Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

INDUSTRIA INGENIARITZA TEKNIKOKO ATALA
SECCIÓN INGENIERÍA TÉCNICA INDUSTRIAL

GRADO EN INGENIERÍA INFORMÁTICA DE GESTIÓN Y SISTEMAS DE INFORMACIÓN

TRABAJO FIN DE GRADO

2015 / 2016

EXTENSIÓN DE UN PAQUETE DE ALGORITMOS METAHEURÍSTICOS EN R PARA LA DOCENCIA

MEMORIA

DATOS DE LA ALUMNA O DEL ALUMNO

NOMBRE ANDER
APELLIDOS CARREÑO LÓPEZ

FDO.:
FECHA: 17-06-2016

DATOS CODIRECCIÓN

NOMBRE JOSU
APELLIDOS CEBERIO URIBE
DEPARTAMENTO LSI

FDO.:
FECHA: 17-06-2016

NOMBRE BORJA
APELLIDOS CALVO MOLINOS
DEPARTAMENTO CCIA

FDO.:
FECHA: 17-06-2016

Ander Carreño López: *Extensión de un Paquete de Algoritmos Metaheurísticos en R para la Docencia*, 16/06/2016.

Trabajo Fin de Grado presentado dentro del Grado en Ingeniería Informática de Gestión y Sistemas de Información en la Escuela Universitaria de Ingenieros de Bilbao (UPV-EHU).

A mi familia y amigos.

RESUMEN

En los últimos años se han realizado numerosos avances en el campo de la optimización combinatoria. Los algoritmos evolutivos han demostrado ser una herramienta muy valiosa para resolver problemas de optimización costosos. En concreto, los Algoritmos de Estimación de Distribuciones han atraído la atención de los investigadores y se han hecho numerosos avances al respecto. Sin embargo, no existe software que acerque estos avances a los alumnos. Este Trabajo de Fin de Grado trata de extender la librería *metaheuR* para que los estudiantes puedan comparar, combinar y aprender estos algoritmos resolviendo los problemas de optimización clásicos junto con el libro *Bilaketa Heuristikoak* aun en desarrollo. Bajo el lenguaje de programación R, este proyecto recoge 3 Algoritmos de Estimación de Distribuciones, el *Estimation of Bayesian Network Algorithm* que aprende una red Bayesiana como modelo probabilístico; el *Edge Histogram Based Sampling Algorithm* que aprende probabilidades marginales de orden dos y el *Plackett-Luce Estimation of Distribution Algorithm* que aprende un vector de pesos de la población. Además, se han realizado experimentos que verifican el correcto funcionamiento de los modelos implementados así como ejecuciones sobre problemas de optimización tales como el *Travelling Salesman Problem* o el *Maximum Independence Set*. Con estos resultados, también se han realizado comparativas y análisis de los resultados que dan una visión de la eficiencia de cada uno de los algoritmos.

AGRADECIMIENTOS

En primer lugar agradecer a los doctores Josu Ceberio y Borja Calvo por su excelente colaboración y dedicación. Sus sugerencias y correcciones han hecho posible este trabajo. Además, siempre han tenido una gran disponibilidad digna de agradecer.

Agradecer también a Nerea Martín por su constante apoyo especialmente en los momentos de mayor carga de trabajo.

Agradecer a mi hermano Asier por las recomendaciones y por el apoyo durante todo el trabajo. Así mismo, agradecer a mi padre Javier y mis abuelas Mª Luisa y Dolores por la ilusión y emoción que me han transmitido.

Por último agradecer a la Escuela Universitaria de Ingenieros de Bilbao por la posibilidad de realizar este proyecto.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

En este capítulo se introduce el trabajo de fin de grado realizado por Ander Carreño López titulado *"Extensión de un Paquete de Algoritmos Metaheurísticos en R para la Docencia"*.

1.1 ESTRUCTURA DEL DOCUMENTO

Este documento se compone de 8 capítulos y 2 apéndices. El Capítulo 1 es una introducción general. El Capítulo 2 se centra en los conceptos teóricos de los algoritmos mencionados a lo largo del documento. En el Capítulo 3 se recoge el objetivo principal de este proyecto, la arquitectura, el alcance, la especificación de tareas, el diagrama de *Gantt*, los riesgos posibles a lo largo del proyecto, y la evaluación económica. El Capítulo 4 se dedica a la captura de requisitos del proyecto. A continuación, en el Capítulo 5, se describe el análisis y diseño propuesto con los diagramas de clases y de secuencia que corresponden a los métodos implementados en el trabajo. En el Capítulo 6 se expone la cómo se ha desarrollado el trabajo. El Capítulo 7 plasma la experimentación realizada para comprobar los algoritmos desarrollados. Las conclusiones y el trabajo futuro forman el Capítulo 8, siendo este el último capítulo del documento.

1.2 PROBLEMA ABORDADO

En este proyecto se aborda el desarrollo de una extensión de un paquete de algoritmos metaheurísticos para la docencia en el lenguaje de programación R. El paquete tiene el nombre de *metaheuR* y está siendo desarrollada como material didáctico para la asignatura *Bilaketa Heurístikoak*[1] junto con el libro que se está escribiendo en paralelo.

Dado que es un paquete con algoritmos puramente científicos, el desarrollo de estos no es trivial. El fundamento matemático en el que se apoyan hace que la adquisición de los conocimientos sea un trabajo añadido que se debe cumplimentar previamente. Además, el paquete consta con un único algoritmo de estimación de distribuciones implementado. Por otro lado, el marco de programación científica es

1 La asignatura *Bilaketa Heuristikoak* se imparte en el grado de Ingeniería Informática en la Facultad de San Sebastián

diferente a lo desarrollado en el grado y requiere de técnicas de depuración e implementación diferentes.

Esto se aplica al lenguaje de programación R que es un lenguaje de programación vectorial de software libre bajo licencia GNU muy extendido en la comunidad científica por su capacidad en cálculo vectorial entre otros. Además, es uno de los lenguajes más utilizados para el desarrollo de aplicaciones de estadística computacional y análisis de datos [2]. Sin embargo, el paradigma y la filosofía de R difiere de un lenguaje de programación orientado a objetos y por tanto, aprender este lenguaje de programación es una tarea añadida.

1.3 MOTIVACIÓN

La optimización es uno de los campos más ambiciosos de la computación. Actualmente abarca problemas de distintos ámbitos como el industrial, el bioinformático, el financiero y la inteligencia artificial, entre otros. Hoy en día no existen muchos softwares que permitan implementar, diseñar y utilizar algoritmos de optimización combinatoria con enfoque didáctico. Por este motivo, el principal objetivo del trabajo es extender el paquete *metaheuR* para poder ofrecer así una herramienta que acerque estos algoritmos a los alumnos. Además, con los algoritmos desarrollados, los usuarios serán capaces de resolver problemas de optimización comunes, como el problema del agente viajero (*Travelling Salesman Problem*), el problema de la asignación cuadrática (*Quadratic assignment problem*), el problema de la mochila (*Knapsack problem*), entre otros.

1.4 PROPÓSITO

El propósito de este proyecto consiste en ampliar el apartado de algoritmos metaheurísticos de el paquete de R *metaheuR* implementando diferentes algoritmos de estimación de distribuciones (EDAs) [Mühlenbein y Paass, 1996]. Estos algoritmos se diferencian en la etapa de aprendizaje y muestreo del modelo probabilístico que los caracteriza, por ello, este TFG se centrará en estas tareas.

Los algoritmos desarrollados son los siguientes: *Bayesian Network Estimation Algorithm* (EBNA) [Bengoetxea y col., 2002; Etxeberria, Pedro Larrañaga y Picaza, 1997], el cual implica el aprendizaje y muestreo de una red Bayesiana; *Edge Histogram-Based Sampling Algorithm* (EHBSA) [Tsutsui, 2002; Tsutsui y col., 2003] en el que, en el dominio de permutaciones, se aprenden probabilidades marginales de orden dos; y *Plackett-Luce Estimation of Distribution Algorithm* [Josu. Ceberio

2 Análisis de uso de lenguajes de programación https://blog.uchceu.es/informatica/ranking-de-los-lenguajes-de-programacion-mas-usados-para-2015/

y col., 2013], un algoritmo desarrollado específicamente para optimizar problemas de permutaciones.

Para este proyecto además se incluye el paquete externo *bnLearn* [Scutari, 2009] que implementa lo necesario para el aprendizaje y muestreo de redes Bayesianas.

Para ilustrar el correcto funcionamiento, se han realizado dos tipos de pruebas, en primer lugar, se ha verificado el funcionamiento de los métodos de aprendizaje, muestreo y cálculo de probabilidad desarrollados y, en segundo lugar, se han realizado ejecuciones de los algoritmos sobre los problemas clásicos de optimización como *Maximum Independence Set* (MIS) y *Travelling Salesman Problem* (TSP).

Por último, cabe decir que *metaheuR* es un proyecto de software libre que se puede obtener de *GitHub* en la siguiente dirección: `https://github.com/b0rxa/metaheuR`.

1.5 RAZONES DE LA ELECCIÓN DEL TFG

La elección de este proyecto ha sido motivada por una serie de razones que se describen a continuación.

- Estrecha relación con los proyectos personales. Dado que me gustaría continuar mi formación académica en el ámbito de la inteligencia artificial, creo que es conveniente introducirme en esta materia.

- Las herramientas y los lenguajes utilizados para el desarrollo de este proyecto son libres y multiplataforma. Una de las premisas más importantes a la hora de desarrollar algo es que sea lo más accesible posible para cualquier tipo de usuario. Por eso, utilizar herramientas que posibiliten esta capacidad ha sido decisivo.

- Trabajar en un proyecto real. Crear un paquete para la docencia es un proyecto que tiene una finalidad clara, ayudar a formar alumnos. Además, va a tener aplicación y va a ser utilizada tanto por alumnos como profesores por lo que el trabajo realizado va a ser útil.

- Servirá de referencia para otros TFG. Este proyecto tiene una estrecha relación con la rama de ciencias de la computación y hasta el momento, han sido pocos los alumnos que han desarrollado su TFG fuera del área de conocimiento de la Ingeniería del Software.

- La afinidad con los directores de proyecto ha sido clave para elegir el trabajo de fin de grado.

1.6 EL LENGUAJE DE PROGRAMACIÓN R

Dado que el software va a ser desarrollado en R y el carácter del mismo es eminentemente científico, su diseño se aleja del esquema habitual estudiado en ingeniería del software, por este motivo, en esta sección se tratan conceptos relacionados con la programación y la filosofía de R para el correcto entendimiento de los diferentes conceptos de la memoria.

R nació en 1993 por Ross Ihaka y Robert Gentleman en la universidad de *Auckland*, Nueva Zelanda; como una combinación del lenguaje de programación S creado por John Chambers en 1976. Actualmente es desarrollado y mantenido por *R Development Core Team* del cual Chambers es miembro.

R es un lenguaje de programación funcional e interpretado. Las tareas que se pretenden hacer se realizan mediante funciones. El estilo de programación deriva del concepto teórico programación funcional o *functional programming* que restringe a los métodos a obtener resultados consistentes o al menos, resultados que puedan ser ampliamente verificados. Sencillamente, este tipo de programación ayuda a que los resultados obtenidos sean los esperados antes de implementar la función.

Un complemento natural del lenguaje funcional son las clases y los métodos. Las clases manifiestan la naturaleza de los objetos transmitidos a través de las funciones. Cuando se crea una nueva clase, se busca la razón de la misma que se describe mediante los *slot* –comparable a los atributos en lenguaje orientado a objetos–. La relación entre clases, se representa a través de funciones genéricas propias del lenguaje R, un ejemplo de esta función es *plot* (ver Figura 1), con ella se pueden representar en forma de gráfico cualquier tipo de dato siempre y cuando este se ajuste a las especificaciones del método. Además, la clase debe implementar los métodos que instanciarán los objetos pertenecientes a esa clase y el método que validará los *slot*, asegurando la consistencia de los valores en todo momento.

Los métodos en R deben encapsular claramente una funcionalidad referente a la tarea que se quiere realizar. Los métodos bien definidos extienden de las funcionalidades genéricas del lenguaje, esto hace que los métodos enriquezcan todas las funcionalidades, no solo las de la clase subyacente. Esto no ocurre en lenguajes de programación orientada objetos como Java o C++. En R, los métodos son parte de una amplia a la vez que compleja estructura funcional basada en objetos.

Como se muestra en la figura 1, en R el código para llamar a la función genérica *plot* es accesible para cualquier objeto, en este caso, un vector. Sin embargo, desde Java, imaginando que la función *plot* está definida en la clase Dibujo y siendo esta pública, es necesario instanciar un objeto de esta clase para después llamar a este método. Además, el método plot de Java únicamente está accesible desde la

```r
plot(c(1,2,3,4,5))
```

a: R

```java
Dibujo d = new Dibujo();
d.plot();
```

b: Java

Figura 1: Ejemplo de funciones genéricas.

clase Dibujo mientras que en R cualquier objeto puede llamar a la función *plot*. En R las funciones tienen un alcance global mientras que en los lenguajes de programación orientados a objetos, las funciones tienen un ámbito de clase. No obstante, no todos los objetos de R pueden hacer uso de la función *plot* ya que esta tiene validaciones y solo funciona con ciertos tipos de datos. Por otro lado, se entiende como buena práctica sobrecargar este método en las clases definidas por el usuario en caso de tener un significado análogo; de esta forma se mantiene la misma estructura funcional de R.

R está diseñado para trabajar con datos que están relacionados entre sí independientemente del tipo. Por ello, R propone un tipo de dato muy característico que son los *data frame*. Este tipo de dato tiene una estrecha similitud con las tablas del lenguaje SQL por ejemplo. En lenguajes de programación como Java o C++, no es posible tener en una misma estructura de datos elementos de diferente tipo; por ejemplo en listas, pilas o colas. En el caso de las tablas SQL diferentes tipos de datos pueden coexistir. Esto es lo que ofrece R, una estructura donde poder almacenar estos datos de diferente tipo pero relacionados entre sí.

Una de las características más importantes que ofrece R es la vectorización de operaciones en el procesador, es decir, la posibilidad de aplicar operaciones simultáneas a vectores enteros y no limitarse a una ejecución secuencial de los datos a elementos simples. Esto hace que los cálculos con vectores o matrices sean mucho más eficientes que en los lenguajes de programación convencionales.

Por otro lado, al ser un software libre está ampliamente respaldado por la comunidad y permite crear paquetes o librerías adicionales como esta y distribuirla a terceros [Chambers, 2008].

2 ALGORITMOS DE ESTIMACIÓN DE DISTRIBUCIONES

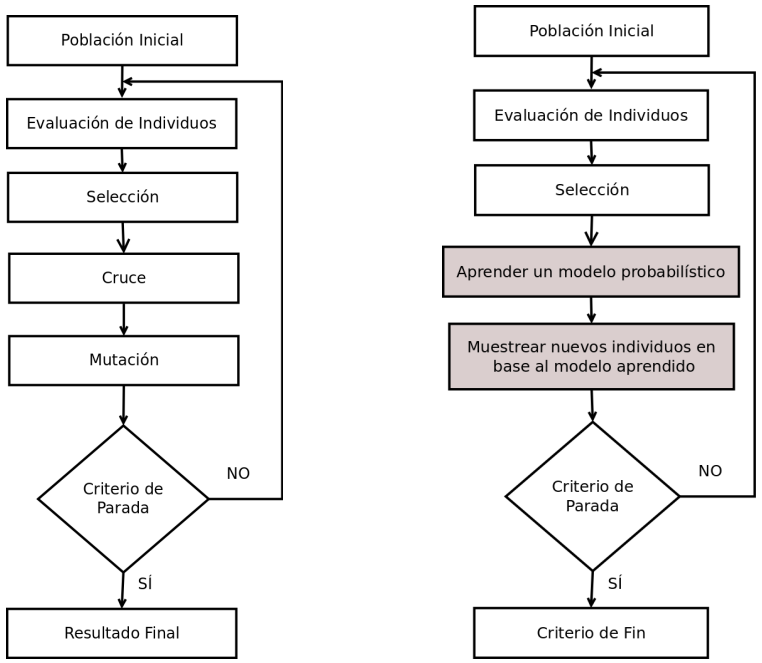
En este capítulo se presenta el fundamento teórico de los algoritmos de estimación de distribuciones que se van a incorporar al paquete *metaehuR* como fruto del trabajo realizado en este TFG.

Este TFG se centra en un conjunto de algoritmos metaheurísticos llamados Algoritmos de Estimación de Distribuciones cuyo acrónimo en inglés es *Estimation of Distribution Algorithms* (EDA) [Pedro Larrañaga y Lozano, 2002; Lozano, 2006; Pelikan, Goldberg y col., 2000; Pelikan, Sastry y col., 2007] que pertenecen a la familia de algoritmos evolutivos (EAs). La principal característica de los EAs es que utilizan técnicas inspiradas en la evolución natural de las especies. En la naturaleza, las especies cambian a lo largo del tiempo adaptándose a nuevas condiciones y entornos. Esta evolución hace que sobreviva el individuo que mejor se haya adaptado. Esta misma idea se traslada al campo de la optimización donde un individuo representa una solución para un problema; una población es un conjunto de soluciones (individuos) y, operaciones como cruce, mutación y selección son utilizadas para que los individuos (soluciones) evolucionen (mejoren). El paradigma más conocido de los EAs son los Algoritmos Genéticos (GAs) [Goldberg, 2000].

A continuación, en la Figura 2 se introduce un esquema comparativo de los GAs y los EDAs. Como mejora de los GAs, se introdujeron los EDAs en el campo de los algoritmos evolutivos. A cada generación, los EDAs aprenden un modelo probabilístico a partir de la población con el fin de representar la relación entre las características (*features*) más prometedoras de las soluciones. Muestreando el modelo probabilístico aprendido en la anterior generación, se generan nuevas soluciones. El algoritmo se detiene cuando cierto criterio de parada se cumple y devuelve la mejor solución encontrada hasta el momento.

Como se puede ver en la Figura 2 los pasos de un algoritmo genético son: evaluar los individuos para tener una medida de lo buenos que son *(fitness)*, ordenar los resultados de mayor a menor calidad *(fitness)*, seleccionar los mejores individuos. Después se procede al cruce entre individuos, habiendo varias técnicas de cruce en este aspecto. A continuación, se introduce, con cierta probabilidad, una mutación a cada nueva solución obtenida. Repitiendo estos pasos, se obtiene una nueva población.

En un EDA la nueva población no es generada mediante el cruce y la mutación de individuos sino que son muestreados de la distribución de probabilidad que se aprende de la población anterior. De este



(a) Esquema genérico de un algoritmo genético

(b) Esquema genérico de un EDA

Figura 2: Esquemas genéricos de las etapas de un GA y un EDA.

modo, se consigue que la relación entre los individuos quede modelada mediante la probabilidad asociada a las soluciones de cada generación.

En realidad, la estimación del modelo probabilístico asociado a cada generación es la tarea más complicada y decisiva de un EDA y por tanto, es lo que determinará el funcionamiento y la calidad de los resultados obtenidos.

A continuación, se presenta el pseudocódigo genérico de un EDA. Pseudocódigo 1 [Bengoetxea y col., 2002].

