como base para de forma compacta representar un espacio probabilístico multidimensional. En la estructura de grafo los vértices corresponden a las variables del dominio mientras que los arcos entre vértices corresponden a las interacciones probabilísticas entre éstos. En la figura 3.1 está representado un modelo en el cual intervienen tres variables a, b, c donde a condiciona probabilísticamente a b y b condiciona probabilísticamente a c. El arco $a \to b$ corresponde a la probabilidad condicionada $P(b|a)$ y el arco $b \to c$ la probabilidad condicionada $P(c|b)$. De este modo la probabilidad conjunta del modelo corresopnde a $P(a, b, c) = P(a) P(b|a) P(c|b)$.

Este es la primera ventaja de PGM, su representación es transparente gracias a la estructura de grafo. Por ejemplo, un experto en el dominio sobre el que se aplique PGM podría entender las dependencias entre variables aleatorias sólamente observando el grafo, sin ser distraído por los datos que subyacen. La segunda ventaja es la batería de algoritmos que PGM ofrece para efectuar inferencia en el modelo. La tercera ventaja es el conjunto de algoritmos para aprender el modelo a partir de un conjunto de aprendizaje. Estas tres ventajas

son las esenciales para el uso de PGM. Éste modelo tiene dos tipos principales: red bayesiana y red de Markov. La red bayesiana es la más popular (un caso particular muy conocido es la *Naive Bayes* que es usada como clasificador, por ejemplo, de *spam*) y es el primero en ser presentado en la siguiente sección.

3.1.1. Redes Bayesianas

Las redes bayesianas es un tipo de modelo del framework PGM usadas en expresión genética (análisis genético)[9], medicina[15], entre otras disciplinas. En particular, como se ha comentado anteriormente, el modelo *Naive Bayes*, que es una simplificación de una red bayesiana, es usado como método estándar de clasificación del cual existen implementaciones en múltiples librerías de programación. En una red bayesiana el grafo es dirigido acíclico, los vértices son variables aleatorias y los arcos corresponden intuitivamente a influencias directas entre variables.

Definición 1. *Una estructura de red bayesiana G es un grafo dirigido acíclico donde los vértices representan variables aleatorias $X_1, ..., X_n$. Sea $\mathrm{Pa}_{X_i}^G$ la expresión de los padres de X_i en G, y $\mathrm{NonDescendants}_{X_i}$ la expresión de los vértices no descendientes de X_i en G. Entonces G contiene el siguiente conjunto de asumpciones de independencia condicional, llamadas independencias locales, y denotadas por $I_l(G)$:*
Por cada variable X_i : $\left(X_i \perp \mathrm{NonDescendants}_{X_i} | \mathrm{Pa}_{X_i}^G\right)$

El concepto de $I_l(G)$ se generaliza mediante el concepto *I-map* que flexibiliza la compatibilidad entre distribuciones de probabilidad y estructuras de red bayesiana. No obstante, no es imprescindible para la definición de red bayesiana. Pero antes de definir la BN primero hay que ver la compatibilidad entre la estructura y una distribución:

Definición 2. *Sea G una estructura de BN sobre las variables $X_1, ..., X_n$ y P una distribución de probabilidad sobre el mismo espacio que define G, entonces se dice que P factoriza en G si P puede ser expresada como el siguiente producto:*
$P(X_1, ..., X_n) = \prod_{i=1}^{n} P\left(X_i | \mathrm{Pa}_{X_i}^G\right)$

Con esto se puede definir el concepto de red bayesiana:

Definición 3. *Una red bayesiana es el par $B = (G, P)$ donde G es una estructura de red bayesiana y P es una distribución de probabilidad tal que factoriza en G y es especificada mediante las distribuciones de probabilidad condicionadas asociadas con los vértices de G.*

Un ejemplo de red bayesiana es precisamente el expuesto en la figura 3.1 en la página anterior. No obstante, las redes bayesianas tienen limitaciones pues hay conjuntos de asumpciones de independencia que no se pueden modelar perfectamente mediante una red bayesiana. Por ejemplo, el conjunto de independencias $\{(A \perp C | \{B, D\}), (B \perp D | \{A, C\})\}$, no puede ser modelado sin añadir otras asumpciones de independencia. En estos casos las redes de Markov ofrecen una solución más flexible, que es lo que se expone a continuación.

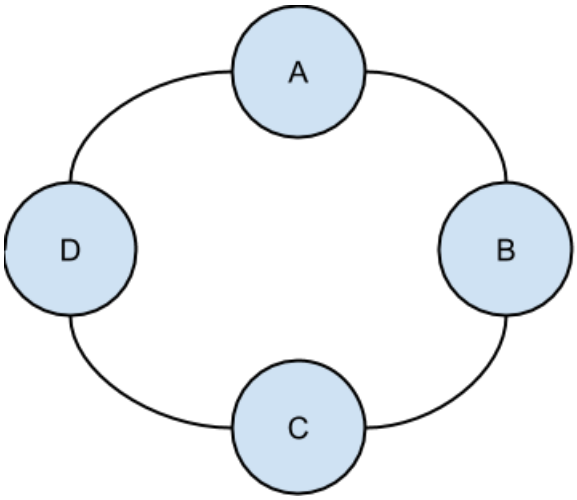

Figura 3.2: Ejemplo de red de markov para las independencias $\{(A \perp C | \{B, D\}), (B \perp D | \{A, C\})\}$

A	B	$\phi(A, B)$
0	0	100
0	1	1
1	0	1
1	1	100

Cuadro 3: Ejemplo de factor

3.1.2. Redes de Markov

Así como las redes bayesianas ofrecen una representación clara de las dependencias entre las variables a cambio de ciertas limitaciones, las redes de Markov (*Markov Networks* en inglés, y MN a partir de ahora) pueden representar cualquier distribución. No obstante, su representación es más compleja y requiere de conceptos teóricos adicionales que se explican a continuación. Las redes de Markov se basan en un grafo no dirigido, donde los arcos en este caso representan interacciones entre variables donde ninguna otra interviene.

Para introducir el concepto de red de Markov, primero hay que explicar el concepto de *factor*.

Definición 4. *Sea D un conjunto de variables aleatorias, un factor ϕ es una función de $Val(D)$ a $\mathbb{R}$. Un factor es no negativo cuando todas sus entradas son no negativas. El conjunto de variables D se llama el scope de ϕ.*

En la tabla 3 se representa un factor ϕ sobre dos variables binarias A, B. El factor muestra un comportamiento tal que cuando el valor de ambas variables es o bien 0, o bien 1 el factor tiene un valor alto, y cuando el valor

de ambas variables no es igual el factor tiene un valor bajo. Además, el factor del ejemplo es un factor no negativo. El scope de ϕ es $\{A, B\}$ y el dominio es la combinación de los valores de las variables del scope, es decir, $\{A = 0, B = 0\}$,$\{A = 0, B = 1\}$,$\{A = 1, B = 0\}$,$\{A = 1, B = 1\}$. El factor tiene un comportamiento de *fuzzy xnor* en particular.

Las redes de Markov se sirven de los factores no negativos para modelar las interacciones entre las variables, es decir, los arcos. Los factores se pueden entender como la afinidad entre los estados de las variables que conectan. Un factor se dice que es normalizado cuando la suma de todos sus valores es la unidad. Esto es $\sum_{v \in Val(\phi)} \phi(v) = 1$. En general de un factor se puede extraer la función de partición (*partition function* en inglés) expresada como Z_ϕ o simplemente Z si no hay ambigüedad que es la suma de los valores del factor. Esto es $Z_\phi = \sum_{v \in Val(\phi)} \phi(v)$. Cuando la función de partición es unitaria entonces se dice que el factor está normalizado. Se puede normalizar un factor en general dividiéndolo por la función de partición $\frac{\phi}{Z_\phi}$. En el ejemplo anterior el valor de la función de partición es $Z = 100 + 1 + 1 + 100 = 202$, y por tanto el factor no está normalizado. El factor normalizado tendría sus valores divididos por 202. Al normalizar un factor éste se puede considerar una distribución de probabilidad entre los valores que puede tomar.

El scope de un factor puede ser un conjunto arbitrario de variables del dominio, con factores sobre pares de variables como en la figura 3.2 en la página anterior no basta. Un contraejemplo es un grafo completamente conexo de n vértices dónde si nos restringiésemos a factores sobre pares de variables, cada factor tendría dos parámetros y el espacio entero de parámetros tendría cardinalidad $4\binom{n}{2}$ mientras que una distribución arbitraria sobre n variables binarias consta de $2^n - 1$ parámetros. De modo que el grafo con factores sobre pares de variables no podría capturar los parámetros de la distribución. En general un factor debido a que es una función positiva puede modelar tanto distribuciones de probabilidad como distribuciones de probabilidad condicionada, por lo que una distribución de probabilidad además de poderse representar como un único factor, su factorización se puede representar como un producto de factores normalizado. No obstante, debe primero definirse el producto de factores.

Definición 5. *Sean X, Y, Z tres conjuntos disjuntos de variables y sean $\phi_1(X, Y)$, $\phi(Y, Z)$ dos factores. Se define el producto de factores $\phi_1 \times \phi_2$ como un factor $\psi : Val(X, Y, Z) \mapsto \mathbb{R}$ tal que*

$\psi(X, Y, Z) = \phi_1(X, Y)\, \mathring{u} \phi_2(Y, Z)$

El punto clave de este producto es el solapamiento de scopes y que al multiplicarse los valores la asignación de los valores de las variables en Y conincide para el par de factores multiplicados.

Siguiendo el ejemplo de factor anterior, podemos multiplicarlo por un factor $\psi(A)$ con valores $\psi(A = 0) = 0{,}5, \psi(A = 1) = 0{,}75$ dando como resultado el factor de la tabla 4 en la página siguiente. En éste se puede observar que las asignaciones de la variable A son compartidas entre los factores que se multiplican.

A	B	$\psi(A) \times \phi(A,B)$
0	0	$\psi(A=0)$ ů$\phi(A=0,B=0) = 50$
0	1	$\psi(A=0)$ ů$\phi(A=0,B=1) = 0{,}5$
1	0	$\psi(A=1)$ ů$\phi(A=1,B=0) = 0{,}75$
1	1	$\psi(A=1)$ ů$\phi(A=1,B=1) = 75$

Cuadro 4: Ejemplo de producto de factores entre los factores ϕ y ψ

Con el producto de factores se puede definir una parametrización no dirigida de una distribución:

Definición 6. *Una distribución P_Φ es una distribución Gibbs parametrizada por un conjunto de factores $\Phi = \{\phi_1(D_1), ..., \phi_k(D_k)\}$ si es definida como sigue:*
$P_\Phi(X_1, ..., X_n) = \frac{\widetilde{P}_\Phi(X_1,...,X_n)}{Z}$
Donde
$\widetilde{P}_\Phi(X_1, ..., X_n) = \phi_1(D_1) \times \cdots \times \phi_k(D_k)$
Y Z es la función de partición de $\widetilde{P}_\Phi$.