Algorithm 1: Pseudocode for EDA approach.

1 $D_0 \leftarrow$ Generate N individuals (the initial population) randomly.
2 **Repeat for** $l = 1, 2, \ldots,$ *until a stopping criterion is met* **do**
3 $\quad DS_{e\,l-1} \leftarrow$ Select $S_e \leqslant N$ individuals from D_{l-1} according to a selection method.
4 $\quad p_l(x) = p(x|D^{S_e}_{l-1} \leftarrow)$ Estimate the probability distribution of an individual being among the selected individuals.
5 $\quad D_l \leftarrow$ Sample N individuals (the new population) from $p_l(x)$
6 **end**

En los siguientes apartados se introduce el fundamento teórico de los tres EDAs que se implementan en este proyecto. Además, se incluye una descripción detallada de las etapas de aprendizaje y muestreo de cada uno de ellos.

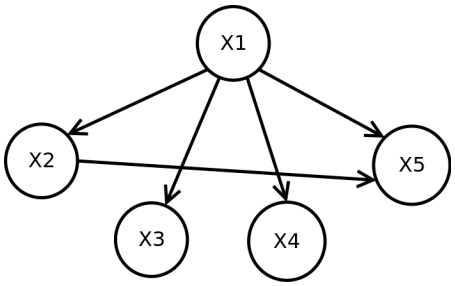


Figura 3: Ejemplo de red Bayesiana.

1. Estimation of Bayesian Network Algorithm (EBNA) [Etxeberria y Pedro Larrañaga, 1999]. Este modelo está sujeto al dominio de permutaciones y aprende una red Bayesiana como modelo probabilístico. Para muestrear este modelo, se sigue el método llamado muestreo lógico probabilístico (PLS). Por último, hay que destacar que el EBNA está sujeto al dominio binario con variables categóricas.

2. Edge Histogram Based Sampling Algorithm (EHBSA) [Tsutsui, 2002]. Este modelo se caracteriza por aprender una matriz de probabilidades marginales de orden 2. En la etapa de muestreo, se obtienen las nuevas soluciones mediante el método *roullete-wheel*.

3. Plackett-Luce Estimation of Ditribution Algorithm (PLEDA) [Josu. Ceberio y col., 2013]. Este algoritmo se caracteriza por implementar un modelo definido sobre el espacio de permutaciones. El aprendizaje consiste en extraer un vector de proporciones de la población dada mediante el algoritmo *Minorization-Maximization* (MM). En la etapa de muestreo se obtienen nuevas soluciones mediante el método *roullete-wheel*.

2.1 ESTIMATION OF BAYESIAN NETWORK ALGORITHM

El algoritmo de *Estimation of Bayesian Network Algorithm* (EBNA) es un EDA que a cada iteración, a partir de la población de individuos aprende una red Bayesiana. Después, se generan las nuevas soluciones muestreando la red. Una red Bayesiana es un modelo gráfico probabilístico en el cual los nodos representan las variables y los arcos las dependencias entre sí (ver Figura 3).

2.1.1 Aprendizaje

Dado un conjunto de individuos, aprender la red Bayesiana que mejor se ajusta a los datos es un problema de optimización *NP-hard*, es decir, la tarea no puede ser resuelta por un algoritmo en tiempo polinomial [Leeuwen, 1990 y Chickering y col., 1994]. Pese a que la comunidad científica ha desarrollado algoritmos que tratan de obtener la estructura que mejor se ajusta a los datos [Etxeberria, Pedro Larrañaga y Picaza, 1997 y P Larrañaga y col., 1996], estos requieren un coste computacional elevado. En consecuencia, en su lugar, se ha decidido utilizar un algoritmo de búsqueda local; en concreto, el *Hill Climbing*. Cabe decir que el éxito de la búsqueda local esta sujeto a la estructura inicial de la que partirá este algoritmo, sin embargo, se ha demostrado [Pedro Larrañaga, Poza y col., 1996] que funcionan bastante bien cuando la estructura inicial es adecuada.

El aprendizaje del modelo requiere aprender la estructura y los parámetros máximo verosímiles de la red Bayesiana. Las medidas más utilizadas que determinan la calidad de las estructuras son el *Akaike's Information Criterion* (AIC) [Akaike, 1974] y el *Bayesian Information Criterion* (BIC)[1] [Chickering y col., 1994] que implementa el algoritmo *Hill Climbing Algorithm*. En este proyecto, se hará uso de la medida BIC. El *Hill Climbing Algorithm* trata de buscar una solución mejor a la actual a cada iteración. En caso de no encontrarla, propone la actual como mejor aproximación. Además, las estructuras son penalizadas en relación a su complejidad, de esta forma, no solo se obtiene una estructura buena sino que además, se consigue la red más simple de entre todas las de igual calidad.

Este algoritmo, dado un conjunto de soluciones D de tamaño N, donde $D = \{x_1, \ldots, x_N\}$, la medida de calidad de cada posible estructura de la red es obtenida calculando el estimador de máxima verosimilitud $\hat{\theta}$ para los parámetros θ y el logaritmo de máxima verosimilitud asociado, $\log P(D|S, \hat{\theta})$. La principal idea del EBNA es buscar la estructura gráfica probabilística que maximice $\log P(D|S, \theta)$. Esto se consigue midiendo la calidad de cada estructura fijándose en el logaritmo de máxima verosimilitud asociado a cada una de ellas [Bengoetxea y col., 2002].

2.1.2 Muestreo

El siguiente paso consiste en muestrear nuevas soluciones. En el caso del EBNA, es necesario muestrear una red Bayesiana. El muestreo de una red Bayesiana puede realizarse siguiendo el *Probabilistic Logic Sampling* (PLS). Este método garantiza que los individuos de la red son muestreados en orden ancestral, es decir, los padres, uno o varios, serán instanciados antes que los hijos. Esto supone ordenar los

1 También conocido como Jeffreys-Schwarz criterion.

individuos. Se llamarán $\boldsymbol{\pi} = (\pi(1), \ldots, \pi(|V_D|))$ a todas los posibles ordenaciones que cumplan que los padres preceden a los hijos. Cuando los valores del padre pa_i de X_i hayan sido asignados, se simularán los valores de X_i siguiendo la distribución $p(x_i|\boldsymbol{pa}_i)$.

2.2 EDGE HISTOGRAM BASED SAMPLING ALGORITHM

El *Edge Histogram Based Sampling Algorithm* (EHBSA) es un EDA que se caracteriza por aprender una matriz de probabilidades de orden dos (adyacencias) de la población [Tsutsui, 2002; Tsutsui y col., 2003]. La población está formada por individuos que pertenecen al dominio de permutaciones. Después, se muestrean nuevas soluciones teniendo en cuenta dicha matriz para obtener nuevas soluciones.

Cabe decir que este tipo de EDA se propuso inicialmente para problemas definidos sobre el dominio de permutaciones, sin embargo, es perfectamente aplicable a cualquier otro problema de optimización combinatoria.

2.2.1 Aprendizaje

Dada una población de N individuos descritos como $D^t = \{\pi_0^t, \pi_1^t, \ldots, \pi_{N-1}^t\}$ a cada iteración t, el EHBSA aprende una matriz $E = \left[e_{i,j}\right]_{n\times n}$ de probabilidades marginales de orden dos en la que $e_{i,j}$ representa el ratio de veces que el elemento i precede de forma adyacente al elemento j en la solución π_k, donde $e_{i,j}$ se define con la siguiente expresión.

$$e_{i,j}^t = \begin{cases} \sum_{k=1}^{N} (\delta_{i,j}(\pi_k^t) + \delta_{j,i}(\pi_k^t)) + \epsilon = 1 & \text{if } i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Donde N es el tamaño de la población y L corresponde al tamaño de cada individuo, $\delta_{i,j}(\pi_k^t)$ es la función indicadora bajo la siguiente expresión.

$$\delta_{i,j}(\pi_k^t) = \begin{cases} 1 \text{ if } \exists h[h \in \{0,1,\ldots,L-1\} \wedge \pi_k^t((h+1) \bmod L) = j)] \\ 0 \text{ otherwise} \end{cases} \tag{2}$$

Por último, ϵ corresponde al parámetro de control definido como

$$\epsilon = \frac{2N}{L-1} B_{ratio} \tag{3}$$

Por tanto, el modelo probabilístico aprendido en un EHBSA corresponde a la matriz de adyacencias de la población.

2.2.2 Muestreo

[Tsutsui, 2002] propuso dos formas alternativas de muestrear, sin plantilla y con plantilla. Esto hace que el algoritmo tome nombres diferentes. En caso de muestreo sin plantilla el algoritmo toma el nombre de *Edge Histogram Based Sampling Algorithm Without Template* (EHBSA/WO). Este obtiene los nuevos individuos siguiendo una técnica de la ruleta sesgada (*roulette-wheel*). En cuanto al método de muestreo con plantilla *Edge Histogram Based Sampling Algorithm With Template* (EHBSA/WT), obtiene las nuevas soluciones utilizando ciertas restricciones para el Algoritmo *roulette-wheel* como se explican en las siguientes secciones.

2.2.3 EHBSA Without Template

En el EHBSA/WO se genera un nuevo individuo $\pi[]$ como se muestra en el Algoritmo 2, siendo N el tamaño de la población:

Algorithm 2: Pseudocódigo general para muestrear el modelo EHBSA.

1 $p \leftarrow 0$.
2 $\pi[0] \leftarrow \mathrm{random}(0, N-1)$; // Obtener el primer elemento de forma aleatoria.
3 **while** $p < N-1$ **do**
4 $\quad rw[j] \leftarrow e^{t}_{s[p],j} (j = 0, 1, \ldots, N-1)$
5 $\quad rw[](rw[c[i]] \leftarrow 0$ for $(i = 0, \ldots, p)$
6 $\quad \pi[p+1] \leftarrow rw[x] / \sum_{j=0}^{N-1} rw[j]$
7 $\quad p \leftarrow p+1$
8 **end**
9 **return** s

Véase que en el EHBSA/WO solo es aplicable al dominio de permutaciones ya que una permutación se caracteriza por que un elemento únicamente aparece una vez en una secuencia. Por ello, el valor de un elemento solo tiene una referencia.

2.2.4 EHBSA With Template

En el EHBSA/WT se trata de estructurar el muestreo mediante segmentos para poder manipular la obtención de nuevas soluciones como se ilustra en la Figura 4. Para obtener un nuevo individuo, se escoge como plantilla una solución –normalmente de forma aleatoria– . Los n puntos de corte $(n > 1)$ son utilizados para dividir dicha plantilla dividiéndolo en n segmentos.

Para obtener un nuevo individuo, se obtienen los elementos acorde a los segmentos de la plantilla controlando así la etapa de muestreo.

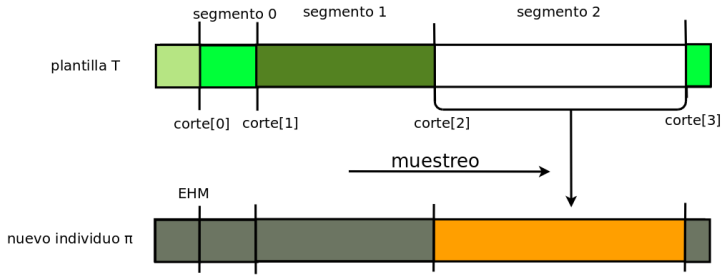


Figura 4: Ejemplo de plantilla del modelo EHBSA/WT.

2.3 PLACKETT-LUCE ESTIMATION OF DISTRIBUTION ALGORITHM

El modelo Plackett Luce es una generalización del propuesto por *Bradley-Terry* [Bradley y Terry, 1952]. El modelo *Bradley-Terry* se planteó para el dominio de permutaciones y se caracteriza por ser un modelo probabilístico gradual que descompone el proceso de obtener un nuevo individuo en n elementos en n etapas secuenciales. Este modelo esta sujeto a la expresión 4.

$$P(a \prec b) = \frac{w_a}{w_a + w_b} \tag{4}$$

donde w_a y w_b son pesos con valores positivos asociados a los elementos a y b. Nótese que cuando el valor w_a es más alto que w_b denota que el elemento a tiene mayor probabilidad que el elemento b.

El algoritmo *Plackett-Luce Estimation of Distribution Algorithm* (PLEDA) fue propuesto por [Josu. Ceberio y col., 2013] y fue diseñado para resolver problemas en el dominio de las permutaciones. Este algoritmo incorpora el modelo *Plackett-Luce*. Por tanto, los parámetros de cada elemento se especifican por un vector de parámetros $\mathbf{w} = (w_1, w_2, \ldots, w_n)$.

Formalmente, para cualquier elemento $i \in B$, siendo B el rango de todos los posibles conjuntos no nulos de $\{1, \ldots, n\}$, $P_B(i)$ es la probabilidad de que el elemento i sea elegido antes que los demás elementos posibles en B y se formaliza mediante la siguiente ecuación:

$$P_B(i) = \frac{w_i}{\sum_{j \in B} w_j} \tag{5}$$

Para explicar de forma sencilla el modelo *Plackett-Luce*, se pide imaginar una urna con infinitas bolas de colores. Considérese realizar un experimento que consiste en sacar bolas de la urna descrita anteriormente. El vector de parámetros $w = (w_1, w_2, \ldots, w_n)$ denota la proporción de bolas que hay de cada color. En la primera fase se extrae una bola de la urna, $\sigma(1)$. La probabilidad asociada a sacar esa bola es

posición	1	2	3	4	5
valor	0,3	0,1	0,02	0,5	0,08

Tabla 1: Vector de pesos estimado del modelo Plackett-Luce.

$$\frac{w_{\sigma(1)}}{\sum_{j=1}^{n} w_{\sigma(j)}} \tag{6}$$

En la segunda fase, se obtiene otra bola de la urna $\sigma(2)$, sin embargo, se descartarán todas aquellas que sean del mismo color que la obtenida en la fase anterior. Por tanto, la probabilidad asociada en esta ocasión es

$$\frac{w_{\sigma(2)}}{\sum_{j=2}^{n} w_{\sigma(j)}} \tag{7}$$

El proceso se repite hasta conseguir una permutación completa σ de todas las bolas de colores. La probabilidad de obtener la permutación σ es el producto de las probabilidades de las elecciones a cada paso:

$$P(\sigma) = \frac{w_1}{w_1 + w_2 + \ldots + w_n} \cdot \frac{w_2}{w_2 + \ldots + w_n} \cdot \ldots \cdot \frac{w_n}{w_n} \tag{8}$$

De forma más cerrada, la probabilidad de σ dado un vector de pesos se describe con la siguiente ecuación.

$$P(\sigma|\mathbf{w}) = \prod_{i=1}^{n-1} \frac{w_{\sigma(i)}}{\sum_{j=i}^{n} w_{\sigma(j)}} \tag{9}$$

2.3.1 Aprendizaje

El aprendizaje del modelo Plackett-Luce consiste en aprender los pesos w dado un conjunto de soluciones. En la literatura se han realizado varias técnicas como el *minorization Maximization* (MM) [Hunter, 2004] o el *Power-Expectation Propagation* (*Power-EP*) [Guiver y Snelson, 2009]. Pese a haber otros métodos para aprender el modelo probabilístico asociado al PLEDA, en este proyecto se ha optado por implementar el MM siguiendo las indicaciones de [Hunter, 2004] y [Josu. Ceberio y col., 2013]. Tras aplicar el algoritmo de aprendizaje, se obtiene como resultado un vector de pesos w que representa las proporciones de los valores posibles.

2.3.2 Muestreo

El muestreo es la etapa en la que se obtienen nuevas soluciones muestreando el modelo probabilístico aprendido como el de la Tabla 1. En este caso, se realiza mediante la técnica de *roulette-wheel*.

Teniendo un vector de pesos w como el de la Tabla 1, se obtiene un valor aleatorio α entre 0 y la suma de todos los valores del vector v. Se busca el menor índice j de la posición que cumpla que $\alpha >= \sum_{i=1}^{j} w(i)$. El índice se selecciona como elemento del nuevo individuo. Después se anula esa posición estableciendo su valor a 0. Se repite el proceso hasta obtener un individuo completo.

3 PLANIFICACIÓN

En este capítulo se exponen los distintos aspectos relacionados con la gestión del proyecto y la planificación. Se describirá el objetivo principal y la arquitectura planteada. Además, se introducen conceptos como el alcance del proyecto mediante la definición del ciclo de vida y de un diagrama de Estructura de Descomposición del Trabajo (EDT). Se incorpora la valoración económica y los riesgos posibles. También se presentará la planificación temporal mediante un diagrama de *Gantt* y las herramientas utilizadas en la elaboración del proyecto.

3.1 OBJETIVOS

El principal objetivo del proyecto es desarrollar una parte del paquete *metaheuR*, en concreto la ampliación de los algoritmos de Estimación de Distribuciones. Para ello, se implementa el aprendizaje y el muestreo de *Estimation of Bayesian Network Algorithm*, *Edge Histogram Based Sampling Algorithm* y *Plackett Luce Estimation of Distribution Algorithm* ya que el *framework* de los EDAs está ya presente en el paquete.

Por otro lado, se implementarán funciones que calculen la probabilidad de los individuos así como los métodos que ayudarán a verificar el correcto funcionamiento de los modelos desarrollados.

Para llevar a cabo este desarrollo se mantendrá principalmente un enfoque de programación científica en el cual se potencia la optimización del código, así como los recursos que este utilizan.

3.2 ARQUITECTURA

La arquitectura que se ha utilizado en este proyecto sigue los estándares del modelo *pipeline*. Se comienza con el aprendizaje y la obtención de conocimientos por parte del ingeniero. A continuación, se desarrolla el EBNA y se hacen los experimentos pertinentes para comprobar su correcto funcionamiento. Después, se desarrollan el EHBSA y, por último, el PLEDA sobre las mismas premisas. Para finalizar, se hacen ejecuciones de todos los EDAs y se analizan los resultados. En la Figura 5 se muestra de forma gráfica el proceso que se ha llevado a cabo.

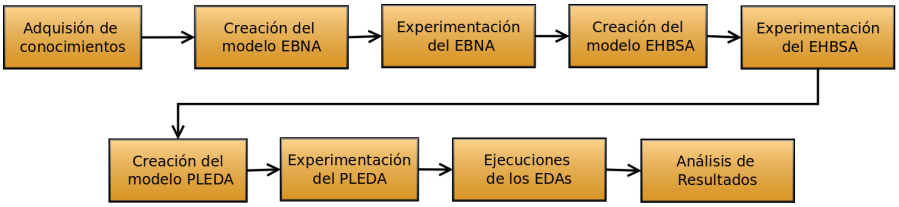


Figura 5: Arquitectura del proyecto.

3.3 ALCANCE

Ciclo de vida. Para la correcta planificación del proyecto se elige la metodología *agile*[1]. Se divide la duración total del proyecto en diferentes fases o *sprints* de 15 días cada uno, acorde a las reuniones de seguimiento que se hacen con los directores del proyecto. En estas reuniones, después de cada *sprint* se revisan las tareas pendientes así como las finalizadas. En caso de haber un cambio de requisitos, se reestima la planificación. Por tanto, se establece una hoja de ruta a seguir durante el próximo *sprint*.

Es importante recalcar que a cada finalización de un *sprint* tenemos una versión útil del proyecto, siguiendo así la metodología. Sin embargo, esto no se aplica a la fase de documentación del proyecto.

Estructura de Descomposición del Trabajo (EDT). La Figura 6 muestra el diagrama EDT seguido durante el proyecto. Esta estructura jerarquizada plasma los diferentes módulos del proyecto y ofrece una visión general de todo el trabajo a realizar. Además, se especifica el responsable del desarrollo, la duración en horas, la descripción, los requisitos previos y el resultado.

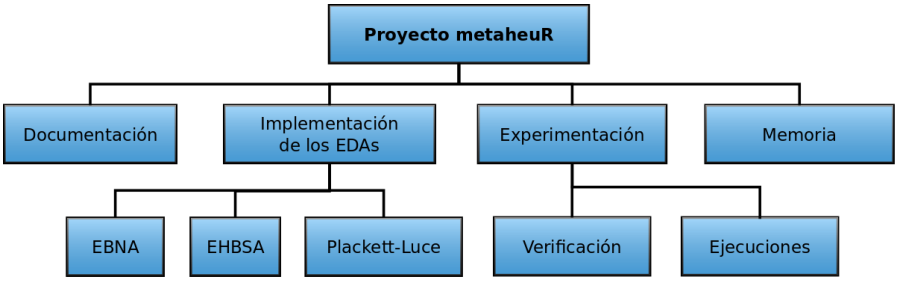


Figura 6: Diagrama EDT.

1. **Documentación**.
   Responsable: Ander Carreño López.
   Duración: 35 horas.
   Descripción: Documentarse y aprender temas relacionados con optimización, métodos heurísticos, algoritmos evolutivos, los

1 La metodología *agile* trata de hacer una mejora continua en la parte de requerimientos y programación. Haciendo entregas continuadas e incrementales. Referencia: https://en.wikipedia.org/wiki/Agile_software_development

EDAs que se van a implementar e informarse sobre lo desarrollado hasta el momento por la comunidad científica. Además, se aprenderá a programar en el lenguaje de programación R.
Entradas: Artículos publicados en revistas o en congresos internacionales, tesis doctorales y libros.
Salidas/Entregables: Resúmenes y apuntes que sirven de referencia al ingeniero durante el desarrollo del proyecto. Estos son revisados por los directores del proyecto.
Recursos necesarios: Un ordenador con LaTeX, conexión a Internet y acceso a la VPN de la escuela para acceder a los artículos del portal http://www.springer.com/gp/.
Precedencias: Ninguna.

2. **Implementación de los EDAs.**
Responsable: Ander Carreño López.
Duración: 275 horas.
Descripción: Diseño y programación de todos los elementos del software mediante la realización de casos de uso extendidos y diagramas de secuencia. Así como plantear la integración en el paquete *metaheuR*.
Entradas: Paquete *metaheuR* parcialmente creada.
Salidas/Entregables: Software completo.
Recursos necesarios: IDE de R (*Rstudio*), paquete *metaheuR*, *GitHub* y conexión a Internet.
Precedencias: Librería *metaheuR*.

   a) **Implementación de los EDAs. EBNA.**
   Responsable: Ander Carreño López.
   Duración: 96,25 horas.
   Descripción: Diseñar la estructura de la clase así como la implementación de los métodos del algoritmo para introducirlos en el paquete *metaheuR*. Se busca en el paquete *bnLearn* [Scutari, 2009] los métodos necesarios para hacer un aprendizaje y un muestreo de la red Bayesiana aprendida. Además se busca la conexión con el paquete *metaheuR* para hacer una integración completa.
   Entradas: Paquete *bnLearn*.
   Salidas/Entregables: Algoritmo completo.
   Recursos necesarios: Paquete *bnLearn* y la población inicial.
   Precedencias: Ninguna.

   b) **Implementación de los EDAs. EHBSA.**
   Responsable: Ander Carreño López.
   Duración: 55 horas.
   Descripción: Diseñar e implementar el algoritmo EHBSA en R partiendo de una población de soluciones.
   Entradas: Población inicial obtenida del paquete *metaheuR*.

Salidas/Entregables: Versión final del algoritmo solicitado.
Recursos necesarios: Población inicial.
Precedencias: Ninguna.

c) **Implementación de los EDAs. PLEDA.**
Responsable: Ander Carreño López.
Duración: 123,75 horas.
Descripción: Implementar el aprendizaje y el muestreo del algoritmo PLEDA en el paquete *metaheuR*.
Entradas: Población inicial.
Salidas/Entregables: Algoritmo completo.
Recursos necesarios: Población inicial.
Precedencias: Ninguna.

3. **Experimentación.**
Responsable: Ander Carreño López.
Duración: 40 horas.
Descripción: Se realizan pruebas para comprobar tanto el funcionamiento como la calidad de los algoritmos. Las pruebas consistirán en comprobar el apredizaje y el muestreo de cada uno de ellos, para ello se utilizarán técnicas para medir la divergencia entre otros. Para ver la calidad de los mismos, se obtendrán resultados a los problemas de optimización clásicos como el *Travelling Salesman Problem* y el *Knapsack Problem*.
Entradas: Los algoritmos implementados.
Salidas/Entregables: Gráficas, divergencia, datos y tablas de resultados válidas para sacar conclusiones.
Recursos necesarios: Ninguno.
Precedencias: Documentación y software en estado avanzado.

   a) **Experimentación. Comprobación de funcionamiento.**
   Responsable: Ander Carreño López.
   Duración: 32 horas.
   Descripción: Se realizan pruebas para garantizar el muestreo y el aprendizaje de los modelos, para ello, se crean poblaciones iniciales no aleatorias para obtener así un modelo concreto conocido a priori entre otros. Para la etapa de muestreo, se muestrean modelos específicos con alta probabilidad de obtener cierto individuo.
   Entradas: Los algoritmos implementados.
   Salidas/Entregables: Gráficas, divergencia, datos y tablas de resultados válidas para sacar conclusiones.
   Recursos necesarios: Ninguno.
   Precedencias: Documentación y software en estado avanzado.

   b) **Experimentación. Pruebas de ejecución.**
   Responsable: Ander Carreño López.
   Duración: 8 horas.

Descripción: Una vez comprobado el funcionamiento de los algoritmos, se ejecutan EDAs y se crean comparativas entre ellos. De esta forma, se ponen en contraposición todos los modelos implementados y se analizan los resultados.
Entradas: Los algoritmos implementados.
Salidas/Entregables: Gráficas, datos y tablas de resultados válidas para sacar conclusiones.
Recursos necesarios: Ninguno.
Precedencias: Documentación y software en estado avanzado.

4. **Memoria.**
Responsable: Ander Carreño López.
Duración: 85 horas.
Descripción: Redacción de la memoria y documentación del trabajo realizado.
Entradas: Memorias de años anteriores. Documentos sobre gestión de proyectos y apuntes de asignaturas cursadas como Ingeniería del Software, Análisis y Diseño de Sistemas Inteligentes, Estructuras de Datos y Algoritmos, Gestión de Proyectos, entre otros.
Salidas/Entregables: Memoria final del proyecto.
Recursos necesarios: Ordenador con LaTeX.
Precedencias: Documentación y software en estado avanzado.

3.4 ESPECIFICACIÓN Y DURACIÓN DE TAREAS

En esta sección se describirán las tareas y subtareas realizadas en el proyecto que se puede ver en la Tabla 2. Además, en la Figura 7 se muestra el diagrama de Gantt que ilustra la planificación inicial del proyecto. No obstante, la planificación real ha variado respecto a la inicialmente prevista. Por este motivo, en el Capítulo 8 se ponen en contraposición ambos diagramas.

Fases	Descripción Fase	Tarea	Descripción Tarea
1	DOCUMENTACIÓN	1	Repasar la optimización de problemas y familiarizarse con los algoritmos metaheurísticos, profundizando en los algoritmos de estimación de distribuciones (EDAs)
		2	Familiarizarse con el entorno de programación R y el paquete *metaheuR*
		3	Llevar a cabo la planificación del proyecto
2	IMPLEMENTACIÓN DE LOS EDAs	4	Estudiar las distribuciones de probabilidad sobre el dominio discreto y el dominio de permutaciones
		5	Implementar un EDA basado en redes Bayesianas (EBNA) haciendo uso del paquete *BnLearn*.
		5.1	Estudiar el paquete *BnLearn* para conocer los métodos de inferencia de una red Bayesiana y cómo muestrear nuevas soluciones.
		5.2	Realizar la implementación y la conexión con el paquete *metaheuR*
		6	Implementar el algoritmo EHBSA en R.
		7	Implementar en R la estimación y el muestreo de los parámetros de la distribución de probabilidad Plackett-Luce.
		7.1	Estudiar el modelo probabilístico Plackett-Luce para descubrir la mejor optimización numérica de los parámetros del modelo dada una población.
		7.2	Finalizar la implementación del aprendizaje del Plackett-Luce en mataheuR conectando el código en C.
		7.3	Implementar el muestreo sobre la distribución Plackett-Luce.
3	PRUEBAS METAHEUR	8	Implementación de un problema del dominio de las permutaciones para posteriores pruebas.
		9	Prueba experimental de los EDAs implementados en la fase 2.
		9.1	Comprobar el correcto funcionamiento de las funciones de aprendizaje, muestreo y cálculo de probabilidad.
		9.2	Ejecuciones de los algoritmos sobre problemas de optimización clásicos.
		10	Análisis y visualización de los resultados.
4	ELABORACIÓN DE LA MEMORIA	11	Redacción del documento de memoria.
		12	Preparación de la presentación.

Tabla 2: Descripción de tareas.

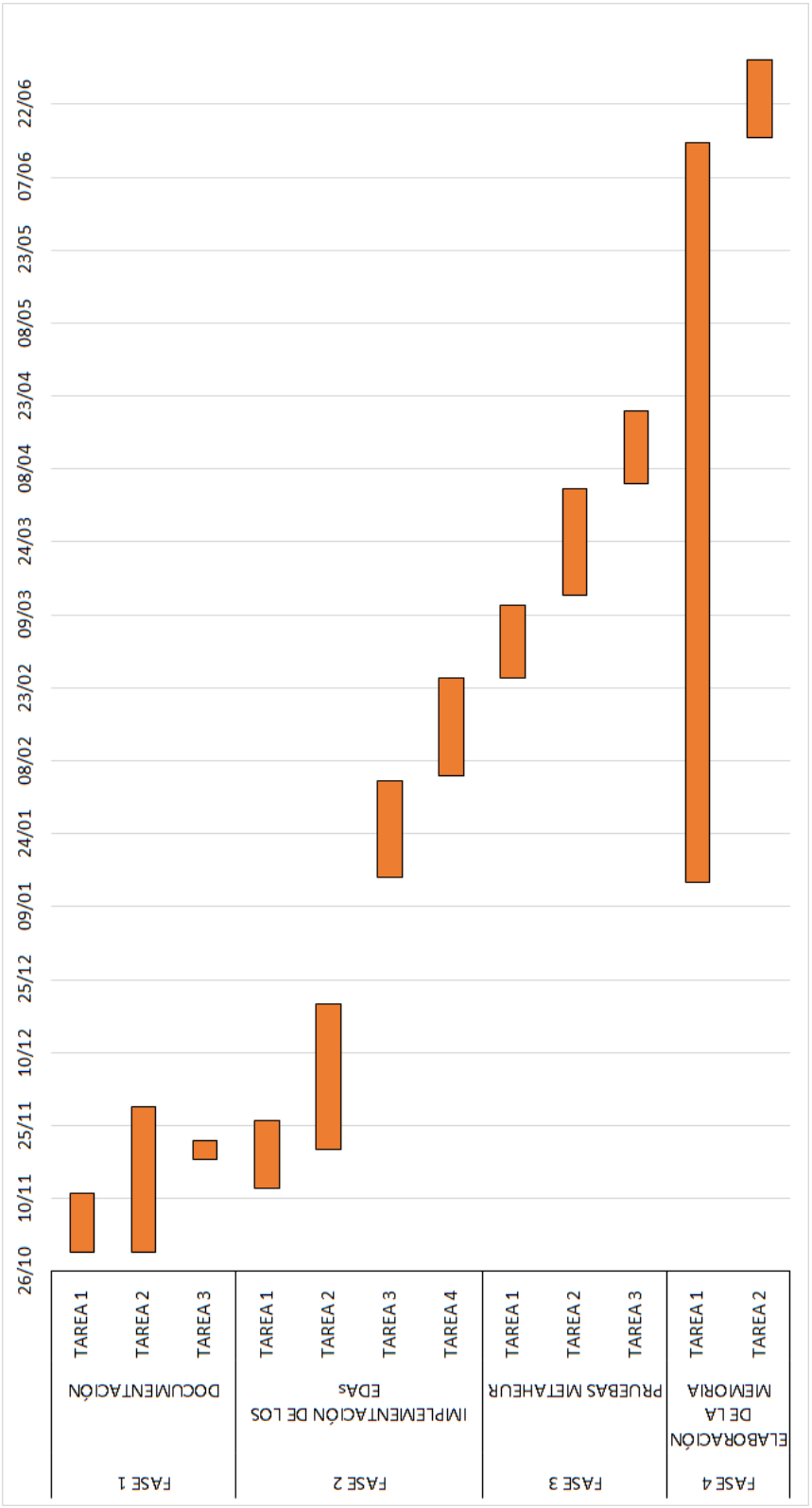


Figura 7: Diagrama de Gantt de la planificación inicial.

3.5 RIESGOS

Durante de la realización del proyecto pueden ocurrir una serie de incidencias, por ello, es importante hacer un recuento de las más comunes y preparar planes de contingencia. De esta forma, se tendrán cubiertos parte de los daños en caso de ocurrir un contratiempo.

3.5.1 Falta de conocimiento de alguna tecnología

Este problema es uno de los más habituales. En ocasiones el ingeniero se encuentra ante la situación de no tener el conocimiento necesario para realizar un determinado problema. Esto se debe a que el ingeniero a priori piensa que es capaz de llevar a cabo el proyecto en su totalidad y por tanto, subestima la complejidad de las tareas.

Subestimar la complejidad de una tarea o sobreestimar las capacidades personales es un grave error que puede comprometer a todo el proyecto.

Plan de prevención

Para prevenir las incidencias arriba descritas se enumeran una serie de medidas a tener en cuenta:

- Tener claras las capacidades del desarrollador. Es muy importante conocer los límites de conocimiento, presión y experiencia. Es vital tener una planificación detallada y realista para no desembocar en posibles callejones sin salida y pérdida de motivación.

- Es necesario que el programador tenga conocimientos previos en el trabajo a abordar, teniendo siempre en cuenta que en el transcurso del mismo va a tener que dedicar tiempo al estudio y aprendizaje de nuevas aptitudes.

- Apoyarse en los tutores del TFG. Es muy importante tener a los directores bien informados de los avances como de los retrasos en las entregas ya que ellos tienen mayor conocimiento sobre la materia, así como experiencia en proyectos de larga duración.

Probabilidad

La probabilidad de aparición de este tipo de contratiempos es muy alta, ya que la materia que desarrolla el trabajo es compleja y científica. No obstante, viene ligada a la experiencia y conocimientos previos del programador.

Impacto

El impacto de estos contratiempos es muy alto debido a que la falta de conocimiento de una determinada herramienta afecta directamente al correcto desarrollo de las tareas.

Una falta de conocimiento puede hacer que se pare el desarrollo durante semanas hasta encontrar una solución. Por tanto, descuadraría la planificación.

Plan de contingencia

Si este riesgo surgiera, en primer lugar se buscaría información en Internet (foros, portales, páginas, etc.). En caso de no ser encontrada la causa del problema y su solución, se procedería a buscar en libros de la biblioteca o preguntaríamos a los tutores del TFG.

En caso de preguntar a los tutores, se explicará claramente el motivo del error y lo que se ha realizado por cuenta propia para poder ser solucionado aunque no se haya tenido éxito.

3.5.2 Desmotivación del ingeniero

Un posible contratiempo en el transcurso del proyecto es la pérdida de motivación del ingeniero. Esto es un problema muy común que afecta directamente a la calidad y el transcurso del TFG.

Puede ocurrir que vaya ligada a los problemas descritos anteriormente.

Plan de prevención

Para poder prevenir las incidencias causadas por la desmotivación del desarrollador, se enumeran a continuación una serie de medidas:

- Conocer los límites del ingeniero. Es muy importante tener claras las posibilidades y las fronteras del desarrollador. Este conocimiento hace referencia a las aptitudes teóricas, límites de presión y a la experiencia previa en las tecnologías utilizadas.
- Conocimientos previos en la materia. Es vital que el desarrollador tenga amplios conocimientos en la materia o que esté dispuesto a adquirlos sabiendo lo que esto supone.
- Apoyo por parte de los directores del proyecto. Es importante que el desarrollador no se sienta solo ante las incertidumbres del proyecto para que no caiga en este tipo de problemas.

Probabilidad

La probabilidad variará en base a los conocimientos previos del alumno. Además, tendrá estrecha relación con las especificaciones

del trabajo presentado, ya que si la explicación del proyecto no representa la complejidad del mismo, es posible que se encuentre ante estas situaciones.

Impacto

En caso de producirse este riesgo, comprometería directamente al proyecto. No se realizarían las entregas a tiempo y se perdería el interés por el mismo. Una falta de motivación puede hacer que el desarrollo se abandone y que no se cumplan los objetivos.

Plan de contingencia

Si por algún motivo ocurre este riesgo, estas serán las medidas que se llevarán a cabo para corregir el problema.

- Comentar con los motivos por los cuales se ha perdido la motivación a los tutores del TFG.
- Intentar cambiar el enfoque del TFG sobre la tarea que está causando el riesgo en concreto y buscar alternativas a estas con los tutores.
- Como último recurso, se cambiaría de tarea y realizar por otra más motivadora.

3.5.3 Enfermedad

Un posible riesgo que existe en todo momento es que el ingeniero caiga enfermo. Esto supondrá un problema ya que puede cambiar la planificación del proyecto hasta que el autor se recupere. Además, un largo periodo sin estrecha relación con el proyecto, puede ser la causa de una pérdida de motivación.

Plan de prevención

Este problema no puede ser previsto ya que depende de factores ajenos a la planificación del proyecto.

Sin embargo, se tomará en cuenta que el programador se vea afectado por alguna enfermedad, por tanto, se añadirán días de holgura en la planificación del TFG.

Probabilidad

La probabilidad de este problema no es predecible. Sin embargo, se debe tener en cuenta la cantidad de veces que se ha caído enfermo en años anteriores para hacer una estimación aproximada.

Impacto

Según el tipo de enfermedad contraído, el impacto puede ser muy grande. Además, al ser una única persona la que desarrolla este trabajo, en muchas ocasiones, este riesgo supondrá la paralización del mismo durante ese periodo de tiempo.

Plan de contingencia

Si ocurriera este riesgo, no podríamos aplicar ningún plan de contingencia. Se sabe que una persona enferma no está en disposición de continuar el desarrollo. Además, al ser un único integrante, no habrá otro compañero que continúe en su ausencia.