Con esto se puede definir las redes de Markov:

Definición 7. *Sea P_Φ una distribución formada por el producto normalizado de los factores $\{\phi_1(D_1), ..., \phi_n(D_n)\}$, entonces se dice que P_Φ factoriza sobre una red de Markov $\mathcal{H}$ si por cada D_i $(i = 1, ..., n)$ es un subgrafo completo de $\mathcal{H}$.*

Los factores en una red de Markov se suelen llamar *potential clique y están contenidos* en los subgrafos del grafo por lo que dado un grafo no dirigido no se puede inferir cómo están distribuidos los factores con certeza. Una estructura explícita en este sentido es el *factor graph*:

Definición 8. *Un factor graph $\mathcal{F}$ es un grafo no dirigido que contiene dos tipos de vértices: los vértices variable, y los vértices factor. El grafo solamente contiene arcos entre tipos diferentes de vértice, no entre vértices del mismo tipo. Un factor graph $\mathcal{F}$ está parametrizado por un conjunto de factores donde cada vértice factor está asociado con un factor el scope del cual es el conjunto de vértices de tipo variable que tiene por vecinos. Una distribución P factoriza sobre $\mathcal{F}$ si puede ser representada mediante los factores del grafo.*

Los *factor graph* también se pueden ver como hipergrafos donde los vértices son las variables y los hiperarcos los factores. Los vértices asociados conectados por un hiperarco son precisamente las variables que el factor tiene por scope. Esta analogía es natural, pues los grafos bipartitos, que es el caso del *factor graph*, son una forma natural de modelar los hipergrafos. La figura 3.3 en la página siguiente representa un *factor graph* con las mismas variables y relaciones que la red de Markov de la figura 3.2 en la página 13.

Toda red bayesiana puede ser representada mediante una red de Markov. Sin embargo, una red de Markov debe ser un *chordal graph* para poder tener una equivalencia como red bayesiana.

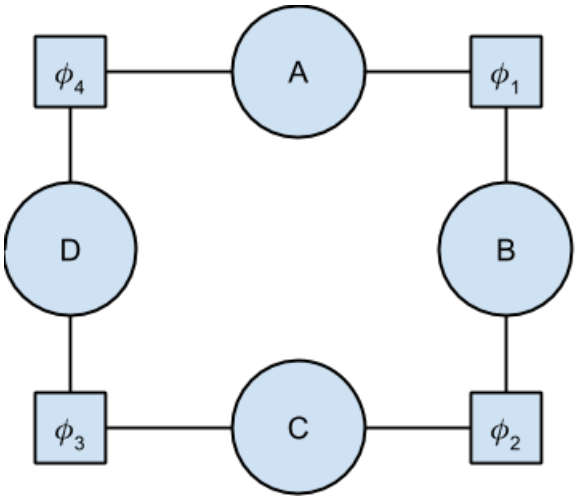

Figura 3.3: Ejemplo de *factor graph*, donde los nodos con forma de círculo son las variables y los cuadrados los factores

3.1.3. Inferencia

Los modelos PGM como modelos probabilísticos pueden responder a preguntas probabilísticas sobre la distribución subyacente. Por ejemplo, la probabilidad marginal de ciertas variables, o la probabilidad a posteriori de algunas variables dada la evidencia de otras. Sobre la figura de ejemplo se pueden responder preguntas como la probabilidad marginal de b, es decir, $P(b)$; o la probabilidad de a dado la observación $c = x$, es decir, $P(a|c = x)$. La inferencia es uno de los procesos que por ejemplo subyace en los métodos de aprendizaje máquina, como la toma de decisiones automática. Por ejemplo un caso más complejo, dada cierto modelo donde una acción humana y el contexto condiciona un evento del cual podemos cuantificar su utilidad nos podríamos preguntar conociendo cierta información del contexto cuál es la acción que maximiza la utilidad final de la decisión.

La inferencia comienza por calcular marginales dadas unas evidencias, es decir, $P(Y|E = e) = \frac{P(Y,E=e)}{P(E=e)}$. Tanto en el numerador como en el denominador se requiere la marginalización sobre un subconjunto de las variables. Es decir, la marginalización de una distribución de probabilidad en un subconjunto de sus variables es la operación básica de inferencia. La marginalización ingenua sobre la distribución de probabilidad, es decir, variable por variable sumar sus entradas en la probabilidad conjunta es *NP-hard* [5]. No obstante, PGM ofrece métodos que pueden mejorar el rendimiento del método ingenuo. Las sección section 3.3 expone algunos métodos para efectuar esta marginalización en PGM. Los métodos de marginalización en PGM se clasifican en tres grupos: inferencia exacta, optimización y basados en partículas.

Los métodos de inferencia exacta como el algoritmo Eliminación de Variable,

comentado en la sección 3.3 en la página 20, marginalizan las variables y mediante ciertas heurísticas pueden reducir la complejidad de la marginaliazación. Los métodos de optimización bajo ciertas hipótesis son exactos, pero en general son aproximados el algoritmo Belief Propagation discutido en la sección 3.4 en la página 22 pertenece a esta família. Luego los métodos basados en partículas, como el método Markov Chain Monte Carlo o el algoritmo Metropolis-Hastings, donde se utilizan diferentes estrategias de *sampling* para aproximar los marginales.

Otro tipo de preguntas son las *maximum a posteriori*. Éstas, dadas una serie de evidencias $E = e$ buscan la asignación de variables Y tal que maximiza la probabilidad. Es decir, $\text{argmax}_Y P(Y|E = e)$. La utilidad de este tipo de preguntas es evidente, dado un paciente con los síntomas observados $E = e$ cabe preguntarse cuál es el conjunto de patologías Y que hace más probable tener estos síntomas observados. El método ingenuo para solucionar este problema también es *NP-hard* [5], como la marginalización. Los métodos de marginalización pueden ser adaptados para computar MAP, además de otros métodos.

3.1.4. Aprendizaje

Hasta ahora el modelo venia dado, es decir, forma parte de los datos de entrada de los algoritmos y métodos. También puede tener interés obtener un modelo para poder descubrir conocimiento de éste, por ejemplo, las dependencias entre las variables aleatorias. En cualquier caso, el modelo ha de ser extraído de alguna fuente. Hay dos fuentes de información para la generación de modelos: los datos de las observaciones pasadas y los expertos del dominio. También hay dos componentes en un modelo PGM que deben ser aprendidos: la estructura de la red, es decir, el conjunto de dependencias; y los parámetros de las probabilidades condicionales locales o de los factores. Los expertos del dominio pueden dar indicaciones sobre la estructura del modelo aunque es necesario traducir ese conocimiento al lenguaje que el modelo tiene. Luego los parámetros difícilmente pueden ser extraídos de expertos del dominio, siempre es más fiable extraer los parámetros de las propias observaciones. Por otro lado, los expertos suelen ser caros o su conocimiento ser insuficiente en algún aspecto. De modo que PGM ofrece tanto métodos para aprender la estructura del modelo mediante las observaciones así como compaginar el conocimiento previo experto con el aprendizaje automático del modelo. Además PGM ofrece métodos de aprendizaje de parámetros.

La estimación de parámetros se efectúa mediante los métodos de máximia verosimilitud (MLE, *Maximum Likelihood Estimation)* y de estimacion bayesiana también conocida como *máximum a posteriori estimation* [5]. El aprendizaje de la estructura consiste en declarar un conjunto de estructuras candidatas y escoger la que mejor se adapta a las observaciones. A la hora de declarar el conjunto de estructuras candidatas se pueden añadir las restricciones estructurales que los expertos puedan aportar. No sólamente se puede determinar qué estructura de dependencias es la más acertada para los datos, sino que también se debe considerar el escenario de las variables ocultas, es decir, aquellas

que no son observables pero forman parte del modelo (semejante a las capas intermedias de las redes neuronales). Esta consideración también se tiene en cuenta en los métodos de aprendizaje de PGM. Con el conjunto de estructuras candidatas declarado, se aprenden los parámetros de cada una de éstas con aproximadamente un 70 por ciento de los datos (*training set*), luego se puntúa cada estructura con un 20 % de datos, finalmente la estructura mejor puntuada puede ser evaluada con el 10 % restante de datos (*training set*). La búsqueda exahustiva de la estructura con mejor puntuación tiene un coste superexponencial [5], por lo que son necesarios métodos adicionales para poder encontrar la estructura que optimiza la puntuación teniendo en cuenta la complejidad del espacio y los máximos locales. Al igual que al estimar los parámetros se puede llegar a hacer una sobreestimación (*overfitting*), también se puede sobreestimar una estructura. Por ejemplo, una estructura de grafo completo siempre se ajustará a los datos pues contiene todas las dependencias posibles. Por ende cuando se puntúan las estructuras también se debe penalizar su complejidad. Además del método de puntuación, hay otros métodos incluso específicos para los dos tipos de modelos.

3.2. Modelo de paso de mensajes

Como se ha visto en el apartado anterior sobre PGM el framework se sirve de los grafos para estructurar las dependencias del modelo probabilístico y de este modo reducir el coste computacional a la hora de trabajar con modelos probabilísticos complejos. No obstante, el grafo que soporta la estructura así como los datos asociados a los nodos (los factores) pueden aún así representar un reto en cuanto a memoria. A la hora de afrontar la computación de un algoritmo sobre un grafo grande se tienen las siguientes aproximaciones[7]:

1. Crear un sistema distribuido personalizado que implica un coste de desarrollo grande y teniendo que modificar el sistema para cada algoritmo o grafo nuevo que en el futuro se pueda presentar.

2. Usar una infraestructura distribuida existente, habitualmente no diseñada para procesar grafos, como *Map Reduce*.

3. Usar una libraría de computación de grafos en una máquina renunciando a la infraestructura distribuida y asumiendo las limitaciones de escalabilidad.

4. Usar un sistema de computación paralela en grafos existente, como *Apache Giraph* o *Apache Spark GraphX*.

En 2010 los sistemas de computación paralela en grafos existentes eran descartados por no ser resilientes al fallo de las máquinas, por ello Google propuso como solución un sistema de computación paralela en grafos resiliente llamado *Pregel [7]*, no obstante es código cerrado propiedad de la multinacional. Sin embargo, la comunidad *open source* (en particular la fundación Apache Fundation)

aprovechó esta oportunidad para desarrollar sus propias implementaciones inspirándose en la filosofía de *Pregel* reusando frameworks existentes (el caso de *Apache Giraph*, que funciona sobre *Hadoop*) o introduciéndola en nuevas herramientas emergentes (el caso de *Apache Spark GraphX*). Esta simbiosis entre la indústria digital privada y la comunidad *open source* recuerda al nacimiento de *Apache Hadoop* que también fue inspirado por un documento de investigación de Google [2].