3.5.4 Pérdida de información

Si ocurriera un fallo en los soportes de almacenamiento donde se encuentra alojado el TFG, se tendría una perdida notoria de tiempo de trabajo y esfuerzo. Por ello, se toman medidas a priori para minimizar los daños posibles.

Plan de prevención

El fallo del soporte de almacenamiento físico de un ordenador no es posible prevenirlo, la vida útil de estos dispositivos no se puede aumentar. Sin embargo, se pueden tomar medidas para que si ocurre este contratiempo, la pérdida sea la menor posible.

- Tener copias de seguridad en sistemas de almacenamiento en la nube. En este proyecto se utilizará la herramienta *Dropbox* que posibilitará tener el proyecto en la nube.
- Tener una copia en un soporte flash. Se tratará tener siempre una copia del trabajo en un soporte flash para poder recuperar la información.
- Tener la información de código en *GitHub*. En este proyecto también se utilizará la herramienta *GitHub* que ayudará a tener un control de versiones de los datos.

Impacto

El impacto es muy alto. Si el soporte de almacenamiento de nuestro ordenador falla, las herramientas de desarrollo del programador quedarán inservibles por lo que tendrá que volver a instalar y configurar todo el entorno necesario.

En cuanto a los datos, no tendrá inconveniente si se ha seguido un correcto plan de prevención.

Plan de contingencia

En el hipotético caso de que ocurra este problema, se llevarán a cabo las siguientes medidas de recuperación.

- Obtener otro disco duro u ordenador con la mayor brevedad posible para proceder a la configuración del mismo.
- Recuperar los datos de los sistemas planteados en el plan de prevención.
- En caso de no tener una copia de seguridad en ningún otro soporte, intentar obtener los datos del disco duro dañado.

3.5.5 Fallo en la conexión a Internet

Un fallo común es que la conexión a Internet se vea comprometida. Este problema provoca que el desarrollador no tenga acceso a las herramientas alojadas en red, así como a la información que pueda consultar.

Además, hasta la solvencia del error, el programador no tendrá acceso al correo electrónico y, por tanto, no podrá comunicarse con sus tutores.

Plan de prevención

Un método efectivo es utilizar el dispositivo móvil con acceso a Internet para hacer *tethering*, esto posibilitará acceder a la red y continuar con el desarrollo del proyecto.

Probabilidad

La probabilidad es muy alta ya que, las obras o las condiciones climatológicas adversas incrementan la aparición de estos contratiempos.

Impacto

El impacto no será muy alto ya que este riesgo afectará a ciertas tareas del TFG pero no a la totalidad de ellas por lo que, en caso de comprometerse el acceso a la red, el programador podrá dejar esta tarea y realizar otra hasta que se subsane el error. Además, los cortes de Internet suelen tener una duración pequeña normalmente por lo que no afectará en exceso.

Plan de contingencia

El plan a seguir en caso de este tipo de errores es el descrito a continuación

- Cambiar de tarea a una que no necesite conexión a Internet.
- Utilizar el dispositivo móvil como anclaje de red (*Tethering*).

3.5.6 Errores humanos

Durante la realización de cualquier trabajo, tanto directores como el programador pueden cometer errores sin intención. Estos errores pueden comprometer el desarrollo del trabajo ya que pueden desembocar en pérdida de información o en una mala interpretación de requerimientos.

Los descuidos por parte del desarrollador pueden ser: bloquear la sesión del ordenador, borrar carpetas sin intención, etc.

Plan de prevención

Para prevenir posibles errores en la incorrecta interpretación de requerimientos, por ejemplo tras una reunión, se enviarán por email los puntos tratados en dicho encuentro.

Por otro lado, en cuanto a la pérdida de información, el ingeniero deberá tener un control de la seguridad y de integridad de su ordenador.

Probabilidad

La probabilidad de ocurran estos contratiempos es muy baja ya que el desarrollador tiene experiencia sobre este tipo de problemas. Sin embargo, puede que el ingeniero no se haya percatado de la realización de ese error.

Impacto

El impacto es variable ya dependerá en el tipo de daño causado por el improvisto. Siendo grave si por ejemplo, se deja la sesión sin bloquear y un usuario malintencionado accede a nuestro ordenador.

Plan de contingencia

Si el riesgo se produce, se procederá a seguir el siguiente plan:

- Evaluar cuáles han sido los daños.
- Cambio de contraseña de los principales entornos de desarrollo, así como las herramientas.
- Recuperar los datos con copias de seguridad.
- Estudiar las razones para entrar en mejora continua.

3.6 EVALUACIÓN ECONÓMICA

En la realización de este proyecto se tienen una serie de costes asociados. Dichos costes los diferenciaremos entre costes fijos y variables.

Costes Variables		Costes Fijos	
Concepto	Precio (hora)	Concepto	Precio (global)
Luz	0,042 €	Ordenador	79,16 €
Agua	0,035 €	Perifericos PC	22,92 €
Gas	0,035 €	Transporte	30 €
Telefonía	0,028 €		
Internet	0,07 €		
Salario	21 €		
Total / hora	**21,21 €**	**Total / global**	**132,08 €**

Tabla 3: Tabla de costes.

Los costes fijos son estimados por mes. Por lo que 8 meses de proyecto suman 1056, 64 €.

Teniendo en cuenta que la duración del proyecto según la planificación inicial es de 613 horas. El coste total del proyecto asciende a 14058, 37 €.

Además, sumando las horas que dedican los tutores, teniendo en cuenta que siguiendo la metodología *agile* se tienen reuniones tras finalizar el *sprint*, y que las 15 reuniones tienen una duración media de 1, 25h y que el salario de un profesor de universidad es de 30 € por hora, el coste por parte de la universidad es de 562, 5 € por director, por tanto, 1125 €.

En definitiva, los costes asociados a este proyecto son de

$$C_{Variables} + C_{Fijos} + C_{Directores} = 15183,37€$$

3.6.1 Retorno de la Inversión (ROI)

Este proyecto no tiene ánimo de lucro. El código quedará en *GitHub* desde el primer momento y el libro de apoyo será gratuito. No obstante, a fin de realizar el ROI y suponiendo que el libro tiene un coste de 30€ se realiza la siguiente estimación.

Dado que este proyecto tiene un uso académico importante, es decir, va a ser útil a corto plazo. Suponemos que las ganancias por alumno que utilice este paquete es de 30€ anuales.

Por tanto, la asignatura que utilizará este proyecto tendrá una ocupación anual de 25 alumnos. Podemos decir que los ingresos anuales de la herramienta se estiman en 750€.

Lo cual implica que el proyecto será rentable pasados 20 años aproximadamente.

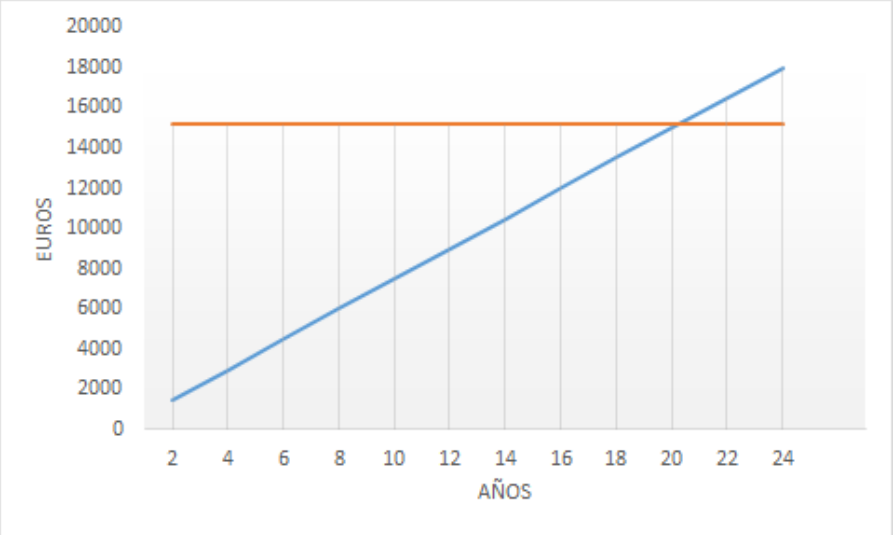


Figura 8: Curva de Retorno de la Inversión.

3.7 HERRAMIENTAS

En esta sección se describen las herramientas utilizadas para la realización de este proyecto:

- RStudio: Entorno de desarrollo para el lenguaje de programación R. Elegido por la cómoda interfaz que nos ofrece y así como por la posibilidad de enlazarlo con *GitHub*.
- Dropbox: Servicio de almacenamiento en la nube. Permite compartir los archivos con una o varias personas. Además ofrece un control de versiones simple.
- GitHub: Plataforma de desarrollo colaborativo que permite alojar proyectos utilizando el sistema de control de versiones *Git*.
- Procesadores de texto LaTeX, en concreto *Gummi* y *TexStudio*.
- Cacoo: Herramienta web utilizada para la creación de diagramas y figuras. Elegida debido a que ofrece la posibilidad de editar diagramas por múltiples usuarios, en tiempo real.
- DIA: Herramienta de software libre utilizada para crear diagramas. Elegida por ser una herramienta muy simple y ligera que permite exportar los gráficos a una gran variedad de formatos.
- Microsoft Excel: Hoja de cálculo que posibilita hacer gráficos.
- Sublime Text: Editor de textos que ayuda a la programación con gran cantidad de *plugins* y además tiene una interfaz muy amena y cómoda.

- Gmail: Herramienta de correo electrónico utilizada para relacionarse de forma directa con los directores.
- Paquetes externos de R utilizados.
  - *bnLearn*: Utilizado para aprender y muestrear una red Bayesiana.
  - *ggplot*: Herramienta para dibujar funciones.
  - *graphviz*: Herramienta para dibujar redes Bayesianas.

4 CAPTURA DE REQUISITOS

En este capítulo se describe la captura de requisitos del proyecto. Así mismo, se presentan los diagramas *UML* correspondientes al diseño del proyecto, siendo estos los diagramas de casos de uso, que ilustran las opciones del usuario sobre las que se ha desarrollado este proyecto.

Los diagramas describen los requisitos para el desarrollo del aprendizaje y el muestreo que incluyen los algoritmos EBNA, EHBSA y PLEDA.

4.1 CASOS DE USO

En esta sección se muestran los diagramas de casos de uso que explican las opciones que puede tomar un usuario cualquiera a la hora de solucionar un problema de optimización con los métodos implementados en este TFG. (Figura 9).

El usuario tendrá la opción de resolver un problema de optimización con uno de los algoritmos implementados. Sin embargo, en el caso del EBNA y el PLEDA el usuario también puede obtener la probabilidad de una población de individuos dado el modelo.

- **EBNA.** Para este algoritmo, al usuario se le presentan tres opciones. En primer lugar, el aprendizaje del modelo gráfico probabilístico; por otro lado, el muestreo de la red Bayesiana aprendida y por último, el cálculo de la probabilidad de los individuos dado un modelo. Para el aprendizaje, el usuario deberá proporcionar la población de individuos inicial. Como resultado, se obtiene el modelo de la clase *BayesianNetwork* cuyo único *slot* contiene el modelo aprendido. La llamada a este método se realiza de la siguiente forma:

```
learned.model <- bayesianNetwork(population)
```

  Para el muestreo, se exige al usuario proporcionar 2 parámetros, por un lado el modelo que se quiere muestrear y por otro lado, el número de individuos que se quieren obtener. El modelo proporcionado debe ser del tipo *BayesianNetwork* y el número de muestras será de tipo *numeric*. Como resultado, se obtiene una lista de individuos. El caso de uso extendido se puede ver en la Tabla 4.

```
new.population <- simulate(learned.model, 1000)
```

Otra opcion interesante que ofrece la clase *BayesianNetwork* es el cálculo de la probabilidad de un individuo dado un modelo. Es decir, una vez calculado un modelo, el usuario tiene la opción de calcular la probabilidad de obtener un individuo en concreto. Este método recibe por un lado el objeto de la clase *BayesianNetwork* y por el otro, la lista de individuos de los cuales se desea conocer la probabilidad (Caso de uso extendido en Tabla 5).

```
probabilities <- calc.prob(learned.model, population)
```

- **EHBSA.** Esta clase posibilita al usuario aprender y muestrear el modelo probabilístico de probabilidades marginales de orden dos que implementa el EHBSA.
  Para el aprendizaje del modelo, es necesario proporcionar una población de la cual se aprenderá la matriz de adyacencias que caracteriza al EHBSA. Como resultado, obtenemos un objeto de la clase EHBSA. Esta acción se realiza de la siguiente manera.

```
learned.model <- ehbsa(population)
```

  Para el muestreo del modelo se informarán dos parámetros, el modelo que se quiere muestrear y el número de muestras que se quieren obtener. El modelo debe ser de la clase EHBSA y el número de muestras deberá ser *numeric*. Como resultado, se obtiene una población de los individuos que se han informado. Este es el ejemplo de llamada de la función *simulate*.

```
new.population <- simulate(learned.model, 1000)
```

  El caso de uso extendido se puede ver en la Tabla 6.

- **PlackettLuce EDA.** Este clase proporciona 3 opciones al usuario, en primer lugar posibilita el aprendizaje del modelo probabilístico *Plackett Luce*. Después, ofrece la posibilidad de muestrear dicho modelo y, por último, posibilita el cálculo de la probabilidad asociada a una lista de individuos dado un modelo ya aprendido.
  El aprendizaje del modelo requiere de una población de individuos de la cual aprender el vector de propociones que caracteriza al modelo *Plackett Luce*. Además, como parámetro opcional, se pueden especificar el número máximo de iteraciones que llevará a cabo el algoritmo *MM* encargado de aprender el modelo. En caso de no especificar el número máximo de iteraciones, por defecto está especificado a 10000 iteraciones. Este modelo requiere de una población de individuos con cierta dispersión [Josu. Ceberio y col., 2013 y Hunter, 2004] ya que sino, el algoritmo MM no podrá converger a solución. Es por ello que el algoritmo siempre introduce el reverso del primer individuo de la población proporcionada para aprender. La llamada de este método se realiza de la siguiente forma.

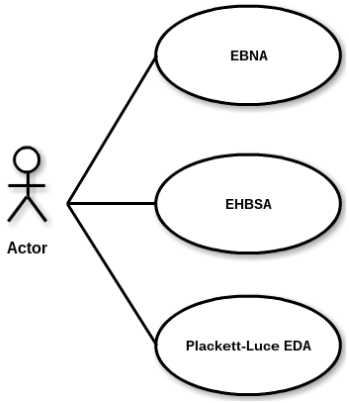


Figura 9: Casos de uso.

```
learned.model <- plackettLuce(population, max.iter=1000)
```

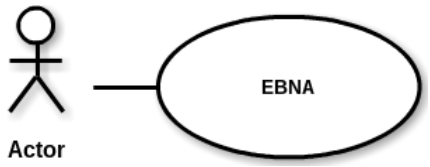
El muestreo del algoritmo se realiza informando dos parámetros, el modelo probabilístico perteneciente a la clase *PlackettLuce* y el número de nuevos individuos a obtener, siendo este parámetro de tipo *numeric*. La llamada a este método se muestra a continuación.

```
new.population <- simulate(learned.model, 1000)
```

El cálculo de probabilidades asociadas a los individuos dado un modelo se realiza con la función *calc.prob*. El caso de uso extendido se puede observar en la Tabla 4. Este método exige 2 parámetros, siendo el primero de ellos un objeto de la clase *PlackettLuce* y el segundo, una lista de individuos de los cuales se quiere obtener la probabilidad. Como resultado, se obtiene un vector con las proporciones de los individuos. Este es el ejemplo de llamada al método.

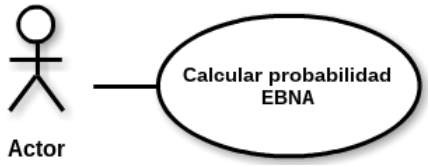
```
probabilities <- calc.prob(learned.model, population)
```

El caso de uso extendido se puede ver en la Tabla 7.



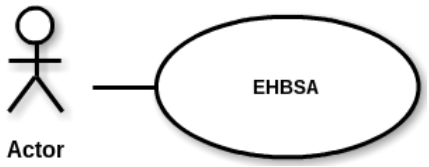
Nombre. Solucionar un problema de optimización mediante el algoritmo EBNA.
Descripción. Obtener la solución a un problema concreto de optimización mediante el algoritmo EBNA.
Actores. Usuario.
Precondiciones. La población inicial del problema debe ser una matriz en la que cada fila sea un individuo.
Postcondiciones. Solución al problema de optimización definido.
Flujo de eventos. 1. El usuario crea una población inicial válida para el problema a optimizar. 2. El usuario llama a la función de optimización con el algoritmo EBNA 3. Se obtiene la solución final.

Tabla 4: Caso de uso extendido. Algoritmo EBNA.



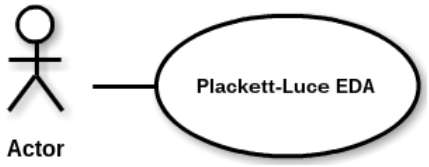
Nombre. Conocer la probabilidad de una población de individuos dado el modelo probabilístico.
Descripción. Obtener la probabilidad de que ese individuo sea muestreado en la distribución de probabilidad aprendida por el EBNA.
Actores. Usuario.
Precondiciones. El modelo probabilístico aprendido, en este caso, la red Bayesiana.
Postcondiciones. Un vector con todas las probabilidades.
Flujo de eventos. 1. El usuario crea una lista de los individuos que desea conocer su probabilidad. 2. El usuario llama a la función de cálculo de probabilidad proporcionándole el modelo aprendido y la lista de individuos. 3. Se obtiene un vector con todas los probabilidades.

Tabla 5: Caso de uso extendido para el cálculo de la probabilidad del algoritmo EBNA.



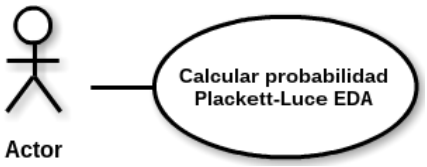
Nombre. Solucionar un problema de optimización mediante el algoritmo EHBSA.
Descripción. Obtener la solución a un problema concreto de optimización mediante el algoritmo EHBSA.
Actores. Usuario.
Precondiciones. La población inicial del problema debe ser una matriz en la que cada fila sea un individuo.
Postcondiciones. Solución al problema de optimización definido.
Flujo de eventos. 1. El usuario crea una población inicial válida para el problema a optimizar. 2. El usuario llama a la función de optimización con el algoritmo EHBSA 3. Se obtiene la solución final.

Tabla 6: Caso de uso extendido. Algoritmo EHBSA.



Nombre. Solucionar un problema de optimización mediante el algoritmo Plackett-Luce EDA.
Descripción. Obtener la solución a un problema concreto de optimización mediante el algoritmo PLEDA.
Actores. Usuario.
Precondiciones. La población inicial del problema debe ser una matriz en la que cada fila sea un individuo.
Postcondiciones. Solución al problema de optimización definido.
Flujo de eventos. 1. El usuario crea una población inicial válida para el problema a optimizar. 2. El usuario llama a la función de optimización con el algoritmo Plackett-Luce. 3. Se obtiene la solución final.

Tabla 7: Caso de uso extendido. Algoritmo PLEDA.