En el modelo computacional definido en *Pregel*, conocido como modelo de paso de mensajes, los vértices son los agentes activos de la computación. El modelo se dota de un grafo dirigido y una función como entrada de datos. La función es aplicada sobre cada vértice del grafo de forma síncrona e iterativa, es decir, que se ejecuta la función para cada vértice y una vez las ejecuciones han finalizado éstas se vuelven a ejecutar. Cada iteración se llama *superstep*. Esta función centrada en el vértice puede efectuar las siguientes operaciones:

- Modificar los atributos del vértice (excepto su identificador),
- Modificar los atributos de los arcos salientes,
- Envíar mensajes a otros vértices (habitualmente a los nodos adyacentes),
- Procesar los mensajes que han sido enviados a este vértice en el *superstep* anterior,
- Modificar la topología del grafo (habitualmente eliminándose a sí mismo),
- Etiquetarse como vértice inactivo

Cuando un vértice recibe un mensaje automáticamente se etiqueta como activo y en el siguiente *superstep* se ejecutará la función para dicho vértice. Cuando todos los vértices están marcados como inactivos el proceso ha finalizado.

Este planteamiento centrado en el vértice (informalmente resumido mediante la frase "*think like a vertex*"[4]) y basado en el paso de mensajes simplifica drásticamente los algoritmos iterativos sobre grafos. Un ejemplo es la implementación de *page rank* como algoritmo de paso de mensajes. *Page rank* es un algoritmo utilizado para determinar la distribución de probabilidad de la posición de un agente que se desplaza aleatoriamente por un grafo dirigido (llamado *random surfer*) [10]. En particular, es usado para determinar la relevancia de páginas web, donde los vértices son las páginas y los arcos los enlaces entre éstas.

```
1 class PageRankVertex : public Vertex<double, void, double> {
2   public: virtual void Compute(MessageIterator* msgs) {
3   if (superstep() >= 1) {
4     double sum = 0;
5     for (; !msgs->Done(); msgs->Next())
6       sum += msgs->Value();
7       *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
8     }
9     if (superstep() < 30) {
```

```
10        const int64 n = GetOutEdgeIterator().size();
11        SendMessageToAllNeighbors(GetValue() / n);
12      } else {
13        VoteToHalt();
14      }
15    }
16  };
```

En el código observamos que cada *superstep* se agregan los mensajes recibidos en un vértice en el bucle de la linea 5. Luego mientras el *superstep* no sea el número 30 o superior cada vértice envía mensajes a todos sus vecinos con la puntuación de éste vértice en la linea 11. Finalmente cuando el *superstep* es el número 30 o superior los vértices votan por detenerse.

Este modelo de computación está inspirado en el modelo *Bulk Synchronous Parallel* desarrollado por Leslie Valiant en *A bridging model for parallel computation*[16] como en el propio documento de Google reconoce[7]. Del modelo BSP existen implementaciones como *Apache Hama*, *BSPlib, BSPonMPI, Paderborn University BSP (PUB),* entre otros. En particular *BSPonMPI* merece una nota aparte, pues implementa la especificación *Message Passing Interface* (MPI) mediante el modelo BSP.

3.3. Sum-product

Durante la inferencia en modelos PGM, la tarea básica es la marginalización de variables tal y como se comenta en la sección 3.1.3 en la página 16. Sea Φ un conjunto de factores sobre un conjunto de variables X y sea Y un subconjunto de X, entonces la marginalización de las variables Y para Φ consiste en calcular la siguiente expresión:

$$\sum_{Y} \prod_{\phi \in \Phi} \phi$$

El factor resultante tiene por scope el conjunto de variables $X \setminus Y$. El cálculo ingenuo implica el producto de los factores y, por ende, el cálculo de la distribución explícita. En cualquier caso, el cálculo de la distribución explícita es NP. Los algoritmos Sum-Product se soportan en el conjunto de factores que determinan la distribución explícita para marginalizar las variables pero sin calcular necesariamente la distribución explícita. No obstante, para calcular el resutlado exacto en el peor de los casos siempre se calculará la distribución explícita y el uso de estos algoritmos no supone ninguna ventaja. El algoritmo principal en esta sección es el llamado *variable elimination*. Este algoritmo, elimina las variables del conjunto Y una por una como en el ejemplo siguiente:

$$X = \{a, b, c, d\}$$

$\Phi = \{\phi_{a,b}, \phi_a, \phi_{a,c}, \phi_c, \phi_{c,d}\}$, donde los subíndices de los factores indican su scope

$$Y = \{c, d\}$$

$$
\begin{aligned}
P(a,b) &= \sum_c \sum_d (\phi_{a,b} \mathbin{\mathring{u}} \phi_a \mathbin{\mathring{u}} \phi_{a,c} \mathbin{\mathring{u}} \phi_c \mathbin{\mathring{u}} \phi_{c,d}) \\
&= \sum_c \left(\phi_{a,b} \mathbin{\mathring{u}} \phi_a \mathbin{\mathring{u}} \phi_{a,c} \mathbin{\mathring{u}} \phi_c \sum_d \phi_{c,d} \right) \\
&= \phi_{a,b} \mathbin{\mathring{u}} \phi_a \sum_c \left(\phi_{a,c} \mathbin{\mathring{u}} \phi_c \sum_d \phi_{c,d} \right)
\end{aligned}
$$

En este ejemplo observamos que no es necesario calcular la distribución explícita, sino factores intermedios cuyo scope es menor que el scope de la distribución. No obstante, en el peor de los casos, alguno de estos factores intermedios puede tener un scope igual al de la distribución. De igual modo, el tamaño de los factores intermedios puede ser impracticable, por lo que la ordenación de la eliminación de las variables es crítico para minimizar la complejidad de la eliminación. No obstante, encontrar el orden óptimo es NP, de modo que el algoritmo se completa con heurísticas para determinar el orden de eliminación de las variables. A continuación el algoritmo *variable elimination*[5].

Algorithm 1 Sum-product variable elimination algorithm

Procedure Sum-Product-VE($\Phi, Z, \prec$)
Let $Z_1, ..., Z_k$ be an ordering of Z such that $Z_i \prec Z_j$ if and only if $i < j$

1: **for** $i = 1, ..., k$ **do**
2: $\quad \Phi' \leftarrow \{\phi \in \Phi : Z_i \in Scope[\phi]\}$
3: $\quad \Phi'' \leftarrow \Phi - \Phi'$
4: $\quad \psi \leftarrow \prod_{\phi \in \Phi'} \phi$
5: $\quad \tau \leftarrow \sum_{Z_i} \psi$
6: $\quad \Phi \leftarrow \Phi'' \cup \{\tau\}$
7: **end for**
8: **return** Φ

El algoritmo itera sobre las variables a eliminar según la ordenación dada en la linea 1. Por cada variable a eliminar se seleccionan los factores que contienen en su scope dicha variable (linea 2), se multiplican y se marginaliza la variable (linea 4). Luego se reasigna Φ como el conjunto de factores excepto el los que contenían la variable junto con el nuevo factor generado (linea 4). Este nuevo conjunto ya no contiene en ninguno de los scopes de los factores la variable eliminada, dado que los que la contenían han sido multiplicados y la variable marginalizada. Al terminar el proceso, el conjunto de factores no contiene ninguna de las variables a eliminar.

Para analizar la complejidad de este proceso hemos de fijarnos en los dos tipos de factores que se manipulan: los que pertenecen originalmente a Φ y los que se generan durante la eliminación de una variable para marginalizar, llamados factores intermedios ($\prod_{\phi \in \Phi'} \phi$). Supongamos que hay $n$variables y, por simplicidad sin falta de generalidad, que queremos eliminar todas las variables.

Observamos que cada factor es multiplicado exactamente una vez, ya que si se multiplica también se extraen del conjunto (linea 2 y 4). El coste de generar el factor intermedio de la iteración i, que llamaremos ψ_i, multiplicando los seleccionados es $O(N_i)$, donde N_i es la cantidad de valores que ψ_i tiene. La marginalización de los factores intermedios en la línea 4 lleva también un coste $O(N_i)$. En definitiva, con $N_{max} = \text{máx}_i\, N_i$, el coste de eliminar las variables es $O(kN_{max})$. Esto hace que el algoritmo sea sensible al tamaño de los factores intermedios. De cualquier modo la cantidad de valores de un factor es exponencial respecto la cantidad de variables de su *scope*. Por ende, el coste del algoritmo es dominado por el tamaño de los factores intermedios con un crecimiento exponencial respecto la cantidad de variables en los factores.

Debido a esta sensibilidad del algoritmo de eliminación de variables al tamaño de los factores intermedios la ordenación de las variables a eliminar es clave. No obstante, determinar la ordenación tal que minimiza la complejidad del algoritmo es *NP-completo* [5]. De este modo, se toman heurísticas para determinar la ordenación de las variables basándose en la topología del grafo con el objetivo de minimizar el coste de la eliminación de cada variable.

Aunque el algoritmo eliminación de variable para marginalizar variables es un método más flexible y menos costoso que la marginalización directa sobre la distribución conjunta aún así sigue teniendo un coste exponencial. Esto lleva a desarrollar otros algoritmos para mitigar el coste computacional. El algoritmo general derivado del algoritmo eliminación de variable es el algoritmo Belief Propagation que es explicado en detalle en la sección siguiente.

3.4. Algoritmo Belief Propagation

Hemos visto cómo el algoritmo eliminación de variable computa los marginales y es más ventajoso que la computación ingenua sobre la distribución conjunta. Pero aún así tiene un coste exponencial respecto la cantidad de variables. Veamos ahora otra aproximación basada en la iteración de paso de mensajes. Para comenzar hay que definir una nuevo modelo PGM, el *cluster graph.*

Definición 9. *Un cluster graph U para un conjunto de factores Φ con variables X es un grafo no dirigido, donde cada vértice i está asociado con un subconjunto de variables $C_i \subseteq X$. Un cluster graph debe cumplir la preservación de familia que consiste en que cada factor $\phi \in \Phi$ debe estar asociado con un cluster C_i, denotado por $\alpha(\phi)$, tal que scope $[\phi] \subseteq C_i$. Cada arco entre un par de clusters C_i, C_j está asociado con las variables $S_{i,j} \subseteq C_i \cap C_j$, llamado sepset (de separation set).*

Este objeto no es un constructo artificial, sino que surge de forma natural durante la ejecución del algoritmo eliminación de variable. Durante la ejecución de éste se genera por cada variable eliminada un factor intermedio ψ_i que se asocia con el cluster $C_i = scope\,[\psi_i]$. Luego, dos clústers C_i, C_j están asociados si para la el cálculo de ψ_j es necesario el factor τ_i . El *cluster graph* es un grafo no dirigido, no obstante, el proceso de eliminación de variables determina una direccionalidad en los arcos. Aunque el *cluster graph* es un modelo general, el

cluster graph inducido por la ejecución del algoritmo eliminación de variable tiene nombre propio debido a sus propiedades.

Proposición 10. *El cluster graph inducido de la ejecución del algoritmo eliminación de variable es un árbol.*

Definición 11. *Dado un cluster tree que se dice que cumple la propiedad running intersection si para toda variable X tal que $X \in C_i$ y $X \in C_j$ todos los clústers en el camino que conectan C_i con C_j contienen X.*

Proposición 12. *El clúster graph inducido de la ejecución del algoritmo eliminación de variable cumple la propiedad running intersection.*

Estas dos propiedades dan al *cluster graph* inducido de la ejecución del algoritmo eliminación de variable nombre propio:

Definición 13. *Un cluster tree que cumple la propiedad running intersection se dice que es un clique tree (también llamado junction tree o join tree). En este caso los clústers también son llamados cliques.*

Con esto podemos replantear el algoritmo eliminación de variable como un algoritmo de paso de mensaje sobre el *clique tree* generado. El algoritmo es llamado *sum-product belief propagation.* Asociemos en cada clúster C_i el factor $\psi_i = \prod_{\phi:\alpha(\phi)=i} \phi$, que tiene el nombre de potencial. Dos clústers C_i, C_j tienen la capacidad de enviarse mensajes del siguiente modo:

$$\delta_{i\to j} = \sum_{C_i \setminus S_{i,j}} \psi_i \prod_{k\in(Neighbors(i)\setminus\{j\})} \delta_{k\to i}$$

Es decir, un mensaje $\delta_{i\to j}$ es la marginalización sobre el *sepset* $S_{i,j}$ del producto del potencial ψ_i junto con el producto de todos los mensajes que el clúster C_i recibe del resto de sus vecinos. Un clique puede enviar mensajes cuando ha recibido los mensajes llegantes según la direccionalidad que induce el proceso de eliminación de variable. Este procedimiento de paso de mensajes siguiendo la ruta que la ejecución del algoritmo eliminación de variable determina resulta en un resultado igual al del algoritmo eliminación de variable. No obstante, el interés del algoritmo es su generalidad pues puede ser generalizado a un *cluster graph* arbitrario.

El algoritmo Belief Propagation general, inicializa los mensajes como factores uniformes, luego envía actualiza los mensajes de forma secuencial, uno tras otro, hasta que el *cluster graph* está calibrado, es decir, los mensajes se estabilizan.

El coste de este algoritmo es por cada iteración el cómputo del mensaje, que consiste en el producto de los recibidos junto con el potencial y la marginalización sobre el sepset $S_{i,j}$. Cabe decir que el producto de los mensajes llegantes no tiene porqué ser recomputado sistemáticamente, debe ser calculado solamente si alguno de sus mensajes entrantes ha sido actualizado. Por lo que el coste está controlado por el tamaño de los sepsets [5]. Dado que el *cluster graph* puede ser modelado y se pueden tomar decisiones en su construcción se puede ajustar la estructura para que los sepsets sean sobre pocas variables. El caso particular

Algorithm 2 Sum-Product Belief Propagation

Procedure BeliefPropagation(G)

Initialize(G)
while G is not calibrated **do**
  Select (i, j) in G
  $\delta_{i \to j}(S_{i,j}) \leftarrow$SP-Message$(i, j)$
end while
for each clique i **do**
  $\beta_i \leftarrow \psi_i \mathring{\mathrm{u}} \prod_{k \in Neighbors(i)} \delta_{k \to i}$
end for
$\{\beta_i\}$

Procedure Initialize(G)

for each cluster C_i **do**
  $\beta_i \leftarrow \prod_{\phi : \alpha(\phi) = i} \phi$
end for
for each edge (i, j) in G **do**
  $\delta_{i \to j} \leftarrow 1$
  $\delta_{j \to i} \leftarrow 1$
end for

Procedure SP-Message(i, j)

$\psi(C_i) \leftarrow \psi_i \mathring{\mathrm{u}} \prod_{k \in (Neighbors(i) \setminus \{j\})} \delta_{k \to i}$
$\tau(S_{i,j}) \leftarrow \sum_{C_i \setminus S_{i,j}} \psi(C_i)$
return $\tau(S_{i,j})$

de construcción de Bethe Cluster Graph donde los sepsets son una sola variable por construcción [5]. La convergencia de este algoritmo en el caso de *clique trees* está garantizada [5]. En general, el algoritmo no tiene porqué converger, no obstante, estructurando adecuadamente el *cluster graph*, usando alguna variación de Belief Propagation más adecuada el algoritmo suele converger [5].

4. Apache Spark

Apache Spark es un motor generalista de procesamiento de datos a gran escala. Tal y como ellos se bautizan en la página web (spark.apache.org) "*Lightning-fast cluster computing*". Durante años el uso del patrón *map reduce* ha dominado el procesamiento de datos masivo dado que se sus etapas (*map, shuffle, reduce*) se han adaptado a las necesidades industriales que generalmente se resumen en las siglas *ETL* (*Extract, Transform, Load*). No obstante, el desarrollo en Apache Hadoop, que es el framework de referencia que implementa *map reduce*, es costoso, pues habitualmente implica la escritura de gran cantidad de código, un testeo complicado debido a la naturaleza distribuida del entorno y un *profiling* complejo. Por estos motivos también han aparecido sistemas que extienden Hadoop con la intención de ofrecer a la industria mejores soluciones. Ejemplos son Ambari, Hive, HBase, Pig, Mahout, Oozie, ... No obstante, la tecnología avanza y el patrón *map reduce* comenzó a ser adoptado en otros lenguajes y, sobretodo, en las bases de datos NoSql como característica propia haciendo posible en muchos casos la reducción del código necesario para implementar una transformación y con las ventajas de estar integrado en la plataforma de gestión de datos, por lo que los costes de mantenimiento también se reducían. No obstante, estos sistemas NoSql tenían limitaciones, como su bajo performance o que debido a compartir proceso con los datos el performance de las operaciones sobre estos al márgen de las transformaciones *map reduce* se veía reducido. Por lo que mediante Hadoop se podía alcanzar una gran eficiencia con un coste de desarrollo y mantenimiento elevado o bien mediante las bases de datos NoSql se podía lograr un prototipado rápido pero con escasa escalabilidad. Al paso surgió Apache Spark con la promesa de poder prototipar rápidamente aplicaciones gracias a su amplia API y módulos manteniendo (incluso mejorando) la eficiencia de Hadoop.

Esta promesa se ha realizado cuando el 5 de noviembre de 2014 Apache Spark se ha alzado con un récord en Daytona GraySort contest[3]. Citando un post de la empresa Databricks:

> "*Winning this benchmark as a general, fault-tolerant system marks an important milestone for the Spark project. It demonstrates that Spark is fulfilling its promise to serve as a faster and more scalable engine for data processing of all sizes, from GBs to TBs to PBs. In addition, it validates the work that we and others have been contributing to Spark over the past few years.*"[17]

Databricks ha tenido una colaboración con Apache Spark muy parecida a la

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 min	23 min	234 min
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

Cuadro 5: Comparación de los records de Hadoop y Spark en Daytona GraySort contest. [17]

que Datastax tiene con Apache Cassandra. Ambas empresas ofrecen servicios de soporte, mantenimiento y certificados sobre los respectivos proyectos siendo uno de los puntos clave de su modelo de negocio y por otro lado dichas empresas aportan código y difusión sobre el software, que en cualquier caso es *open source* por ser desarrollado bajo la fundación Apache Fundation. Con la eficiencia sobre Hadoop probada como se puede obsevar en la tabla 1, la prueba que a Spark le queda pasar es la de verificar su adopción en el mercado y la comunidad mediante su usabilidad en sistemas industriales.

No obstante, el récord de Spark podría haber sido mucho más espectacular. El objetivo de Daytona GraySort contest es poner a prueba la gestión del I/O de los sistemas por lo que el sistema de caché en memoria fue desativado tal y como se cita en el artículo anteriormente referido "*All the sorting took place on disk (HDFS), without using Sparks in-memory cache*". Spark ofrece un sistema de caché en memoria gestinado mediante los objetos RDD (*Resilient Distributed Dataset*) que son el concepto principal del sistema.

4.1. RDD

RDD es una abstracción desarrollada por la Universidad de California (Berkeley) en el artículo *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*[18]. Un RDD es una colección de items particionados y de sólo-lectura con las siguientes propiedades:

- pueden ser creados mediante operaciones deterministas sobre datos materiales, es decir, existentes en algún sistema (sistema de ficheros, base de datos, ...);
- o mediante operaciones deterministas sobre otros RDDs, llamadas transformaciones;
- no tienen que ser materializados necesariamente en toda ocasión pues se mantiene la información suficiente sobre su ascendencia para ser derivados de los datos materiales originales; y
- los niveles de persistencia y la estrategia de partición pueden ser ajustados por los usuarios.

El tercer punto es clave para la gestión de fallos de nodos, pues si un nodo que mantiene una partición de un RDD falla su partición puede ser recuperada de los datos originales por otro nodo, sin necesariamente mantener una replicación material del RDD. La gestión de la estrategia de partición es algo primordial para adecuar los sistemas distribuidos a las necesidades de las aplicaciones y no resulta una novedad. No obstante, la gestión de los niveles de persistencia sí es novedoso. Spark ofrece tres niveles de persistencia:

- en memoria como objetos Java deserializados,
- en memoria como objetos Java serializados (cuyo mecanismo de serialización es personalizable también), y

- en disco.