Nombre. Obtener la probabilidad de una población de individuos dado el modelo aprendido por Plackett-Luce EDA.
Descripción. Conocer la probabilidad de una población de individuos una vez aprendido el modelo probabilístico que define al PLEDA.
Actores. Usuario.
Precondiciones. El modelo probabilístico aprendido, en este caso, un vector de pesos.
Postcondiciones. Un vector con todas las probabilidades de la población proporcionada.
Flujo de eventos. 1. El usuario crea una lista con los individuos que desea conocer su probabilidad. 2. El usuario llama a la función de cálculo de probabilidad proporcionándole el modelo aprendido y la lista de individuos. 3. Se obtiene un vector con todas los probabilidades.

Tabla 8: Caso de uso extendido para el cálculo de probabilidades dado el modelo PLEDA.

5 ANÁLISIS Y DISEÑO

En el presente capítulo se detalla el análisis y diseño del proyecto. Además, se exponen los diagramas de clases y los diagramas de secuencia que ilustran el flujo de los métodos implementados.

5.1 DIAGRAMA DE CLASES

En esta sección se expondrá el diagrama de clases completo del paquete *metaheuR* (ver Figura 10). Después, se hará especial incapié en las partes realizadas en este proyecto. Para ello, se remarcan las clases realizadas en este proyecto en color para facilitar la lectura del mismo.

Dada la naturaleza de R, como se explicó en la Sección 1.6, el diagrama de clases es inconexo. Esto se debe a que las clases son unidas entre sí por funciones genéricas propias del lenguaje R que no están necesariamente implementadas en las clases que se exponen en este proyecto. Un ejemplo de este tipo de funciones es el método *simulate* que comparten las tres clases implementadas en este TFG, además de muchas otras clases de R.

La clase *Permutation* no ha sido implementada en este TFG pero se hace uso de ella ya que las clases EHBSA y PlackettLuce hacen uso de esta clase. El objetivo de la clase *Permutation* consiste en proporcionar una estructura de datos que asegure que los individuos muestreados cumplen con las propiedades de ser una permutación. Además, existen funciones que posibilitan la obtención de permutaciones aleatórias del tamaño elegido.

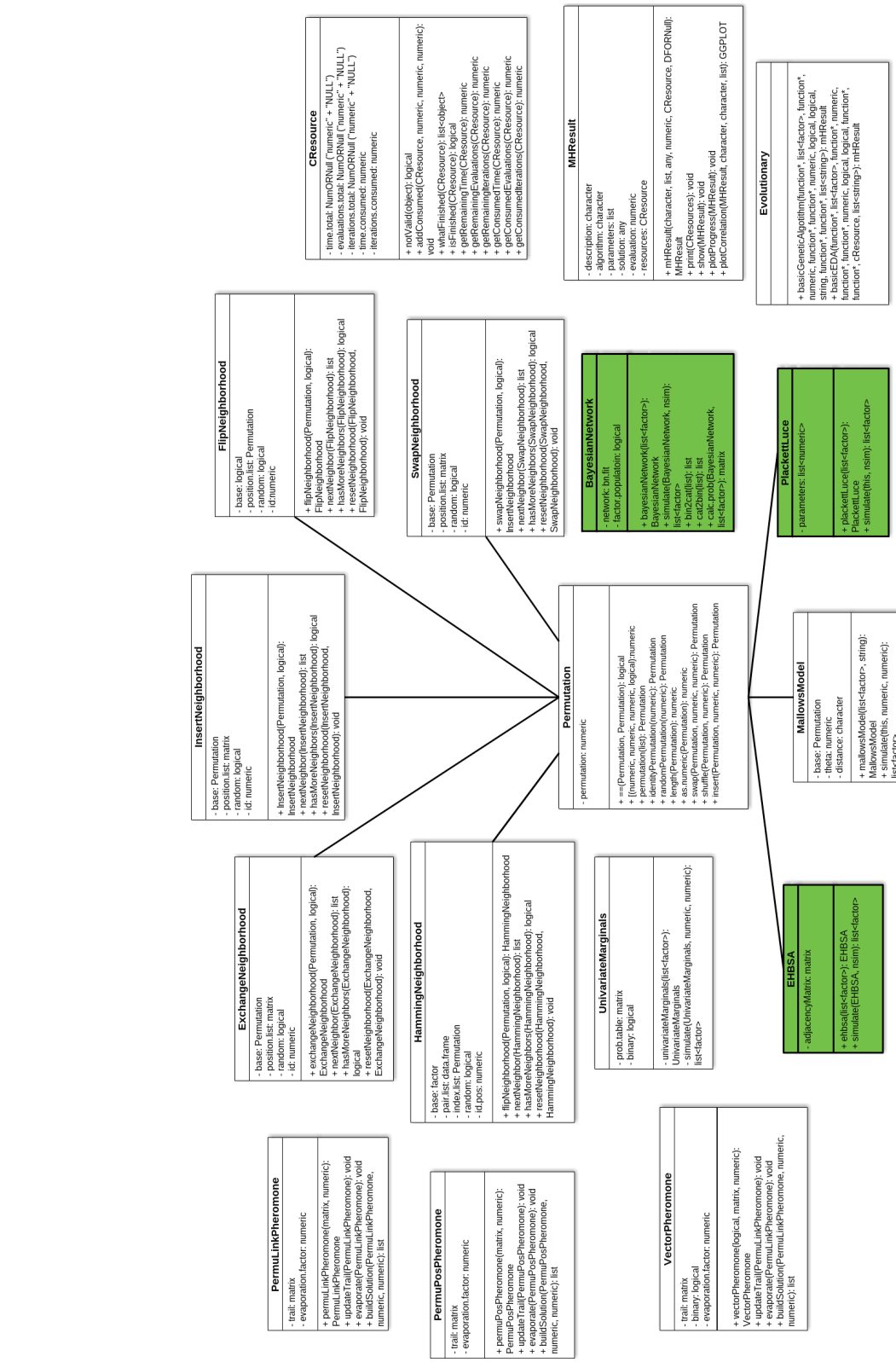


Figura 10: Diagrama de clases completo.

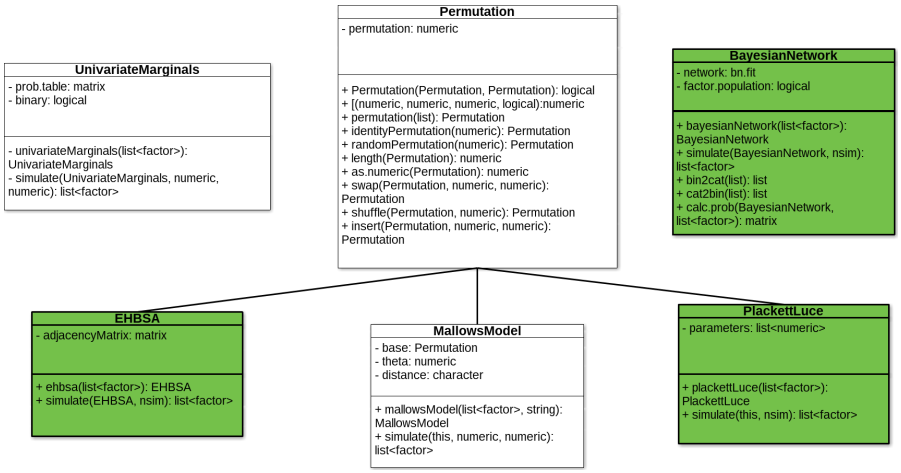


Figura 11: Diagrama de clases realizado.

En concreto, en este proyecto se han desarrollado las clases mostradas en al Figura 11 cuyo fundamento teórico se ha presentado en el Capítulo 2.

- **BayesianNetwork.** Algoritmo *EBNA*.
- **EHBSA.** Algoritmo *EHBSA*.
- **PlackettLuce.** Algoritmo *Plackett-Luce EDA*.

BayesianNetwork

La clase *BayesianNetwork* alberga dos atributos, el primero de ellos es *network* que pertenece a la clase *bn.fit* proporcionada por el paquete *bnLearn* [Scutari, 2009]. Este atributo contiene la red Bayesiana aprendida así como los parámetros esenciales que la definen. Y en segundo lugar, el *slot factor.population* es una variable de tipo lógico que será verdadero o falso dependiendo del tipo de datos de la población proporcionada para el aprendizaje. En caso de que la población fuese de valores categóricos, será verdadero; falso en caso contrario.

Los métodos que ofrece la clase son por un lado *bayesianNetwork* que se encarga de dada una población de individuos aprender la red Bayesiana que mejor se ajusta a dicha población. Por otro lado, el método *simulate* muestrea la red aprendida tantas veces como se le pasa por parámetro obteniendo entonces una nueva población de soluciones posibles al problema. Además, el método *calc.prob* es el encargado de calcular la probabilidad de los individuos dado el modelo. Por último, los métodos *bin2cat* y *cat2bin* son los encargados de realizar la conversión de tipos de la población. Estos transforman los individuos con variables binarias a categóricas y de variables categóricas a lógicas respectivamente.

EHBSA

La clase *EHBSA* se encarga de gestionar el aprendizaje y el muestreo en el algoritmo *EHBSA*. Esta clase tiene un único atributo, *adjacencyMatrix*, siendo la matriz de adyacencias aprendida de la población.

El método *EHBSA* es el encargado de aprender la matriz de adyacencias dada una población. Por otro lado, el método *simulate* se encarga de muestrear el modelo tantas veces como le indicamos por parámetro. La técnica utilizada para este proceso es *roulette wheel* o método de la ruleta sesgada.

PlackettLuce

La clase *PlackettLuce* permite ejecutar el algoritmo *Plackett-Luce EDA* dentro del paquete *metaheuR*. Esta clase consta de un único atributo, *parameters*. Este vector contiene los pesos de los elementos de los individuos de la población dada. (Capítulo 1)

Los métodos de la clase son *plackettLuce*, encargado de aprender el modelo probabilístico dada una población, y el método *simulate* que muestrea el modelo tantas veces como se le indica como parámetro. Por último, el método *calc.prob* se encarga de obtener la probabilidad de una población de individuos dado el modelo.

5.2 DIAGRAMAS DE SECUENCIA

En esta sección se muestran los diagramas de secuencia asociados a los métodos implementados. A modo de abreviación, se muestra únicamente un diagrama ya que la secuencia de ordenes es la misma para todos. No obstante, los diagramas restantes se pueden encontrar en el Apéndice A.

La Figura 12 alberga el diagrama de secuencia para la función de aprendizaje (izquierda), el muestreo (derecha) y el cálculo de la probabilidad de una población de individuos dado el modelo. Como vemos, el método de aprendizaje recibe una población y nos devuelve un objeto de la clase *BayesianNetork*. En el caso del muestreo, el método *simulate* recibe tanto el objeto *BayesianNetwork* como el número de nuevos individuos que queremos obtener; devolviéndonos una lista de objetos de la clase *factor* con las nuevas soluciones muestreadas. Por último, la imagen inferior de la figura corresponde al cálculo de probabilidad de una población de soluciones dado el modelo. Como resultado de este método, se obtienen las probabilidades asociadas a los individuos en un vector.

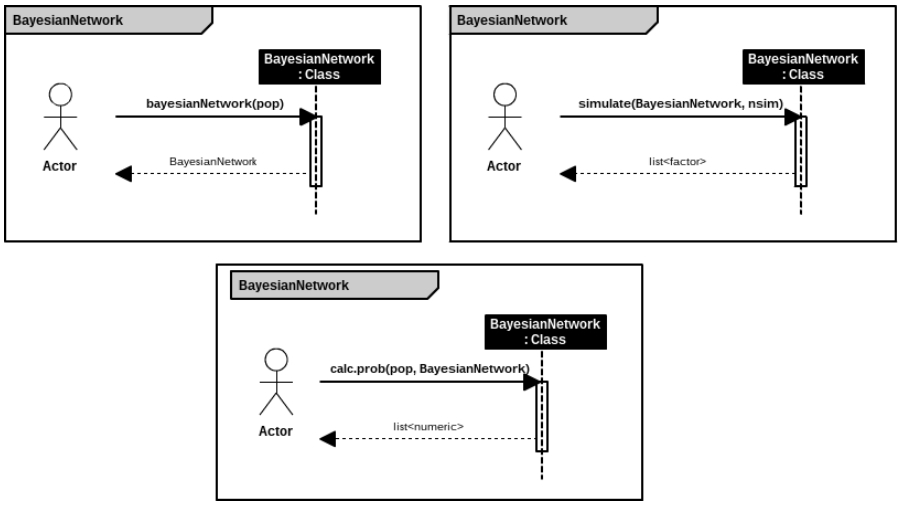


Figura 12: Diagrama de secuencia BayesianNetwork.

6 IMPLEMENTACIÓN

En este capítulo se recogen las técnicas utilizadas en la implementación de los EDAs desarrollados en este TFG. Además, se añaden referencias a capítulos anteriores que ayudarán a sobrellevar la sección.

Durante este capítulo se exponen los pasos realizados con trozos de código así como con figuras y tablas explicativas para los 3 algoritmos implementados, EBNA (Sección 2.1), EHBSA (Sección 2.2) y Plackett Luce EDA (Sección 2.3).

6.1 ESTIMATION OF BAYESIAN NETWORK ALGORITHM

El EBNA es un EDA que aprende un modelo gráfico probabilístico, en concreto, la red Bayesiana que mejor se ajusta a la población, tal y como se expuso en la Sección 2.1. Este EDA optimiza problemas de optimización combinatoria.

A continuación se explica cómo se realiza el aprendizaje de la red Bayesiana dada una población de individuos. Para ello, a modo de ejemplo primero se genera una población binaria de forma aleatoria como la de la Tabla 9. A partir de esta, se aprende la red que mejor se ajuste mediante la función *bayesianNetwork* (ver Código 1).

Este método internamente hace una conversión de la población de variables binarias a categóricas para que el paquete *bnLearn* pueda aprender la red Bayesiana mediante la función *hc*. Como se describió en la sección 2.1, se utiliza el algoritmo de búsqueda local *Hill Climbing* para encontrar la red Bayesiana que mejor se ajusta a los datos [Scutari, 2009]. Dado que el usuario puede aprender una población con variables tanto binarias como categóricas, se guarda en el *slot* de tipo lógico *population.factor* un valor binario que especifica el tipo de población que se utilizó en la etapa de aprendizaje. Como se puede observar en el Código 2, en primer lugar se hacen dos comprobaciones, la población inicial debe ser una lista de individuos y después, se debe comprobar el tipo de datos de las variables de los individuos. Si estas son de tipo lógico, el *slot factor.population* se especificará a falso mientras que será verdadero cuando las variables de los individuos sean categóricas. A continuación, en caso de que las variables sean lógicas, se realiza una conversión mediante la función *bin2cat* (ver Código 3). Después, la población se especifica en forma de matriz y por último, se llama a la función *hc* para aprender la estructura de la red. Para obtener los parámetros de dicha estructura, es necesario llamar

Individuo	Solución				
1	F	T	F	F	F
2	F	F	F	T	F
3	F	T	F	F	F
4	F	T	F	F	F
5	F	F	F	F	F
6	F	T	F	F	F
7	F	F	F	F	F
8	F	F	F	F	F
9	F	T	F	F	F
10	F	T	F	F	F
11	F	F	F	T	F
12	F	F	F	T	F
13	F	T	F	F	F
14	F	T	F	F	F
15	F	F	F	T	F

Tabla 9: Población inicial de vectores binarios.

Código 1: Creación de la población binaria y aprendizaje de la red Bayesiana en R.

```
library(metaheuR)
n <- 5
w <- runif(n)
l <- sum(w[runif(n) > 0.66])
v <- w*3 + runif(n)
problem <- # Un problema de optimizacion
pop.size <- 15
population <- lapply(1:pop.size,
                     FUN=function(i){
                       return(problem$correct(runif(n)>0.5))
                     })
population <- lapply(population, FUN=bin2cat)
model <- bayesianNetwork(population) # Se aprende la red
    Bayesiana
```

Código 2: Aprendizaje de la clase *BayesianNetwork*.

```
bayesianNetwork <- function(data, ...) {
  if(class(data) == "list"){
    if(class(data[[1]]) == "logical"){
      # The model is going to learn from a logical population
      factor.population <- F # is not a factor population
      aux <- lapply(data, FUN=bin2cat)
      pop.cat.df <- data.frame(t(matrix(unlist(aux), nrow =
          length(aux[[1]]), byrow = F)), stringsAsFactors = T)
      names(pop.cat.df) <- paste("X", 1:length(pop.cat.df),
          sep = "")
      for(i in 1:length(pop.cat.df)){
        pop.cat.df[[i]] <- factor(x=pop.cat.df[[i]], levels=c(
            "T","F"))
      }
    }else if(class(data[[1]]) == "factor"){
      factor.population <- T
      pop.cat.df <- data.frame(t(matrix(unlist(data), nrow =
          length(data[[1]]), byrow=F)), stringsAsFactors = T)
        }
    network <- hc(x = pop.cat.df)
    network <- bn.fit(network, data=pop.cat.df)
    obj <- new("BayesianNetwork", network = network, factor.
        population = factor.population)
  }else{
    stop("The data must be a list")
  }
}
```

a la función *bn.fit* que dada la estructura y la población, obtiene los parámetros máximo verosímiles que mejor se ajustan a los datos.

Una vez aprendida la red Bayesiana se muestrea una nueva población a partir del modelo aprendido con la siguiente sentencia:

```
new.population <- simulate(model, n=15)
```

En esta ocasión, se ha decidido obtener únicamente 15 muestras, sin embargo, se puede reemplazar el valor de n por el número que se desee. En la Tabla 10 se muestra la nueva población.

Internamente el método *simulate* hace uso de la función *rbn* del paquete *bnLearn* [Scutari, 2009]. Además, el usuario debe obtener el mismo tipo de datos que proporcionó cuando se aprendió el mode-

Código 3: Función *bin2cat* de la clase *BayesianNetwork*.

```
bin2cat <- function(bin.v){
  chr.v <- rep("F", length(bin.v))
  chr.v[bin.v] <- "T"
  catVector <- factor(chr.v, levels=c("T", "F"))
  return(catVector)
}
```

Código 4: Función de muestreo de la clase *BayesianNetwork*.

```
function(object, nsim=1, seed=NULL, ...) {
  if(!is.null(seed)) {
    set.seed(seed)
  }
  aux <- rbn(object@network, nsim, debug=F)
  aux <- as.list(as.data.frame(t(aux)))
  if(object@factor.population){
    # We must return the same type of population which was
        given before
    for(i in 1:length(aux)){
      for(j in 1:length(aux[[i]])){
        if(is.na(aux[[i]][j])){
          aux[[i]][j] <- sample(levels(aux[[i]]), 1)
        }
      }
    }
  }else{
    aux <- lapply(aux, FUN=cat2bin)
    for(i in 1:length(aux)){
      for(j in 1:length(aux[[i]])){
        if(is.na(aux[[i]][j])){
          aux[[i]][j] <- runif(1) >= 0.5
        }
      }
    }
  }
  return(aux)
})
```

lo, es decir, individuos de variables lógicas o categóricas. Por tanto, como se muestra en el Código 4, se hace una comprobación del *slot factor.population* para devolver una población u otra.