La primera opción es la más eficiente en cuanto a uso de CPU aunque su uso de memoria principal puede ser un inconveniente. Una alternativa es el segundo punto donde los objetos se serializan y deserializan a memoria para rendibilizar mejor el espacio de memoria principal a cambio del coste de CPU de la serialización y deserialización. Finalmente el uso de disco permite trabajar con un gran volumen de datos aunque con un coste de I/O. Se pueden establecer estrategias mixtas mediante la API de Spark[5]:

```
def apply(
       useDisk: Boolean,
       useMemory: Boolean,
       deserialized: Boolean,
       replication: Int = 1): StorageLevel
```

La tercera propiedad de los RDD además de ser clave para las estrategias de resilencia distribuida también tiene como consecuencia que las transformaciones de los RDD son *lazy*, es decir, no son aplicadas hasta que no es requerido, momento en el que se dice que el RDD se materializa. En general las operaciones de agregación materializan los RDD. También las operaciones que almacenan el RDD en fichero y las que retornan los datos contenidos en el RDD. Ejemplos en la API de RDD son: *aggregate, count, first, collect, max, min, reduce, take, saveAsObjectFile, saveAsTextFile, takeOrdered, takeSample*. La consecuencia del *lazy apply* es que los RDD se generan manteniendo la información de cómo ser derivados de su ascendencia sin ser materializados y llegado el momento de materializarse se pueden efectuar optimizaciones pues la secuencia de transformaciones es conocida en el momento de la materialización. En flujos lineales la consecuencia es un ahorro de memoria pues los objetos intermedios no tiene que ser necesariamente persistidos, sino que pueden ser computados al vuelo. En el siguiente código de ejemplo en Python usando la API de Spark RDD observamos cómo primero se genera un RDD a partir de un fichero de texto, luego se efectúan tres transformaciones sobre el RDD generando cada una un nuevo RDD. Finalmente el *saveAsTextFile* materializará los RDD en cadena para finalmente persistirlo en un fichero.

```
file = spark.textFile("hdfs://...")
counts = file.flatMap(lambda line: line.split(" "))
            .map(lambda word: (word, 1))
            .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

En este ejemplo los RDD intermedios no son persistidos por Spark en el sentido que son materializados en memoria de forma temporal mientras algún descendiente solicita esta materialización para poder efectuar su propia mate-

[5]Declaración de método org.apache.spark.storage.StorageLevel$.apply (https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.storage.StorageLevel$)

rialización. Spark diferencia dos tipos de dependencias entre RDDs[18]:

- *narrow dependencies*, donde cada partición del RDD padre es usada como mucho por una partición del RDD hijo;
- *wide dependencies,* donde cada partición del RDD padre es usada por múltiples particiones del RDD hijo.

Esta diferenciación permite primero una optimización en el caso de las *narrow dependencies,* pues se puede planificar una secuencia de *narrow dependencies* en el nodo donde está la partición del parent para las particiones del hijo. Un ejemplo sería ejecutar una transformación *map* seguida de una transformación *filter* donde la primera al ser una *narrow dependency* de su antecesor las particiones del RDD resultante de la transformación estarían igual distribuidas que el antecesor reduciendo el uso de I/O. Según la especificación del RDD *MappedRDD* el RDD tiene las mismas particiones y localizaciones preferidas que el RDD parent, pero aplicando la función *map* a los items del RDD parent en su método iterador. De este modo la materialización de los RDD es a demanda y por defecto sin persistencia, es decir, la materialización no implica que el RDD persista para otros usos posteriores.

4.2. API

Spark está programado prácticamente entero en Scala, sin embargo, ofrece una API (*Application Programming Interface*) en tres lenguajes: Scala, Java y Python. La API de Ya que Scala es un lenguaje que se compila a *Bytecode* y luego es interpretado por la *Java Virtual Machine* extender la API al lenguaje Java es una tarea razonable más teniendo en cuenta la adopción que tiene Java en la indústria digital. Por otro lado Python es un lenguaje de creciente adopción en la indústria conocido por ser un lenguaje de alto nivel con gran expresividad semántica. En este caso la integración de la API de Spark es mediante la librería Py4J.

Además la API está orientada a programación funcional, es decir, las funciones que conforman la API reciben por parámetro funciones para transformar los datos de los RDD. Cuando se programa con un lenguaje funcional, como Scala (y Python en menor medida) se puede explotar esta característica para generar programas muy expresivos. Por ejemplo la función *map* sobre un RDD de objetos de tipo T toma una función que transforma el tipo original T al tipo saliente U, retornando un RDD con objetos de tipo U.

```
def map[U: ClassTag](f: T => U): RDD[U]
```

En Java, que no es un lenguaje orienado a funciones, su uso es verboso pues requiere pasar como argumentos objetos que implementan interfaces. No obstante, con las nuevas características de Java 8, como la referencia a métodos y las funciones lambda su uso será más práctico.

4.3. Graphx

GraphX es un módulo de especial interés para el experimento pues implementa una API para procesar grafos dirigidos en Spark. La clase principal es *Graph* que toma dos parámetros por tipo: *VD* para el tipo de los vértices y *ED* para el tipo de los arcos. Cada vértice está formado por un identificador numérico y los datos adjuntos de tipo VD. Los arcos están formados por el par de vértices conectados (origen y destino) y por los datos adjuntos de tipo ED. Bajo la clase *Graph* se almacena un RDD para los vértices y otro para los arcos accesibles mediante los métodos:

```
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
```

Además ofrece acceso a las tripletas, que son el trío formado por un arco y los dos vértices que conecta:

```
val triplets: RDD[EdgeTriplet[VD, ED]]
```

Se ofrecen transdormaciones específicas para grafos como *mapVertices, mapEdges, mapTriplets, mapReduceTriplets, mask, subgraph* entre otras.

5. Implementación

Dado que el algoritmo *Belief Propagation* es un claro candidato de ser paralelizado mediante el patrón de paso de mensajes que es su expresión natural y Spark a su vez ofrece una interfaz que implementa herramientas para realizar dicho patrón su combinación tarde o temprano se tenía que realizar.

Para implementar el algoritmo Belief Propagation se requiere la estructura de *cluster graph* además del modelo de factores, variables y las operaciones entre factores. Para implementar la estructura *cluster graph* se ha usado la clase *Graph* de Sparck GraphX. Sin embargo, se ha tenido que implementar enteramente el modelo de factores y las operaciones entre éstos. También la implementación tiene en cuenta la gestión de los RDD, que son un punto clave para el correcto funcionamiento de Spark. La implementación se divide en tres *packages:*

- edu.ub.guillermoblascojimenez.gmx, que contiene la declaración de la estructura principal, el *cluster graph;*
- edu.ub.guillermoblascojimenez.gmx.impl, que contiene la implementación del *cluster graph*, así como la implementación de *Belief Propagation;*
- edu.ub.guillermoblascojimenez.gmx.model, que contiene el modelo necesario subyacente al *cluster graph*, como las variables o los factores.

Dado que el factor es la estructura básica sobre la que *cluster graph* se construye por aquí se comienza a continuación.

5.1. Implementación de factor

Los factores son una entidad que surgen durante la modelización de las redes de Markov tal y como se explica en la sección 3.1.2 en la página 13 y también usados en el algoritmo Belief Propagation explicado en la sección 3.4 en la página 22. Los factores se definen sobre un conjunto D de variables aleatorias y son funciones de $Val(D)$ a $\mathbb{R}$ como por ejemplo el factor de la tabla 3 en la página 13. Dado que un factor requiere el concepto de variable comentamos antes brevemente su implementación: La representación de las variables discretas es sencilla. Constan de dos atributos. Un identificador único textual y su cardinalidad. Se asume que una variable de cardinalidad n puede tomar valores $0, ..., n-1$. La declaración e implementación se pueden encontrar en los ficheros del package edu.ub.guillermoblascojimenez.gmx.model Variable.scala y VariableImpl.scala respectivamente.

La implementación del factor está separada en varios ficheros en el package edu.ub.guillermoblascojimenez.gmx.model, que pretenden desacoplar las funcionalidades:

- Factor.scala, que contiene la declaración de la API del factor
- AbstractArrayFactor.scala, que contiene la implementación de la gestión de la estructura de datos subyaciente del factor, el *array*
- ArrayFactor.scala, que contiene la implementación de las operaciones propias de un factor definidas en Factor.scala y utilizando la estructura subyaciente ofrecida por el fichero AbstractArrayFactor.scala

Dado que un factor requiere el conjunto de variables que tiene por *scope* la API declarada en Factor.scala ofrece un método con estos requisitos:

```
def apply(variables: Set[Variable]) : Factor
```

La estructura de datos subyacente para representar un factor es un *array* unidimensional donde cada entrada corresponderá a una asignación de las variables. Para gestionar esta estructura es necesario un método de indexación de las asignaciones. Aunque un factor de variables discretas se puede representar mediante una anidación de *arrays* esta representación es ineficiente (*Double[][],* por ejemplo, para un array bidimensional). Por ejemplo, si el factor es sobre dos variables A, B se podría declarar la estructura *Double[][]* donde el índice del primer array corresponde a las asignaciones de la variable A, y las del segundo array a las de la variable B. Si $A = a, B = b$ entonces bastaría con acceder mediante la expresión *[a][b]*. Para tres variables, haría falta declarar una triple anidación y dado que un factor puede ser declarado sobre un conjunto arbitrario de variables requeriría una anidación arbitraria de *arrays (Double[]...[])*. El ejemplo por excelencia de usar una anidación de *arrays* o usar un array lineal y gestionar los índices es el de las matrices, aunque sólamente son dos dimensiones. Para el caso de matrices el proceso por el cual se transforma una estructura multidimensional en un array lineal se llama *row-major order* o *column-major*

Índice del *array*	Asignación (A, B, C)	Valor de ϕ
0	$(0, 0, 0)$	0.5
1	$(1, 0, 0)$	2
2	$(0, 1, 0)$	0.76
3	$(1, 2, 0)$	21
4	$(0, 2, 0)$	0.4
5	$(1, 2, 0)$	2.7
6	$(0, 0, 1)$	3.5
7	$(1, 0, 1)$	8.2
8	$(0, 1, 1)$	1.1
9	$(1, 1, 1)$	10.1
10	$(0, 2, 1)$	0.1
11	$(1, 2, 1)$	20

Cuadro 6: Ejemplo de representación de factor en *array* lineal

order dependiendo de si se toman primero las filas o las columnas para la indexación de los elementos. Para el caso de los factores basta con generalizar el proceso inductivamente. La ventaja de usar un array unidimensional respecto una anidación de arrays es el ahorro de memoria pues anidar arrays requiere mantener los punteros de los niveles de anidación. Luego un *array* tiene un coste de acceso aleatorio constante, es decir, es *Random Access*. Dado que algunas operaciones en factores consisten el acceso a entradas concretas con un orden diferente al del orden de indexación (como en el producto de factores, que se verá después) mantener un coste bajo en el acceso es crucial.

Para representar esta idea pongamos por ejemplo un factor ϕ de scope $\{A, B, C\}$ de variables discretas con cardinalidad $2, 3, 2$ respectivamente. Los valores que las variables pueden tomar son $A \in \{0, 1\}, B \in \{0, 1, 2\}, C \in \{0, 1\}$ tal y como se describe al principio de esta sección. Hay 2ů3ů2=12 valores que el factor puede tomar. Construiremos un array lineal de longitud 12 y asociaremos inductivamente valores a las variables de forma ordenada (A, B, C será el orden). En la tabla 6 está representado el resultado del proceso asociando valores arbitrarios en la tercera columna.

La operación de obtener el índice que le toca a una asignación $A = a, B = b, C = c$ ha sido $a + 2$ůb$+2$ů3ůc y viceversa, para obtener la asignación de variables de un índice i basta con efectuar la división entera y extraer el módulo:

$$\begin{aligned} c &= i/\left(2\text{ů}3\right) \\ b &= \left(i \bmod \left(2\text{ů}3\right)\right)/2 \\ a &= \left(\left(i \bmod \left(2\text{ů}3\right)\right) \bmod 2\right) \end{aligned}$$

Se puede generalizar inductivamente y de este modo para un conjunto de variables $X_1, ..., X_n$ con cardinalidades respectivas $k_1, ..., k_n$ y una asignación $X_i = x_i$ tal que $x_i \in \{0, ..., k_i - 1\}$ para $i = 1, ..., n$ asociar un índice unívoco

con la siguiente fórmula:

$$I = \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} k_j \right) x_i$$

También se puede efectuar en orden inverso $I' = \sum_{i=1}^{n} \left(\prod_{j=i}^{n-1} k_j \right) x_i$ y aunque la asignación de indices es diferente es igualmente funcional para indexar linealmente las asignaciones de las variables. En la implementación se ha usado la primera ordenación. Primero se calculan los llamados *strides* que son las multiplicaciones parciales $\prod_{j=1}^{i-1} k_j$, es decir, $stride[i] = \prod_{j=1}^{i-1} k_j$, para así evitar repetir este cálculo. Luego la indexación es la operación $\sum_{i=1}^{n} stride[i] \mathring{u} x_i$. Y la implementación en Scala corresponde a:

```
1  // AbstractArrayFactor.scala
2
3  def computeStrides(variables: Set[Variable]) : Map[Variable, Int] = {
4      val strides: mutable.HashMap[Variable, Int] = mutable.HashMap[Variable,
           Int]()
5      var stride = 1
6      // Variables are arranged to strides with default order since Variables
7      // are comparable objects
8      val sortedVariables: List[Variable] = variables.toList.sorted
9      sortedVariables foreach { case (v) =>
10         strides(v) = stride
11         stride = stride * v.cardinality
12     }
13     strides.toMap
14 }
15
16 protected def indexOfAssignment(assignment: Map[Variable, Int]) : Int = {
17     assert(scope.diff(assignment.keySet).isEmpty)
18     assignment.foldLeft(0)({case (z, (v, i)) => z + strides.getOrElse(v, 0) * i})
19 }
```

Con esta operación de indexación se implementan la escritura y lectura de valores del factor:

```
// ArrayFactor.scala

def update(assignment: Map[Variable, Int], value: Double) =
       values(indexOfAssignment(assignment)) = value

def apply(assignment: Map[Variable, Int]) : Double =
       values(indexOfAssignment(assignment))
```

Cabe decir que el tipo de la asignación (*Map[Variable, Int]*) es una forma natural de asociar a cada variable de un conjunto el valor que toma. En el caso

que la asignación no contenga una variable que forme parte del *scope* del factor, que es un escenario de error, una excepción de aserción se lanzará, pues es la primera comprobación que efectúa el método *indexOfAssignment*. Sin embargo, si en la asignación hay variables que no forman parte del *scope* del factor serán ignoradas, es decir, se les asigna como *stride* el valor 0.

El coste de acceder a un valor dada una asignación es $O(n)$ con n el número de variables. Análogamente sea m la cantidad de valores del factor entonces el coste es $O(\log m)$. Esta operación de indexación es clave según la cita:

> *«One of the keys design decisions is indexing the appropiate entries in each factor for the operations that we wish to perform. (In fact, when one uses a naive implementation of index computations, one often discovers that 90 percent of the running time is spent in this task)»*(Probabilistic Graphical Models: Principles and Techniques, Box 10.A)

Se han implementado dos proyecciones de factor: la marginalización y la marginalización MAP (*Maximum a posteriori*). Una proyección consiste en una función que transforma factores con *scope* X en factores con *scope* Y, donde $Y \subset X$. La marginalización consiste en sumar los valores de las asignaciones en X que coninciden con las asignaciones de Ymientras que la MAP marginalization consiste en tomar el máximo. La figura //TODO// es un ejemplo de estos procesos. La implementación es directa en el fichero ArrayFactor.scala con los métodos *marginalize* y *maxMarginalize*:

```
override def marginalize(variables: Set[Variable]): Factor = {
       assert((variables diff scope).isEmpty)
       val X: Set[Variable] = scope diff variables
       val phi: ArrayFactor = ArrayFactor(X, 0.0)
       for (i <- 0 until this.size) {
              val assignment = this.assignmentOfIndex(i)
              phi(assignment) = phi(assignment) + this.values(i)
       }
       phi
}
```

Dado un factor ψ, referenciado por `this` en el código, con scope Z y un conjunto de variables Y, referenciado por la `variables` en el código, se genera un factor ϕ con scope $Z \setminus Y$ en la linea 4. A continuación, en la linea 6, se itera sobre los índices del array extrayendo la asignación que asociada al índice con el proceso inverso de indexación explicado al principio de la sección. Luego se acumulan el valor de la asignación de ψ en ϕ mediante los métodos anteriormente expuestos. La implementación de la marginalización MAP es idéntica salvo que en lugar de sumar se toma el máximo de los valores. El factor resultante no contiene el conjunto de variables que se han pasado por parámetro. También hay métodos *marginal, maxMarginal* que marginalizan las variables que forman parte del *scope* del factor pero no están en el conjunto de variables pasadas por parámetro. El coste computacional es $O(m \log m)$, con m el número de valores

que tiene el factor original, dado que se itera sobre todos ellos y por cada uno se efectuan operaciones de asignación que como ya hemos visto antes tienen coste $O(\log m)$.

Dos factores con *scope* idéntico pueden multiplicarse entre ellos como en la definición 5 en la página 14 se describe.

La implementación del producto de factores figura en el fichero AbstractArrayFactor.scala y permite personalizar la función de producto que se aplica a los valores. Por ejemplo, podría efectuarse un producto de log-factores (factores cuyos valores han sido logaritmos) pasando como operación la suma, pues dado que son logaritmos no se deben multiplicar, sino que sumar. No obstante, estos usos quedan como expectativas futuras del código.

```
1  def product[I <: AbstractArrayFactor](phi1: I, phi2: I, psi: I, op: (Double, Double) =>
       Double) : I = {
2      assert(phi1 != null)
3      assert(phi2 != null)
4      assert(psi != null)
5      assert(op != null)
6      val X1 = phi1.scope
7      val X2 = phi2.scope
8      val X: Set[Variable] = X1 ++ X2
9      val sortedX = X.toList.sorted
10     assert(X.equals(psi.scope))
11     val assignment: mutable.Map[Variable, Int] = mutable.HashMap[Variable, Int]()
12     for (v <- X) {
13         assignment(v) = 0
14     }
15     var j = 0      // phi1 index
16     var k = 0      // phi2 index
17     for (i <- 0 until psi.size) {
18         psi.values(i) = op(phi1.values(j), phi2.values(k))
19         breakable {
20             sortedX foreach { case (v) =>
21                 assignment(v) = assignment(v) + 1
22                 if (assignment(v) equals v.cardinality) {
23                     assignment(v) = 0
24                     j = j - (v.cardinality - 1) * phi1.strides.getOrElse(v, 0)
25                     k = k - (v.cardinality - 1) * phi2.strides.getOrElse(v, 0)
26                 } else {
27                     j = j + phi1.strides.getOrElse(v, 0)
28                     k = k + phi2.strides.getOrElse(v, 0)
29                     break()
30                 }
31             }
32         }
33     }
34     psi
35 }
```

El algoritmo itera los índices del factor ψ en la linea 17, luego agrega los valores de los factores ϕ_1, ϕ_2 en la linea 18. Después, en de la linea 19 a la 32, los índices j, k correspondientes a ϕ_1, ϕ_2 respectivamente se actualizan en un proceso de indexación que es derivado del explicado al principio de la sección.

La complejidad del algoritmo es $O(n \mathring{u} m)$, con $n$el número de valores del factor ψ y m el número de variables del factor ψ. Ya que n es combinatoria de los valores de las variables la complejidad se puede simplificar en $O(n \mathring{u} \log n)$.

Con esta operación de producto se implementa el producto de factores en ArrayFactor.scala. También se ofrecen métodos para invertir un factor, que consiste en invertir todos sus valores con la consideración que $0^{-1} = 0$. De este modo se puede implementar la división de factores $\phi/\psi = \phi \times \psi^{-1}$. También hay métodos para obtener la función de partición y normalizar un factor.

Los factores solamente son modificados via la asignación directa de valor. El resto de operaciones generan un factor nuevo. Por ejemplo:

```
// Let be phi a Factor
phi(assignment) = 0.3 // modifies the factor
val psi_1 = phi.normalized() // generates a new factor and phi is
    unmodified
val psi_2 = phi * phi // generates a new factor with phi^2 and phi is
    unmodified
val psi_3 = phi / 5 // generates a new factor and phi is unmodified
```

Estos son todos los detalles de implementación del factor, que son utilizados en la implementación de Belief Propagation explicada a continuación.

5.2. Implementación de Belief Propagation

Con la implementación de factor realizada se puede implementar el *cluster graph*, que es un grafo cuyos vértices son factores y los arcos son la intersección de *scopes* de los factores que conecta, llamado *sepset*. Esta estructura de *cluster graph* es sobre la que opera el algoritmo Belief Propagation, por lo que la explicación de la implementación comienza por aqui. El fichero ClusterGraph.scala contiene la declaración de la API de la clase ClusterGraph. La declaración contiene como único atributo un objeto Graph definido como Graph[Factor, Set[Variable]] en la linea 3, y tal y como se exige, tanto la clase Factor, como la clase Variable son serializables. Se ofrecen dos métodos que calibran el grafo, retornando un nuevo grafo calibrado, además de otros métodos de menor relevancia en las lineas 6 y 8.

```
1  abstract class ClusterGraph protected () extends Serializable {
2        /** Underlying raw graph */
3        val graph: Graph[Factor, Set[Variable]]
4
5        /** Calibrates the marginals of the graph */
6        def calibrated(maxIters:Int = 10, epsilon:Double=0.1) :
             ClusterGraph
7
8        /** Calibrates MAPs of the graph */
9        def map(maxIterations :Int = 10, epsilon:Double=0.1):
             ClusterGraph
10
```

```
        [...]
}
```

Para construir un objeto ClusterGraph puede crearse directamente dado el objeto RDD Graph subyacente, que puede ser generado a partir de cualquier otro RDD o de datos materiales (ficheros, bases de datos,...). Dado que un *cluster graph* es generado a partir de una red bayesiana o una red de Markov tendría sentido, dada la implementación de dichos modelos, implementar las funciones para transformarlas en un *cluster graph,* sin embargo, esta tarea queda fuera de la intención de este proyecto, aunque dentro de su ambición. Así que también se ofrece un constructor que recibe por parámetro el conjunto de factores asociado a cada cluster y el conjunto de pares de clusters conectados, con el objetivo de poder definir programáticamente estas estructuras.