6.1.1 Cálculo de probabilidad

Una de las funciones de esta clase es la obtención de la probabilidad de una población de individuos dado el modelo. A modo de ejemplo, se quiere saber la probabilidad de todo el espacio de búsqueda, es decir, la probabilidad de todas las soluciones posibles para el problema definido. Para obtener la probabilidad se ejecuta la siguiente sentencia:

```
calc.prob(model, all.space)
```

Dicha sentencia devuelve todas las probabilidades en un vector.

Num	Solución				
1	T	T	F	F	F
2	F	T	F	F	F
3	F	T	F	F	F
4	T	T	F	F	F
5	T	T	F	F	F
6	T	T	F	F	F
7	T	T	F	F	F
8	F	T	F	F	F
9	T	T	F	F	F
10	T	T	F	F	F
11	T	T	F	F	F
12	T	T	F	F	F
13	F	T	F	F	F
14	T	T	F	F	F
15	T	T	F	F	F

Tabla 10: Población muestreada de vectores binarios.

Código 5: Creación de la población de permutaciones en R.

```
library(metaheuR)
n <- 5
pop.size <- 15
permu.i.pop <- lapply(1:pop.size,
                      FUN=function(i){
                        return(randomPermutation(n))
                      })
ehbsa.model <- ehbsa(permu.i.pop) # Sentencia que aprende el
    modelo ehbsa.
```

6.2 EDGE HISTOGRAM BASED SAMPLING ALGORITHM

Dada una población de soluciones, el EHBSA aprende una matriz de probabilidades marginales de orden dos. Este algoritmo se basa en contar las apariciones adyacentes de pares de elementos. Tomando como ejemplo un problema de tamaño 5, es decir, los individuos consisten en permutaciones de 5 variables, dada una población como la descrita en la Tabla 11, la matriz correspondiente sería la mostrada en la Tabla 12. La población de la cual se ha aprendido el modelo probabilístico, como se muestra en el Código 5, se ha generado de forma aleatoria con 15 individuos.

Internamente, el aprendizaje del modelo se realiza contando el número de apariciones que el elemento i precede de forma adyacente al j. Como se puede observar en el Código 6, en primer lugar se comprueba que la población esa una lista de individuos. A continuación, como se explicó en la Sección 2.2, se comprueba si el parámetro de control ha sido especificado por el usuario, en caso de no haber sido proporcionado intencionadamente, se asigna el valor 0,5. Después, se cuenta el número de apariciones para rellenar la matriz de adyacencias.

Código 6: Aprendizaje del modelo probabilístico EHBSA.

```
ehbsa <- function(data, bratio, ...) {
  if(class(data) == "list"){
          if(missing(bratio)){
            bratio <- 0.5
          }
                # We can create the object
          # Learn matrix definition
          adjacencyMatrix <- matrix(c(0) , nrow=length(data
              [[1]]), ncol=length(data[[1]])) #empty matrix
            for(i in 1:length(data)){
              for(j in 2:length(data[[i]]@permutation)){
                adjacencyMatrix[ data[[i]]@permutation[j
                    -1], data[[i]]@permutation[j] ] <-
                    adjacencyMatrix[ data[[i]]@permutation
                    [j-1], data[[i]]@permutation[j] ] + 1
              }
            }
          colnames(adjacencyMatrix) <- paste("X", 1:length(
              adjacencyMatrix[1,]), sep="")
          rownames(adjacencyMatrix) <- paste("X", 1:length(
              adjacencyMatrix[1,]), sep="")
          adjacencyMatrix <- adjacencyMatrix + bratio
          diag(adjacencyMatrix) <- 0
          obj <- new("EHBSA", adjacencyMatrix =
              adjacencyMatrix)
        }else{
            stop("The data must be a list")
        }
}
```

Individuo	Población				
1	5	1	4	3	2
2	5	2	3	4	1
3	3	2	1	4	5
4	5	4	3	1	2
5	1	2	3	4	5
6	4	2	3	5	1
7	4	5	2	1	3
8	4	1	3	5	2
9	3	2	4	1	5
10	5	2	3	1	4
11	1	2	3	4	5
12	3	1	2	4	5
13	3	5	2	4	1
14	5	3	2	4	1
15	3	4	1	2	5

Tabla 11: Población inicial de permutaciones.

Código 7: Muestreo del modelo EHBSA.

```
function(object, nsim=1, seed=NULL, ...) {
  # we create a roullette wheel
  new.population <- matrix(, nrow=0, ncol = length(
      object@adjacencyMatrix[1,]))
  for(q in 1:nsim){
    l.matrix <- object@adjacencyMatrix
    l <- c(sample(0, length(l.matrix[1,]), replace=TRUE)) #new
        list with numbers
    l[1] <- round(runif(1, 1, length(l.matrix[1,])), 0)
    l.matrix[, l[1]] <- c(0)
    for(i in 2:length(l.matrix[1,])){ # iterate positions of
        the new individual
      val <- runif(1, 0, sum(l.matrix[l[i-1],]))
      j <- 1
      acc <- l.matrix[l[i-1], j]
      #acc <- 0
      while(val > acc){
        j <- j + 1
        acc <- acc + l.matrix[l[i-1],j]
      }
      l[i] <- j
      l.matrix[, j] <- c(0)
    }
    l <- permutation(l)
    new.population <- c(new.population, l)
  }
  return(new.population)
})
```

	1	2	3	4	5
1	0.00	5.50	2.50	3.50	1.50
2	2.50	0.00	5.50	4.50	1.50
3	3.50	4.50	0.00	4.50	3.50
4	6.50	1.50	2.50	0.00	5.50
5	2.50	5.50	1.50	1.50	0.00

Tabla 12: Matriz de adyacencias aprendida por el EHBSA.

	X1	X2	X3	X4	X5
Pesos	0,15968265	0,23724955	0,22436262	0,08362443	0,29508076

Tabla 13: Vector de pesos aprendido por el modelo Plackett Luce tras ejecutar el comando.

Pese a que el EHBSA aprende probabilidades marginales de orden dos, para ahorrar coste computacional, no se normalizan las filas de la matriz para obtener dichas probabilidades. En el muestreo del modelo probabilístico se tiene en cuenta esta asunción. Como se puede ver en el Código 7, en primer lugar se genera una nueva matriz nula que se irá rellenando con los nuevos individuos. A continuación se generan las nuevas soluciones individualmente. Cada individuo, es generado obteniendo un elemento aleatorio entre 0 y el número de elementos totales. A continuación, se anula la columna de el elemento muestreado. Los siguientes elementos del individuo son generados teniendo en cuenta el elemento seleccionado en el paso anterior. Para ello, se utiliza la técnica de la ruleta sesgada (*roulette-wheel*) [Bäck, 1996]. Después, se anula la columna correspondiente al elemento muestreado de la matriz de adyacencias. La sentencia que obtiene la nueva población del modelo EHBSA corresponde con la mostrada a continuación:

```
new.population <- simulate(ehbsa.model, n=15)
```

6.3 PLACKETT-LUCE EDA

Como se explicó en la Sección 2.3, este algoritmo aprende un vector de pesos que representa al conjunto de soluciones. Dada una población como la de la Tabla 11, mediante el algoritmo *Minorization-Maximization* (MM) que trata de buscar la convexidad de una función para obtener su mínimo o máximo [Hunter, 2004]. El aprendizaje del modelo se realiza con la siguiente sentencia:

```
plackettLuce.model <- plackettLuce(population)
```

Para que el algoritmo MM converja a solución, siempre se introduce el reverso del primer individuo de la población dada para aprender. Esto se realiza para que la función a optimizar no sea lineal ya que si lo fuera, nunca se encontraría el máximo de dicha función. Por otro lado, el método de aprendizaje tiene un parámetro opcional llamado *max.iter* que especifica el número máximo de iteraciones que realizará el MM. Una vez se supere este límite especificado a 10000 iteraciones por defecto, se devolverá el vector de pesos obtenido hasta esa iteración. En la Tabla 13 se muestra el vector de pesos aprendido por el algoritmo.

En el muestreo de este modelo, se utiliza la técnica de *roullette-wheel* [Bäck, 1996] mediante la siguiente sentencia:

```
new.pop <- simulate(plackettLuce.model, n=15)
```

6.3.1 Cálculo de probabilidad

Este modelo posibilita el cálculo de la probabilidad de una población de inviduos dado un modelo probabilístico con la Ecuación 9. Para ello, se ejecuta la siguiente sentencia.

```
calc.prob(plackettLuce.model, population)
```

Se obtiene un vector con las probabilidades de los individuos.

7 EXPERIMENTACIÓN

En este capítulo, se describen las pruebas realizadas para comprobar el correcto funcionamiento de los EDAs implementados, así como el rendimiento y eficiencia de los mismos sobre problemas de optimización combinatoria.

7.1 VERIFICACIÓN DEL CORRECTO FUNCIONAMIENTO

Esta sección describe los experimentos realizados para comprobar que las funciones de aprendizaje y muestreo de los distintos modelos probabilísticos funciona correctamente. De forma adicional, para el EBNA y el *Plackett-Luce* EDA, se comprueba la función que calcula la probabilidad de una población de soluciones dado el modelo.

7.1.1 Estimation of Bayesian Network Algorithm

Dada la complejidad de este algoritmo, y que además, el aprendizaje y muestreo del modelo se computan utilizando el paquete *bnLearn*, se han realizado cuatro experimentos diferentes que se detallan a continuación.

Para cada uno de ellos, se expone en primer lugar la hipótesis junto con los valores esperados, para concluir el apartado con los resultados finales y una breve discusión sobre las conclusiones obtenidas.

Experimento 1: Comprobación de la estructura de datos de la población

Se sabe que el paquete *bnLearn* acepta tanto una población compuesta por individuos de tipo *factor* como una lista de soluciones de tipo categóricos para el aprendizaje de una red Bayesiana. Para corroborar dicha afirmación se generan dos poblaciones.

- pop.1 ← Una lista de listas de valores lógicos.
- pop.2 ← Una lista de listas de *factor*.

VALORES ESPERADOS

Se espera obtener una red Bayesiana idéntica.

RESULTADOS OBTENIDOS

Los resultados de este experimento no son los esperados, por tanto,

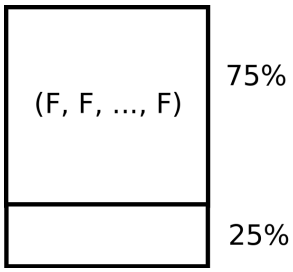


Figura 13: Población con el 75 % siendo el mismo individuo.

se corrige el proceso de aprendizaje del algoritmo añadiendo la validación de que la población pueda ser tanto lógica como categórica. Por ello, se añade el nuevo atributo *factor.population*.

El código relativo a este experimento se puede consultar en el Apéndice 11.

Experimento 2: Comprobación del correcto aprendizaje del modelo

En este experimento se comprueba que se esté aprendiendo el modelo probabilístico correctamente. Para ello se crea una población en la cual el 75 % de los valores son el individuo binario compuesto por todos sus *features* a *False*. El 25 % restante se genera de forma aleatoria. Se puede ver gráficamente en la Figura 13.

VALORES ESPERADOS

Se espera obtener una probilidad cercana al 75 % en los nodos de la red hacia el valor *False* ya que el 75 % de la población es ese valor.

RESULTADOS OBTENIDOS

Los resultados obtenidos son los esperados, obteniendo una probabilidad muy alta de obtener valor *False* en cada posición de la solución a muestrear.

El código del experimento se puede consultar en el Apéndice 12.

Experimento 3: Comprobar la obtención de una probabilidad

En este apartado se trata de comprobar el funcionamiento de obtener la probabilidad de cierto individuo en base a la red aprendida. Por tanto, se generará una población como la de la Figura 13.

Una vez aprendida la red, se calcula la probabilidad de todos los 2^n individuos del espacio de búsqueda, siendo n el número de *features* de cada individuo.

VALORES ESPERADOS

Se espera que la solución con mayor probabilidad del espacio de búsqueda, sea el que se repite el 75 % en la población de la Figura 13.

RESULTADOS OBTENIDOS

Se obtienen los resultados esperados, obteniendo una probabilidad de 0,7376 para el individuo que aparece el 75 % de las veces en la población, Figura 14.

El código del experimento se puede consultar en el Apéndice 13.

Experimento 4: Comprobación del correcto muestreo del modelo

Sabiendo que el EBNA obtiene una red Bayesiana como modelo probabilístico en su etapa de aprendizaje, el objetivo de este experimento es verificar el aprendizaje y el muestreo de la red conjuntamente. Para ello, se genera una población inicial (*i.pop*) de manera aleatoria. A continuación se aprende la red que mejor se ajusta a dicha población (*i.network*). Después muestrea la red (*i.network*) obteniendo así una nueva población (*sampled.pop*). Con esta nueva población (*sampled.pop*) se aprende una nueva red Bayesiana (*f.network*). Por último se comparan ambos modelos (*i.network* y *f.network*)

Para comparar los modelos *i.network* y *f.network*, como se muestra en el Algoritmo 3, se calcula la distribución de probabilidad de forma explícita sobre el espacio de búsqueda y posteriormente se computa la divergencia de *Kullback-Leibler* que se calcula mediante la expresión 10. Esta medida proporciona la distancia entre ambas distribuciones de probabilidad.

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)} \tag{10}$$

Los valores iniciales para el experimento son:

- **i.pop**. 25 individuos y 10 atributos binarios.
- **i.network**. La red Bayesiana aprendida con la población inicial.
- **all.pop**. Todos los posibles individuos que puedan ser generados.

Se espera que el valor de la divergencia se aproxime a 0 a medida que se repite el tamaño de la población.

VALORES ESPERADOS

Se espera obtener una divergencia cercana a 0.

RESULTADOS OBTENIDOS

Se obtienen los resultados esperados. Los valores van decrementando a medida que se incrementa las población inicial. Este comportamiento se puede observar en la Figura 14. El eje X corresponde al tamaño de la población y el eje Y representa la divergencia. Los individuos cuentan con 5 características en cada prueba.

El script está disponible en el Apéndice 14.

Algorithm 3: Pseudocode for testing EBNA.

```
1 i.pop ← Generate the initial population randomly.
2 i.network ← bayesianNetwork(i.pop) ;   // Learn an object of the
  class BayesianNetwork.
3 all.pop ← Generate all the possible binary solutions from 0 to
  2^numFeatures
4 sampled.pop ← simulate(network)
5 f.network ← bayesianNetwork(sampled.pop)
6 kullback-leibler(i.network, f.network, all.pop) ;      // Compute the
  Kullback-Leibler divergence between first and last networks with
  the complete individual space.
```

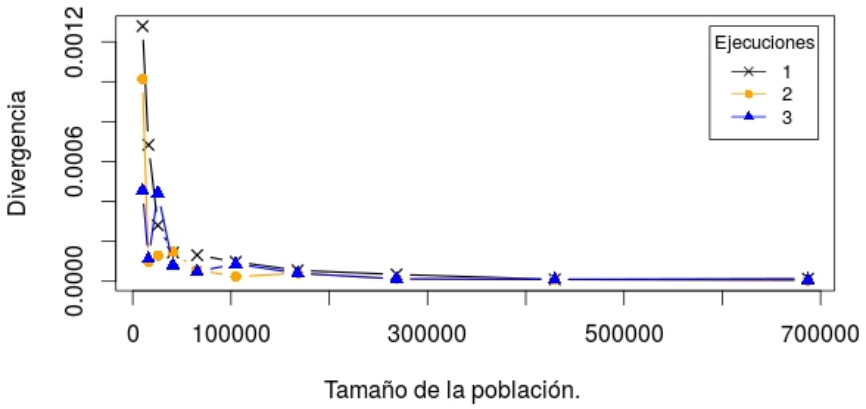


Figura 14: Cálculo de la divergencia de *Kullback-Leibler* con una población de 5 variables.

	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
X_1	0	50	0	0	0	0	0	0	0	0
X_2	70	0	50	0	0	0	0	0	0	0
X_3	0	70	0	50	0	0	0	0	0	0
X_4	0	0	70	0	50	0	0	0	0	0
X_5	0	0	0	70	0	50	0	0	0	0
X_6	0	0	0	0	70	0	50	0	0	0
X_7	0	0	0	0	0	70	0	50	0	0
X_8	0	0	0	0	0	0	70	0	50	0
X_9	0	0	0	0	0	0	0	70	0	50
X_{10}	0	0	0	0	0	0	0	0	70	0

Tabla 14: Matriz de adyacencias aprendida por el EHBSA con permutaciones identidad e inversas de la identidad.

7.1.2 Edge Histogram Based Sampling Algorithm

Para comprobar el correcto funcionamiento de este algoritmo, es necesario verificar el aprendizaje y el muestreo del modelo probabilístico que lo caracteriza. Por ello, se han realizado 3 experimentos. El primero de ellos, verifica el aprendizaje mientras que los otros dos comprueban el muestreo de nuevas soluciones.

Experimento 1: Comprobar el aprendizaje del modelo

Sabiendo que el EHBSA calcula una matriz de adyacencias, se ha creado una población de 120 permutaciones de las cuales 50 eran la permutación identidad (1 2 3 ... n) y los 70 individuos restantes eran la reversa de la identidad (n (n − 1) (n − 2) ... 1).

VALORES ESPERADOS

En este experimento, se espera obtener la matriz representada en la Tabla 14.

Se espera que los resultados se concentren en la diagonal principal de la matriz de adyacencias.

RESULTADOS OBTENIDOS

El resultado obtenido tras el aprendizaje se puede ver en la Tabla 14. Se observa que las apariciones se concentran en el área cercana a la diagonal. Por tanto, en base a este experimento se verifica que aparentemente el algoritmo aprende correctamente.

El script se puede consultar en el Apéndice 15.

Experimento 2: Comprobación del muestreo I

En este experimento, se manipulará la matriz de adyacencias tal que el único valor posible al muestrear el modelo sea el que se ha definido, en este caso (3 2 1 4 5 6 7 8 9 10). La matriz asociada a este experimento se puede ver en la Figura 15.

	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	$\sum$
X_1	0	0	0	100	0	0	0	0	0	0	100
X_2	100	0	0	0	0	0	0	0	0	0	100
X_3	0	100	0	0	0	0	0	0	0	0	100
X_4	0	0	0	0	100	0	0	0	0	0	100
X_5	0	0	0	0	0	100	0	0	0	0	100
X_6	0	0	0	0	0	0	100	0	0	0	100
X_7	0	0	0	0	0	0	0	100	0	0	100
X_8	0	0	0	0	0	0	0	0	100	0	100
X_9	0	0	0	0	0	0	0	0	0	100	100
X_{10}	0	0	0	0	0	0	0	0	0	100	100

Tabla 15: Matriz de adyacencias del EHBSA forzada para obtener un individuo concreto.

	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	$\sum$
X_1	0	0	5	70	5	1	1	3	5	10	100
X_2	80	0	0	0	0	0	0	0	0	20	100
X_3	2	80	0	5	5	2	2	2	2	0	100
X_4	0	0	0	0	80	20	0	0	0	0	100
X_5	0	0	0	10	0	90	0	0	0	0	100
X_6	30	0	0	0	0	0	70	0	0	0	100
X_7	0	0	10	0	10	0	0	80	0	0	100
X_8	0	0	30	0	0	10	0	0	60	0	100
X_9	0	0	20	0	0	0	0	0	0	80	100
X_{10}	70	0	30	0	0	0	0	0	0	0	100

Tabla 16: Matriz de adyacencias del EHBSA forzada para obtener con cierta probabilidad un individuo concreto.

VALORES ESPERADOS

Como resultado del experimento se espera poder muestrear únicamente la permutación (3 2 1 4 5 6 7 8 9 10).

RESULTADOS OBTENIDOS

Los resultados obtenidos son altamente satisfactorios, el 100 % de las ocasiones ha resultado ser la permutación esperada.

El código referente a este experimento se puede consultar en el Apéndice 16.

Experimento 3: Comprobación del muestreo II

Para constatar el correcto muestreo del algoritmo se manipula la matriz de adyacencias aprendida de modo que obtengamos con una cierta seguridad una permutación esperada.

Como se ve en la Tabla 16, se va a obtener con cierta seguridad la permutación (3 2 1 4 5 6 7 8 9 10).

VALORES ESPERADOS

En este experimento se espera obtener una permutación en concreto, (3 2 1 4 5 6 7 8 9 10) la mayoría de las veces.

Parámetro	w_1	w_2	w_3	w_4	w_5
Valores	0,540	0,317	0,100	0,034	0,008

Tabla 17: Vector de pesos aprendido por el modelo *Plackett-Luce* en el experimento 2.

RESULTADOS OBTENIDOS

Los resultados obtenidos tras el experimento son los esperados. Sin embargo, se esperaba una mayor tasa de acierto. Para el experimento se han realizado 100 iteraciones de las cuales el 61 % han resultado ser la permutación esperada (3 2 1 4 5 6 7 8 9 10). Sin embargo, gran cantidad de las veces se obtiene una permutación distinta.

Esto ocurre porque a cada paso del muestreo, se debe actualizar la matriz de adyacencias con el fin de garantizar que el nuevo individuo sea una permutación.

Por tanto, los resultados son compatibles con la hipótesis de que el algoritmo aprende correctamente.

Se puede consultar el script en el Apéndice 17.

7.1.3 Plackett-Luce EDA

En el siguiente apartado se exponen las pruebas realizadas para la comprobación del algoritmo.

Para demostrar en correcto funcionamiento de los algoritmos, se expondrán primero las descripciones de los experimentos seguidos de los valores esperados y por último, se valorarán los resultados obtenidos.

Experimento 1: Comprobación del correcto aprendizaje del modelo

En este experimento se comprueba el aprendizaje del modelo probabilístico que caracteriza al modelo *Plackett-Luce*. Para ello, se crea una población de permutaciones de tamaño 5 en la cual el 75 % de los individuos son el mismo. En concreto, el individuo que se va a repetir es la permutación identidad [1 2 ... n].

VALORES ESPERADOS

Se espera obtener un vector de parámetros no uniforme que seguirá la siguiente expresión:

$$w = w_1 > w_2 > \ldots > w_n$$

RESULTADOS OBTENIDOS

Se obtiene un vector de parámetros en el cual los valores se ajustan a lo esperado. Por tanto, el mayor valor se presenta en el parámetro w_1 seguido de w_2, etc. Se puede ver el vector aprendido en la Tabla 17.

El código correspondiente a este experimento se puede consultar en el Apéndice 18.

Experimento 2: Comprobación del muestreo del modelo

El experimento consiste en crear una población aleatoria y aprender el modelo probabilístico asociado. A continuación, se muestrea para obtener una nueva población. Después se aprende un nuevo modelo *Plackett-Luce*. Por último se comparan el modelo inicial con el final mediante la divergencia de *Kullback-Leibler*.

Para el cálculo de la divergencia, se utilizará el espacio total de permutaciones cuyo tamaño es $n!$.

VALORES ESPERADOS

Se espera tener una divergencia de 0 entre ambos modelos probabilísticos.

RESULTADOS OBTENIDOS

Los resultados obtenidos son altamente satisfactorios, la divergencia es cercana a 0 y se corrobora que el EDA funciona correctamente. Se puede consultar el script en el Apéndice 19.

7.2 COMPARATIVA DE EDAS

El objetivo de esta sección es observar un comportamiento de los algoritmos implementados para poder ver y comparar la calidad de los mismos. Para ello, se resuelven problemas de optimización clásicos como el problema del agente viajero –*Traveling Salesman Problem* (TSP)– y el problema del subconjunto máximo independiente –*Maximum Independence Set* (MIS)–.

Para sobrellevar la lectura de la sección, en primer lugar se definen los problemas de optimización sobre los cuales se han realizado los experimentos. A continuación, se exponen las bases experimentales y se muestran los resultados obtenidos en tablas y gráficos. Por último, se obtienen conclusiones sobre los resultados obtenidos.

7.2.1 Problemas de optimización combinatoria

Los problemas de optimización clásicos representan diferentes escenarios que tratan de recrear situaciones reales.

Dada su naturaleza, estos problemas se dividen en dos categorías. Aquellos que sus soluciones se definen como variables de valores reales son conocidos como problemas continuos. Por otro lado están los que su solución es discreta, también conocidos como problemas

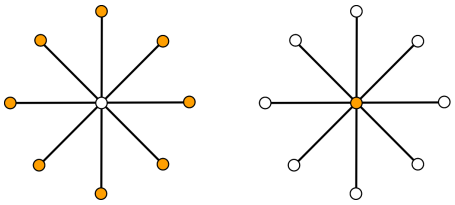


Figura 15: Ejemplo grafo MIS.

de combinatoria. Cabe destacar que dentro de los problemas de optimización combinatoria, existe un subconjunto llamado problemas de permutaciones. Un ejemplo de este tipo de problemas es el TSP, el cual se describe en esta sección.

Problema del agente viajero

El problema del viajero –*Travelling Salesman Problem*– [Goldberg y Lingle, 1985] ampliamente conocido como *TSP* responde a la siguiente pregunta: Dada una lista de ciudades y las distancias entre todas ellas, ¿Cuál es la ruta más corta que recorre todas las ciudades sólo una vez y finaliza en la ciudad de origen?

Este problema de optimización es un problema de permutaciones. En este documento, las ciudades se numeran de 1 a n donde n es la cantidad total de ciudades. Por tanto, las soluciones del *TSP* se pueden expresar como permutaciones de 1 a n.

Dada una matriz $D = [d_{i,j}]_{n\times n}$ que contiene la distancias entre todas las ciudades posibles, la función objetivo viene definida por la suma de las distancias entre las ciudades visitadas en el orden σ. Se obtiene la siguiente Ecuación 11.

$$f(\sigma) = \sum_{i=2}^{n} d_{\sigma(i-1),\sigma(i)} + d_{\sigma(n),\sigma(1)} \tag{11}$$

Problema del subconjunto máximo independiente

El problema del subconjunto máximo independiente –*Maximum Independent Set*– (MIS) en la teoría de grafos, un subcojunto independiente es un conjunto independiente de nodos que no es subconjunto de cualquier otro conjunto independiente del grafo. Asumiendo el ejemplo de la Figura 15, los nodos coloreados son subconjuntos independientes, sin embargo, se ve que el izquierdo es el MIS.

Por tanto, el problema para un grafo $G = (V, E)$ y un conjunto máximo independiente S, se define con la siguiente expresión:

$$v \in S \wedge N(v) \notin S \neq 0 \tag{12}$$

donde $N(v)$ representa los nodos vecinos de v.

7.2.2 Ejecuciones de los EDAs

En esta sección se comparan los resultados obtenidos tras la ejecución de 4 EDAs. 3 de ellos implementados en el proyecto: EBNA, EHBSA y PLEDA y un cuarto, el *Univariate Marginals Distribution Algorithm* (UMDA) [Pelikan y Mühlenbein, 1998]. Estas ejecuciones muestran el comportamiento sobre problemas de optimización como el MIS y el TSP.

Dado que el EBNA es un EDA diseñado para resolver problemas de optimización combinatoria, se compara con el UMDA. Además, este algoritmo ya se encontraba en el paquete *metaheuR*.

El código correspondiente a las ejecuciones está disponible en el Apéndice 20.

Bases experimentales

La experimentación se llevará a cabo sobre dos problemas de optimización clásicos, el MIS y el TSP. En el caso del MIS, las instancias (grafos) se crean de forma aleatoria. Por otro lado, para el TSP se hace uso de *TSPLib*[1] ya que se conoce el valor de la solución óptima del problema. Dado que son algoritmos estocásticos, dependen de la población inicial y el muestreo del modelo probabilístico. Por tanto, en ambos casos, cada instancia se optimizará 5 veces de forma que el resultado del algoritmo sea la mediana de esas 5 ejecuciones.

Las ejecuciones se han realizado con los siguientes parámetros de los EDAs.

- Tamaño de la población ← Será igual al tamaño del problema.
- Porcentaje de selección ← Corresponde al porcentaje de individuos seleccionados en cada generación. Se establece al 50 %.
- Selección de población ← El método de selección de los individuos, en este caso se elige elitista que selecciona a los mejores individuos de forma descendiente.
- Criterio de parada ← tiempo, 300 segundos.
- Repetición de cada instancia ← 5.

Para medir la calidad del algoritmo en base al óptimo, se utiliza el error normalizado que se define mediante la Ecuación 13. De esta forma, se obtiene una comparación normalizada de los algoritmos.

$$\text{Error} = -\frac{\text{Valoróptimo} - \text{Valorobtenido}}{\text{Valoróptimo}} \tag{13}$$

1 TSPLib es una repositorio de problemas TSP de la cual se pueden obtener tanto las matrices de coste que definen el problema como la mejor solución encontrada hasta la fecha. Enlace: comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Instancia	Mejor	UMDA	EBNA
1	6,000	0,933	**0,233**
2	8,925	0,704	**0,101**
3	10,000	0,776	**0,096**
4	10,000	0,785	**0,148**
5	11,980	0,738	**0,164**

Tabla 18: Resultados UMDA y EBNA.

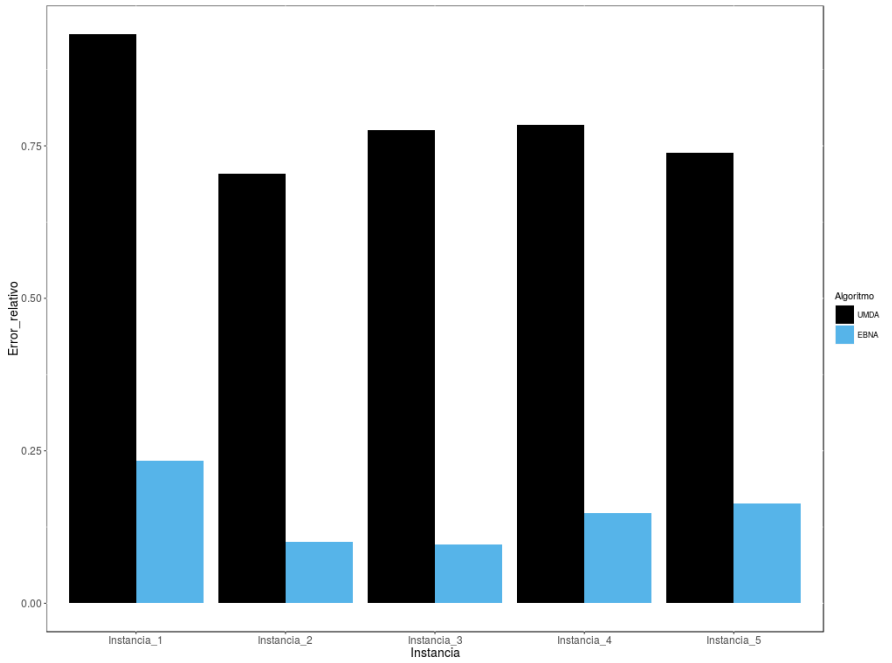


Figura 16: Comparativa de errores relativos entre el UMDA y el EBNA.

Resultados UMDA y EBNA

En la Tabla 18 se muestra el error normalizado respecto al óptimo. Viendo los resultados, se puede decir que el EBNA funciona mejor que el UMDA en el problema MIS. Sin embargo, viendo las curvas de optimización de las Figuras 17 y 18, se puede observar que las curvas de optimización del EBNA tienen mayor pendiente que las del UMDA, esto hace que el EBNA llegue a una solución mejor en un número menor de evaluaciones, sin embargo, pasadas ciertas iteraciones, el EBNA es capaz de obtener un resultado mejor.

Para concluir, cabe destacar que el EBNA obtiene una precisión mucho mayor ya que obtiene un resultado mejor que el EHBSA. Además, pese a que el UMDA ha realizado más evaluaciones que el EBNA por ser más rápido, el EBNA ha conseguido mejores resultados. Por último, decir que el EBNA llega antes a la solución óptima que el UMDA ya que en un número menor de iteraciones consigue una solución mejor.

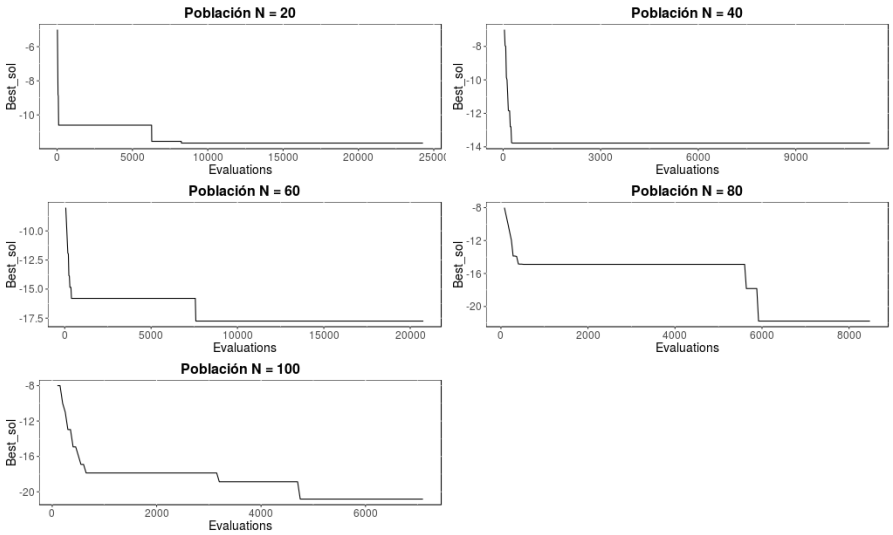


Figura 17: Resultado de las ejecuciones del UMDA sobre instancias del MIS.

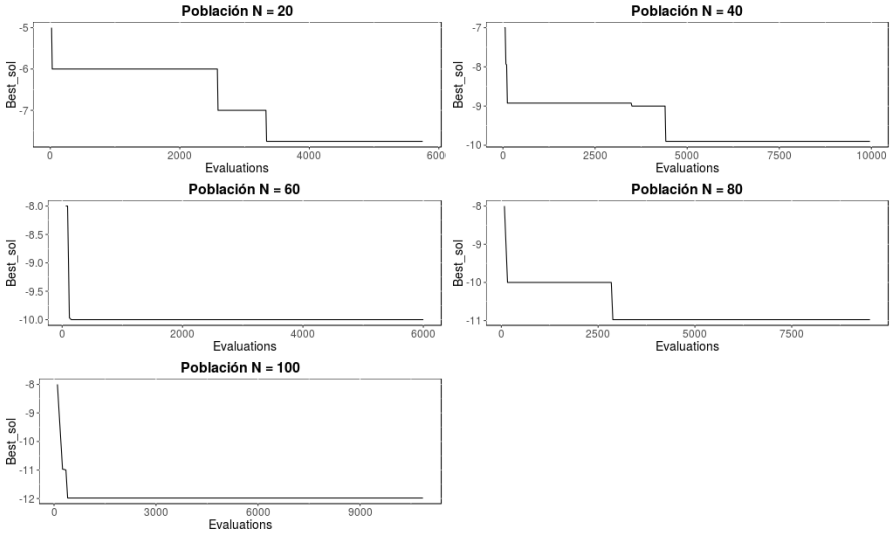


Figura 18: Resultado de las ejecuciones del EBNA sobre instancias del MIS.

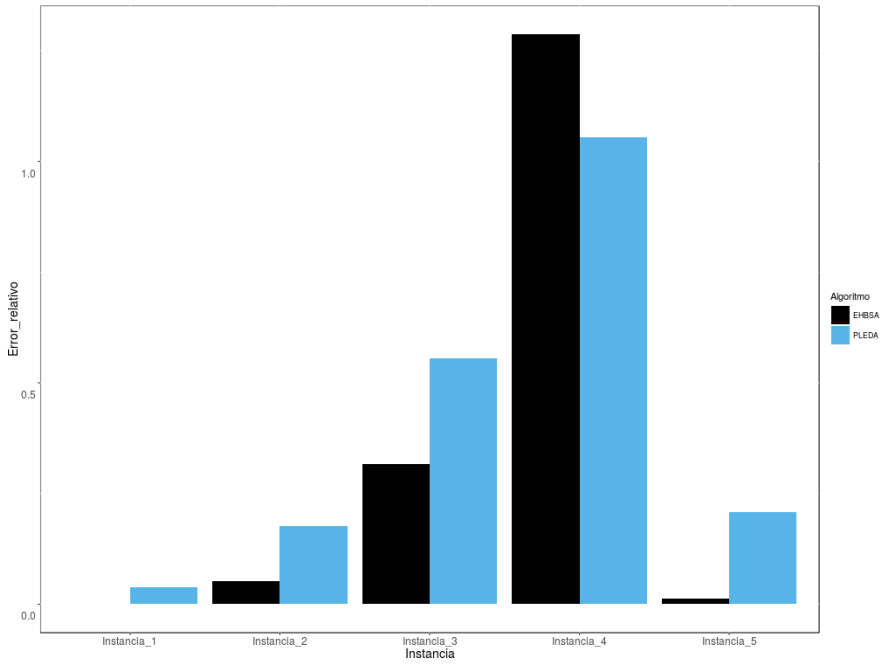


Figura 19: Comparativa de errores relativos entre el EHBSA y el PLEDA.

Resultados EHBSA y PLEDA

Para las ejecuciones del EHBSA y el PLEDA se han obtenido instancias TSP de *TSPLib* siendo las siguientes:

- burma14: 14 ciudades.
- ulysses22: 22 cuidades.
- ftv33: 33 ciudades.
- dantzig42: 42 ciudades.
- ft53: 53 ciudades.

Tras realizar las ejecuciones, en la Tabla 19 se muestra el error normalizado frente a la solución óptima extraída de TSPLib. En base a los resultados obtenidos, se puede decir que el EHBSA optimiza mejor el TSP. No obstante, viendo las curvas de optimización del EHBSA y el PLEDA en las figuras 20 y 21 se observa que el EHBSA realiza muchas más iteraciones que el PLEDA en el mismo intervalo de tiempo. Por otra parte, el EHBSA tiene una mayor precisión que el PLEDA. Si se compara la velocidad en la que se obtiene una mejor solución, el PLEDA obtiene mejores resultados en un número menor de iteraciones.