```
def apply
       (clusters: Map[Set[Variable], Set[Factor]],
        edges: Set[(Set[Variable], Set[Variable])],
        sc: SparkContext) : ClusterGraph = {
       ClusterGraphImpl(clusters, edges, sc)
}
```

Ambos métodos *calibrated* y *map* ejecutan el mismo algoritmo pero utilizando proyecciones de factores diferentes. La implementación del algoritmo se encuentra en el fichero BeliefPropagation.scala. El algoritmo es iterativo, y su finalización viene dada por un error que se debe minimizar. De la implementación se comentará el detalle de esta a nivel programático, la relación con el algoritmo original Belief Propagation y la gestión de los RDD. El esqueleto del algoritmo está en la función *apply* y es el siguiente:

```
/*
 * Core BP algorithm
 */
def apply
        (projection: (Factor, Set[Variable]) => Factor,
         maxIterations : Int,
         epsilon : Double)
        (graph : Graph[Factor, Set[Variable]])
        : Graph[Factor, Set[Variable]] = {

        assert(maxIterations > 0)
        assert(epsilon > 0.0)

        // deltas are set
        var g: Graph[BPVertex, Factor] = graph
                .mapVertices((id, f) => BPVertex(f))
                .mapEdges((edge) => Factor.uniform(edge.attr))
                .cache()
        var iteration: Int = 0
        var iterationError: Double = Double.PositiveInfinity
        do {
                val newG = iterate(projection)(g).cache()
```

```
23                  iterationError = calibrationError(g, newG)
24
25                  // unpersist things
26                  g.unpersistVertices(blocking = false)
27                  g.edges.unpersist(blocking = false)
28
29                  // update cached things
30                  g = newG
31                  iteration += 1
32          } while (iteration < maxIterations && iterationError > epsilon)
33
34          g.unpersistVertices(blocking = false)
35          g.edges.unpersist(blocking = false)
36
37          val outputGraph = g
38                  .mapVertices((id, v) => v.potential().normalized())
39                  .mapEdges((edge) => edge.attr.scope)
40          outputGraph
41  }
```

El algoritmo toma una función de proyección (que dado un factor y un conjunto de variables como espacio, proyecta el factor en éste espacio; por ejemplo, la marginalización), una cantidad máxima de iteraciones, y un epsilon que es la cota superior de error. Luego toma como parámetro un grafo con estructura de *cluster graph*. De la linea 15 a la 20 se inicializan las variables de control (*iteration* y *iterationError*) y se inicializa el grafo, que contendrá los factores originales y los mensajes inicialmente uniformes. Después se ejecuta el bucle *do-while* en el que primero se procesa la siguiente iteración de Belief Propagation, luego se computa el error de calibración, después se eliminan los RDD persistentes innecesarios y finalmente se actualizan las variables de control. El bucle termina cuando se ha alcanzado el máximo de iteraciones o bien cuando el error de calibración es menor que la cota de error epsilon. Al terminar el bucle, se vuelve a estructuran los datos de nuevo en el formato de *cluster graph,* se dejan de persistir los RDD innecesarios y finalmente se retorna el grafo calibrado y normalizado.

El grafo que se declara en la linea 15 tiene por vértices el par formado por el factor del clúster junto con el producto de mensages entrantes. Luego los arcos contienen los mensajes. La declaración es Graph[BPVertex, Factor], y el tipo BPVertex corresponde a la siguiente declaración tal y como se ha comentado previamente, que además permite el cálculo del potencial del cluster como el producto del factor original y de los mensajes entrantes.

```
private class BPVertex (val factor: Factor, val incomingMessages:
    Factor) extends java.io.Serializable {
       def potential() : Factor = factor * incomingMessages
}
```

Recordemos que el algoritmo Belief Propagation se basa en la generación de mensajes mediante la proyección del potencial en el *sepset,* y luego estos mensajes son recibidos por el cluster que los agrega en su potencial. De este

modo el método *iterate*, que computa la siguiente iteración de Belief Propagation tiene la implementación:

```
1  private def iterate
2         (projection: (Factor, Set[Variable]) => Factor)
3         (g : Graph[BPVertex, Factor]) : Graph[BPVertex, Factor] = {
4         // compute new deltas
5         // for each edge i->j generate delta as
6         // i->j_message = i_potential / j->i_message
7         // Trick: for each edge i->j set j_potential / i->j_message and
              then
8         // reverse all edges
9         val newDeltas = g
10                .mapTriplets((triplet) =>
                      projection(triplet.dstAttr.potential() /
                      triplet.attr, triplet.attr.scope))
11                .reverse
12
13        // Compute new potentials and put them into a new graph
14        // for each node i collect incoming edges and update as:
15        //             i_potential := PRODUCT [j->i_potential, for j in
              N(i)]
16        val messages = newDeltas
17                .mapReduceTriplets((triplet) => Iterator((triplet.dstId,
                      triplet.attr)), reduceDeltas)
18
19        // keep the factor and update messages
20        val newG = newDeltas
21                .outerJoinVertices(g.vertices)((id, v, bpVertex) =>
                      bpVertex.get.factor)
22                .outerJoinVertices(messages)((id, factor, message) =>
                      BPVertex(factor, message.get))
23         newG
24 }
```

Primero se computan los mensajes en la linea 9. Son computados mediante la fórmula $\delta_{i\to j} = \pi_{i,j}\left(\beta_i\right)/\delta_{j\to i}$ donde $\pi_{i,j}$ es la proyección del sepset de los clusters i, j, es decir, $scope\left[\beta_i\right] \cap scope\left[\beta_j\right]$. La operación *mapTriplets* mapea cada arco con la información de la tripleta. En este caso, dado que el mensaje que un cluster i envía a un cluster j ha de ser dividido por el mensaje $\delta_{j\to i}$ se utiliza la función *reverse* tal y como el comentario del código indica. Esto es porque al usar la función *mapTriplets* sólamente se tiene visibilidad del vértice origen, del vértice destino y del arco, es decir, de $\beta_i, \beta_j, \delta_{i\to j}$, lo que imposibilita el cálculo nuevo de $\delta_{i\to j}$ ya que el mensaje $\delta_{j\to i}$ no está disponible. Sin embargo, sí se puede calcular para el arco $i \to j$ el mensaje $\delta_{j\to i}$, es por ello que luego se revierte la dirección de los arcos en la linea 11, de este modo el arco $j \to i$ contiene el mensaje $\delta_{j\to i}$ que es lo que queríamos. Luego los mensajes se agregan para cada clúster en la linea 16, es decir, que el objeto *messages* es

una colección de vértices que contiene el producto de los mensajes entrantes, el vértice i contiene $\prod_{j \in Neighbors(i)} \delta_{j \to i}$. La función *mapReduceTriplets* permite mapear cada tripleta y asociar el resultado a un vértice, luego los resultados para cada vértice se agregan con la función de reducción. El objeto resultante es un conjunto de vértices con el resultado de la agregación. Esta función es la que implementa a efectos prácticos el patrón de paso de mensajes. Ahroa que se tienen los mensajes y los mensajes agregados hay que reconstruir el grafo con esta información. En la linea 20, se construye un grafo que tiene por arcos los mensajes calculados en la linea 9 y por vértices el par formado por los factores originales y los mensajes entrantes agregados de cada cluster. En este punto el grafo resultante tiene los potenciales actualizados con los nuevos mensajes y en los arcos los mensajes enviados.

Ahora que hemos visto cómo iterar el algoritmo se ha de ver cómo se calcula el error. En este caso se espera la estabilización de los mensajes. Se comparan los mensajes de la iteración anterior y de la nueva y se calcula la distancia entre cada par, luego las distancias se suman y es considerado el error de la calibración.

```
private def calibrationError(
       g1 : Graph[BPVertex, Factor],
       g2 : Graph[BPVertex, Factor]) : Double = {
              g1.edges
                     .innerJoin(g2.edges)((srcId, dstId, ij, ji) =>
                         Factor.distance(ij.normalized(),
                         ji.normalized()))
                     .aggregate(0.0)((e, errorEdge) => e +
                         errorEdge.attr, _ + _)
}
```

En la liena 5 se combinan los grafos de las dos iteraciones extrayendo de cada par formado por el arco de la iteración anterior y el mismo arco de la iteración nueva su distancia. En la linea 6 se agregan los errores sumándolos. Dos factores de igual *scope* se puede medir su distancia según lo diferentes que sean los valores que tienen para sus asignaciones de valores de variable. Por ende, basta considerar la lista de valores de un factor con una indexación fijada como un vector real. Con esto la distancia de factores es la distancia euclídea de los vectores de valores. La implementación de esta distancia está en el fichero ArrayFactor.scala.

En este punto ya hemos hecho una valoración de la programación necesaria y de cómo el algoritmo se ha llevado a cabo. No obstante, sólamente se ha comentado la utlidad de la API de Spark y GraphX para implementar el algoritmo. También se debe hacer mención de la gestión de los objetos RDD.

5.3. Gestión de los objetos RDD

En la sección 4.1 en la página 27 se explica la teoría que soporta los RDDs así como algún ejemplo para clarificar ideas. No obstante, los ejemplos son casos

sencillos con flujos lineales y por tanto no se puede explotar la persistencia de los RDDs. Para flujos más complejos (como el de este algoritmo) es conveniente persistir los RDD adecuados para optimizar la materialización de los RDD descendientes. Un ejemplo son los algoritmos iterativos. El siguiente *toy example* en pseudocódigo pone en uso esta característica:

```
data = Spark.loadRddData(...)
error = Infinity // computed error
e = 0.001 // some error bound
while (error < e) {
       // process the data given a processData function
       data = data.map(processData)
       // compute error given a processError function
       error = data.aggregate(processError)
}
```

La función *data.aggregate* materializa el RDD, mientras que la función *map* genera un RDD nuevo que desciende del anterior RDD alojado en la variable *data*. De éste modo para computar el error de la iteración $n + 1$ se requiere materializar el RDD generado en la iteración $n + 1$ que a su vez requiere la materialización del RDD de la iteración n, como se puede ver en la figura 4.1. Cada iteración se vuelve más costosa pues requiere la materialización de los RDDs generados en las iteraciones anteriores pues los RDDs no se persisten. El siguiente ejemplo modificado sí usa la persistencia:

```
data = Spark.loadRddData(...)
oldData = data
error = Infinity // computed error
e = 0.001 // some error bound
while (error < e) {
       oldData = data
       // process the data given a processData function
       data = data.map(processData)
       // persist the new data
       data.persist()
       // compute error given a processError function
       error = data.aggregate(processError)
       // unpersist the old data
       oldData.unpersist()
}
```

Aqui cada objeto RDD en la variable *data* es persistido con la instrucción *data.persist()*, de forma que cuando su materialización es requerida no es requerido materializar los RDD de sus ascedentes. La consecuencia es que la transformación *map* no requiere la materialización de todos los RDD antecesores, basta con la materialización del RDD generado en la iteración anterior pues es persistente, como se puede ver en la figura 4.2. La instrucción *oldData.unpersist()* hace que el RDD de la iteración anterior deje de ser persistente ya que el nuevo

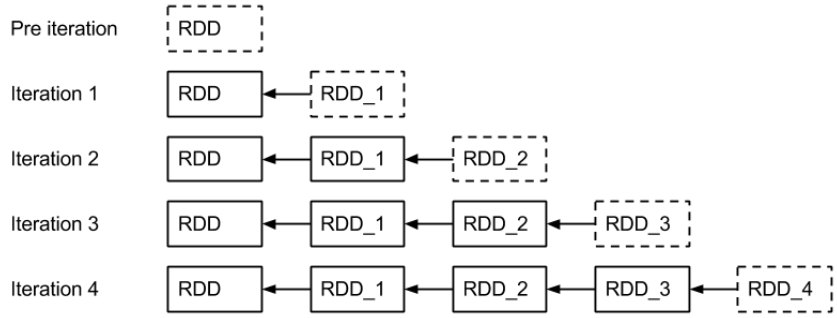

Figura 5.1: Esquema de la ascendencia de los RDDs sin persistencia. Las flechas indican las dependencias de ascendencia donde el origen es el descendiente y el destino el ascendente, los RDD con línea discontinua son los generados en dicha iteración.

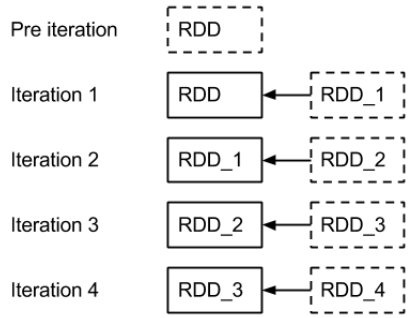

Figura 5.2: Esquema de la ascendencia de los RDDs, tal que todos son persistidos. Las flechas indican las dependencias de ascendencia donde el origen es el descendiente y el destino el ascendente, los RDD con línea discontinua son los generados en dicha iteración.

RDD es persistente por lo que la materialización del anterior ya no es requerida de nuevo. Sin esta instrucción todos los RDD serían persistidos generando una fuga de memoria.

Los módulos de Spark hacen un uso regular de esta estrategia. En particular los algoritmos del módulo de aprendizaje máquina MlLib[6]. Ejemplos son la implementación en Spark de los algoritmos ALS (*Alternating Least Squares matrix factorization*) y Pregel. El objetivo de la persistencia de RDD es mejorar el rendimiento de la materialización aunque son una fuente de *bugs* relacionados con la fuga de memoria, por ende cada caso de uso debe ser analizado debidamente. Sin embargo, para mejorar la usabilidad de esta característica los RDDs persistidos son dejados de persistir cuando el *driver program* (que es el que se

[6]https://spark.apache.org/mllib/

ejecuta en la máquina master) deja de tener acceso[7].

Esto se aplica a la implementación efectuada tal y como el siguiente código (del cual se han borrado lineas irrelevantes para la gestión de los RDD y además comentadas ya previamente) se puede observar el mismo patrón:

```
def apply (...) {
        // deltas are set
        var g: Graph[BPVertex, Factor] = // ...

        do {
                val newG = iterate(projection)(g).cache()

                // unpersist things
                g.unpersistVertices(blocking = false)
                g.edges.unpersist(blocking = false)

                // update cached things
                g = newG
        } while (/* end condition */)

        g.unpersistVertices(blocking = false)
        g.edges.unpersist(blocking = false)

        val outputGraph = // transform g ...
        outputGraph
}
```

En la línea 6 se genera el nuevo objeto de la iteración y se cachea. Luego el objeto de la iteración anterior deja de ser persistido en las lineas 9 y 10. Luego en la línea 13 se sustituye el RDD de la iteración anterior por el de la nueva iteración. Después del bucle el RDD de la última iteración deja de ser persistido, pues ahora su persistencia no tiene utilidad pues solamente se usará este RDD para terminar de transformar el objeto para poder ser presentado al usuario. Aunque en cada iteración se generan múltiples RDDs en el método *iterate* cada RDD solamente depende o bien de un RDD de su misma iteración, o bien de un RDD de la iteración anterior que es persistente.

5.4. Entorno de desarrollo

La implementación va acompañada de una serie de herramientas para facilitar su desarrollo en sus diferentes etapas.

5.4.1. Maven

Maven es un programa gestor del pipeline de compilación de Java. Incluye la gestión de las dependencias, la compilación y los tests. Además puede ser

[7]Spark Issue 1103 (https://issues.apache.org/jira/browse/SPARK-1103)

extendida su funcionalidad mediante plugins. Es un software usado de forma sistemática en la indústria.

5.4.2. ScalaStyle

Es un programa validador del estilo de programación del lenguaje Scala. Usado para mejorar el mantenimiento del código.

5.4.3. GitHub

GitHub es un provedor de repositorios remotos git. Git, a su vez, es un sistema de control de versiones para código fuente también usado sistemáticamente en la indústria digital y sucesor natural de SVN. El repositorio es público y tiene por dirección electrónica https://github.com/GuillermoBlasco/gmX.

5.4.4. IntelliJ IDEA

Es un IDE (Interface Development Environment) desarrollado por la empresa IntelliJ centrado en el desarrollo de código Java. Tiene múltiples herramientas que integran diferentes tecnologías en el entorno de trabajo como Maven, Scalastyle o git.

5.5. Resultados

Por un lado la implementación de las operaciones del factor han sido testeadas mediante tests unitarios extraídos de los modelos de ejemplo de los autores Daphne Koller y Nir Friedman [5]. La corrección de la implementación se ha realizado comparando las trazas de ejecución de múltiples ejemplos con las de la implementación centralizada disponible en la librería BinaryMaxSum [11]. La ambición era poder probar la implementación sobre un grafo de tamaño medio usando varias máquinas a la vez. Sin embargo, aunque el despliegue de Spark es, según el manual, sencillo, la preparación de las instancias remotas, la configuración de éstas, el despligue del software y luego las comprobaciones tienen un coste alto en tiempo y recursos. Además se requieren también conocimientos de administración de sistemas que no era el propósito de este trabajo.

El código está publicado bajo una licencia BSD en el proveedor de repositorios GitHub, con dirección https://github.com/GuillermoBlasco/gmX. El artefacto resultante de la compilación del código ha sido publicado mediante un servidor HTTP para poder ser importado por otros proyectos como dependencia (via Maven, u otro mecanismo). La documentación del código está publicada también en el mismo servidor HTTP. El servidor es una instancia de Amazon Web Services (t2.micro), que es de forma temporal gratuita. Los enlaces son los siguientes:

- http://ec2-54-148-53-205.us-west-2.compute.amazonaws.com/maven2/, la dirección del repositorio de artefactos de software

- http://ec2-54-148-53-205.us-west-2.compute.amazonaws.com/gmx/scaladocs/, la dirección de la documentación del código.

6. Conclusiones y trabajo futuro

Llevar a cabo este proyecto ha requerido conocimientos de dos disciplinas: las matemáticas para PGM y la ingeniería para Spark. Los conocimientos expuestos en esta memoria sobre PGM son los necesarios para poder entender el algoritmo principal y una síntesis de todos los adquiridos durante el proceso de elaboración de este trabajo. En el otro campo, se ha extraído información de múltiples referencias para exponer diferentes soluciones en la parte tecnológica. Spark incorpora los conceptos propios de los sistemas distribuidos y, además, contiene conceptos innovadores, que es lo que ha hecho diferenciarse del resto de sistemas distribuidos. Pese a la juventud de Spark y PGM, se ha desarrollado un código funcional uniendo los ambos mundos con resultados satisfactorios. También este proyecto me ha empujado a participar en los eventos y grupos de trabajo de la comunidad local de computación distribuida.

Además, el presente proyecto puede ser extendido en múltiples líneas de trabajo:

- Implementación de variantes del algoritmo Belief Propagation, como *Tree-based reparametrization, Norm-product Belief Propagation,* Belief Propagation en espacio logarítmico, entre otros.
- Implementación de otros métodos relacionados con modelos gráficos probabilísticos como métodos de aprendizaje, para hacer así una librería que cubra el ciclo de vida del modelo.
- Adecuar la implementación a las nuevas versiones de Spark que se han publicado, así como proponer a la comunidad la inclusión de este proyecto en su módulo de aprendizaje máquina (*Mllib*).

Referencias

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Sixth Symposium on Operating System Design and Implementation*, dec 2004.

[3] James Nicholas 'Jim' Gray. Sort benchmark. http://sortbenchmark.org/, 2015.

[4] Systems Infrastructure Team Grzegorz Czajkowski. Large-scale graph computing at google. http://googleresearch.blogspot.com.es/2009/06/large-scale-graph-computing-at-google.html, jun 2009.

[5] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, first edition, jul 2009.

[6] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[7] Malewicz, Grzegorz, Austern, Matthew H., Bik, Aart J.C, Dehnert, James C., Horn, Ilan, Leiser, Naty, Czajkowski, and Grzegorz. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[8] Frank McSherry. Scalability! but at what cost? http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html, 2015.

[9] Friedman N, Linial M, Nachman I, and Pe'er D. Using bayesian networks to analyze expression data. *Journal of Computational Biology*, 3-4(7):601–620, 2000.

[10] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[11] Marc Pujol and Toni Penya-Alba. binarymaxsum. http://binarymaxsum.github.io/, 2014.

[12] Yahoo! Research. Cutting. research.yahoo.com/files/cutting.pdf, 2008.

[13] Konstantin V Shvachko and Arun C Murthy. Scaling hadoop to 4000 nodes at yahoo! developer.yahoo.com/blogs/hadoop/scaling-hadoop-4000-nodes-yahoo-410.html, sep 2008.

[14] Sumeet Singh. Apache hbase at yahoo! - multi-tenancy at the helm again. developer.yahoo.com/blogs/hadoop/apache-hbase-yahoo-multi-tenancy-helm-again-171710422.html, jan 2013.

[15] J. Uebersax. *Genetic Counseling and Cancer Risk Modeling: An Application of Bayes Nets.* Ravenpack International, Marbella, Spain, 2004.

[16] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, aug 1990.

[17] Reynold Xin. Spark officially sets a new record in large-scale sorting. http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html, nov 2014.

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.