A modo de conclusión, cabe destacar que el modelo probabilístico que aprende el EHBSA es más rápido de aprender y por tanto, el número de evaluaciones que realiza son muy superiores a las del

Instancia	Mejor	EHBSA	PLEDA
burma14	3323	**0, 000**	0, 038
ulysses22	7013	**0, 052**	0, 176
ftv33	1591	**0, 317**	0,556
dantzig42	699	1, 287	**1, 054**
ft53	15645	**0, 013**	0, 207

Tabla 19: Resultados del EHBSA y PLEDA.

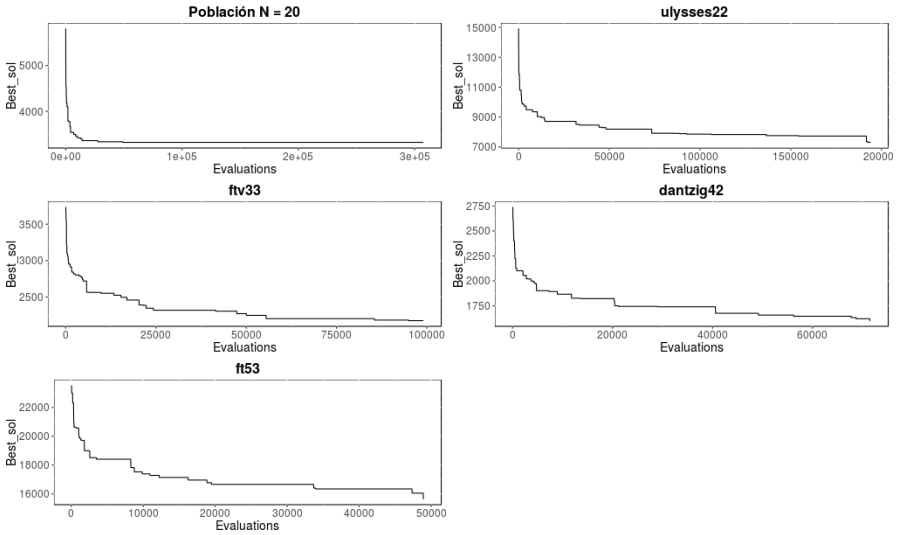


Figura 20: Resultado de las ejecuciones del EHBSA sobre instancias TSP.

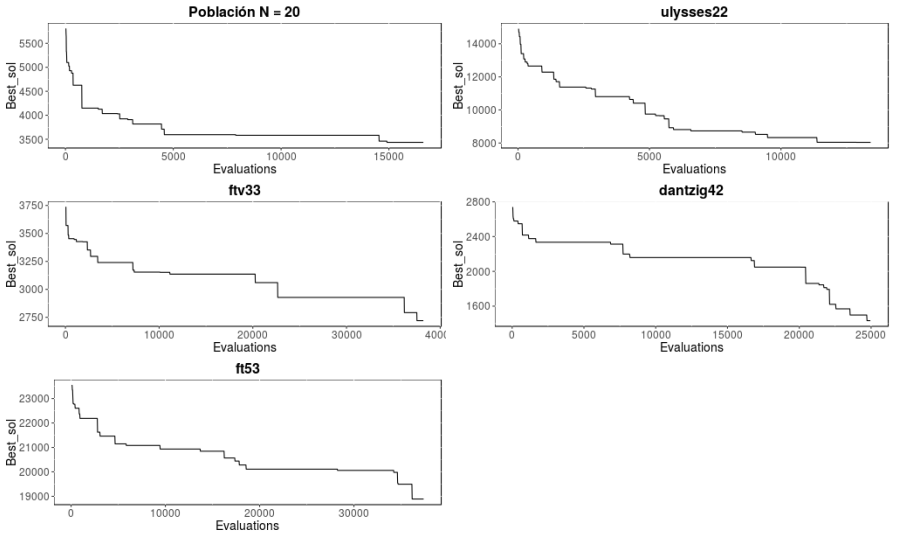


Figura 21: Resultado de las ejecuciones del PLEDA sobre instancias TSP.

PLEDA. Sin embargo, las curvas de optimización señalan que el EHBSA tiene menor pendiente que el PLEDA y que por tanto el EHBSA, obtiene peores soluciones que el PLEDA en el mismo número de iteraciones. Sin embargo, dado que las ejecuciones han sido limitadas a 300 segundos y no a número de iteraciones, el EHBSA ha obtenido mejores resultados.

8 CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se recoge la planificación real seguida durante el transcurso del trabajo, una conclusión de los resultados, una reflexión personal y el trabajo futuro a abordar.

8.1 PLANIFICACIÓN SEGUIDA

La planificación inicial se describe en el Capítulo 3. Sin embargo, no ha sido posible mantener exáctamente esa planificación. Por tanto, la Tabla 20 muestra las diferencias habidas entre la estimación inicial y la realidad seguida.

Fase	Tarea	Comienzo estimado	Comienzo real	Dif.	Fin estimado	Fin real	Dif.
1	1	30/10/2015	30/10/2015	0	11/11/2015	11/11/2015	0
	2	30/10/2015	30/10/2015	0	29/11/2015	29/11/2015	0
	3	18/11/2015	20/11/2015	+2	22/11/2015	25/11/2015	+3
2	4	12/11/2015	15/11/2015	+3	26/11/2015	04/12/2015	+8
	5	20/11/2015	05/12/2015	+8	20/12/2015	10/01/2016	+21
	6	15/01/2016	31/01/2016	+16	04/02/2016	24/02/2016	+20
	7	05/02/2016	25/02/2016	+20	25/02/2016	16/03/2016	+20
3	8	25/02/2016	16/03/2016	+20	11/03/2016	20/03/2016	+9
	9	13/03/2016	20/03/2016	+7	04/04/2016	1/04/2016	-3
	10	05/04/2016	01/04/2016	-4	20/04/2016	20/04/2016	0
4	11	14/01/2016	14/01/2016	0	14/06/2016	16/06/2016	+2
	12	15/06/2016	-	-	01/07/2016	-	-
TOTAL días				+72			+80

Tabla 20: Planificación real.

Así mismo, se incluye el diagrama de *Gantt* real seguido durante el proyecto en la Figura 22 en contraposición con el inicial mostrado en la Figura 7.

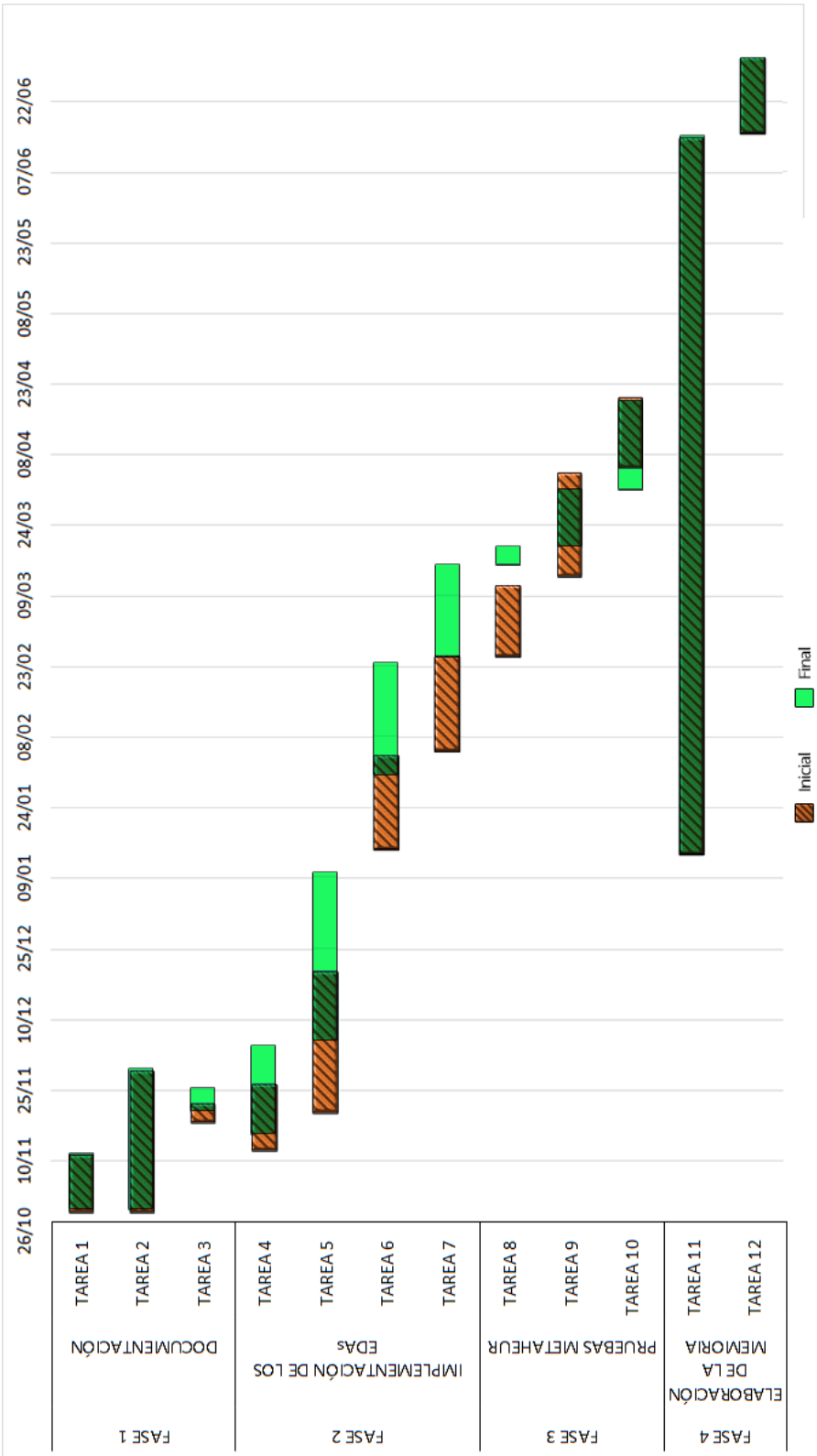


Figura 22: Diagrama de *Gantt* final.

8.2 CONCLUSIONES

En el transcurso del proyecto se han realizado numerosas ejecuciones (Capítulo 7) que han demostrado tanto la rapidez como la utilidad de los EDAs. Estos algoritmos tratan de resolver problemas de optimización que pueden surgir en muchos aspectos de la ingeniería, economía y medicina. El EBNA demuestra ser un algoritmo que obtiene las soluciones de forma más eficiente ya que la curva de optimización tiene mayor pendiente que la del UMDA. En cuanto al EHBSA, dado que su modelo probabilístico es más sencillo que el del PLEDA, realiza muchas más iteraciones. Sin embargo, el PLEDA obtiene mejores resultados en un número menor de iteraciones. Por otro lado, cabe destacar que para la evaluación de un algoritmo, hay que tener en cuenta tanto el tiempo de ejecución como el número de evaluaciones que ha realizado, ya que ambas variables son decisivas a la hora de resolver un problema de optimización.

En cuanto al lenguaje de programación R, hay que destacar que la curva de aprendizaje es costosa ya que dista bastante de la programación orientada a objetos. No obstante, una vez se domina el lenguaje, es muy sencillo programar en él. Además, la facilidad con la que se tratan las matrices y las listas ha ayudado enormemente a desarrollar este proyecto. Por otro lado, la velocidad con la que se realizan las operaciones entre este tipo de objetos es muy superior a la de otros lenguajes de programación que no vectorizan las operaciones. Asimismo, la gran cantidad de funciones genéricas y los paquetes desarrollados por terceros han hecho que ciertas tareas como la de obtener todo el espacio de permutaciones de tamaño n haya sido mucho más fácil.

En cuanto a *metaheuR* hay que decir que el paquete es muy completo y robusto. La ayuda interna que incluye hace que entender cualquier función del paquete sea una tarea sencilla. Además, no ha habido ningún inconveniente a la hora de utilizar ninguna función ya que siempre han habido ejemplos de uso en la propia ayuda del paquete.

Por último, el trabajo realizado ha supuesto un reto ya que pertenece al ámbito de ciencias de la computación mientras que el grado tiene estrecha relación con la ingeniería del software. Por ello, el estudio de artículos, tesis doctorales y libros ha sido un reto que ha merecido enormemente la pena superar. No obstante, el desarrollo de métodos y funciones bajo el esquema de programación científica ha conllevado gran dificultad. Esto es debido a que las técnicas de depuración difieren de las comunes. En estos casos, las funciones devuelven un resultado pero es necesario asegurarse de que sea el correcto. Además, debido al fundamento matemático sobre el que se apoyan estas, es bastante complicado encontrar el error.

8.3 TRABAJO FUTURO

El paquete *metaheuR* está operativo pero no terminado. Aun hay que incluir nuevos problemas de optimización, algoritmos e incluso utilidades para representar gráficos de ejecuciones. Además, para mejorar la comodidad del usuario, se podría crear una interfaz gráfica para que no sea necesario conocer el lenguaje R para utilizar este paquete.

A DIAGRAMAS DE SECUENCIA

Este apéndice recoge los diagramas de secuencia comentados en el Capítulo 5.

A.1 CLASE BAYESIANNETWORK

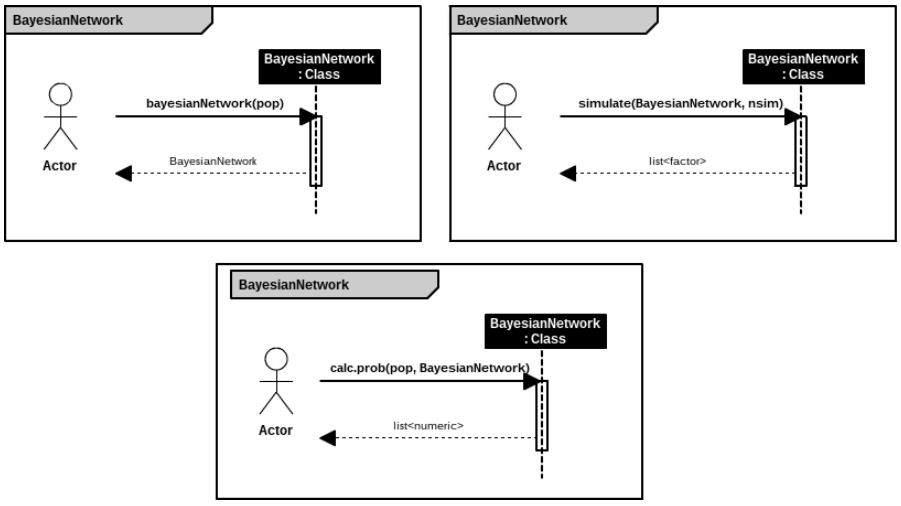


Figura 23: Diagrama de secuencia de la clase *BayesianNetwork*.

A.2 CLASE EHBSA

Como se ha explicado a lo largo del documento, esta clase únicamente incluye los métodos de aprendizaje y muestreo ya que no es posible obtener mediante una ecuación cerrada la probabilidad de un individuo dado el modelo.

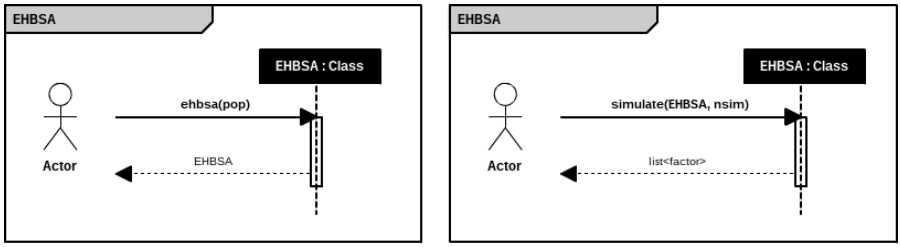


Figura 24: Diagrama de secuencia de la clase *EHBSA*.

A.3 CLASE PLACKETTLUCE

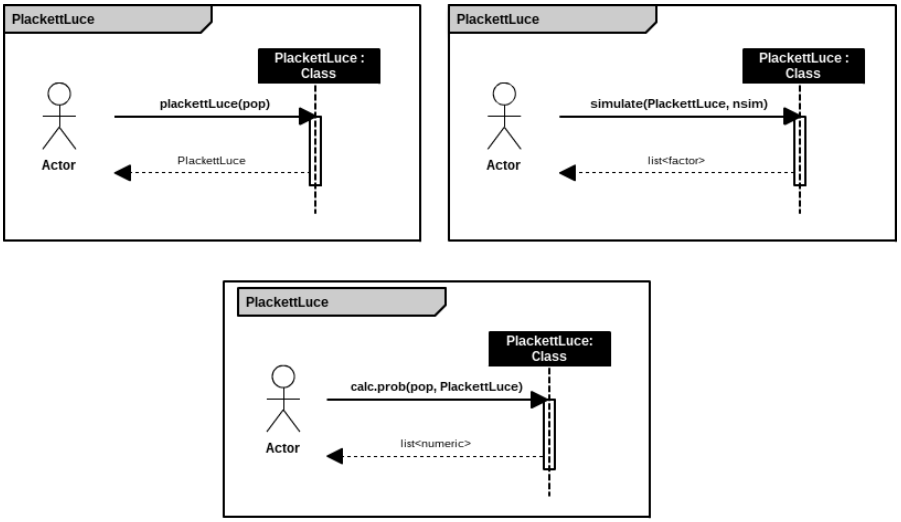


Figura 25: Diagrama de secuencia de la clase *PlackettLuce*.

B FUNCIONES IMPLEMENTADAS

En este apéndice a modo de extensión se muestran los ejemplos de programación implementados en el transcurso del documento. Es importante tener en cuenta que toda la librería referente tanto a lo desarrollado en el TFG como su totalidad se encuentra en la siguiente dirección: https://github.com/b0rxa/metaheuR.

B.1 CLASES IMPLEMENTADAS

En esta sección se muestran las clases implementadas en este proyecto que corresponden a los 3 modelos, *BayesianNetwork* (EBNA), EHBSA y PlackettLuce.

B.1.1 BayesianNetwork

Esta clase es la encargada del modelo EBNA explicado a lo largo del documento.

Código 8: BayesianNetwork.R

```
#' An S4 class to represent distributions based on Bayesian
    networks
#'
#' @slot network An object of the type bn.fit (see bnLearn
    pacakge for details)
#' @slot factor.population A logical value which determine if
    the learned population was factor or not.
#'

if (!require("bnlearn")) {
  message("This function requires the bnlearn package. It can
      be installed running the command install.packages('
      bnlearn ').")
  ans <- readline(prompt="Do you want me to install it now (Y/
      N, default N)?")
  if (ans=="y" | ans=="Y") {
    install.packages("bnlearn")
  }
}
setClass(
        Class="BayesianNetwork",
        representation=representation(network="bn.fit", factor
            .population="logical")
)
```

```
setValidity(
   Class="BayesianNetwork",
   method=function(object) {
     return (TRUE)
})


# GENERIC METHODS
    ------------------------------------------------------------
setMethod(
        f="simulate",
        signature="BayesianNetwork",
        definition=function(object, nsim=1, seed=NULL, ...) {
          #print("SIMULATE START")
          if(!is.null(seed)) {
            set.seed(seed)
          }
          aux <- rbn(object@network, nsim, debug=F)
          aux <- as.list(as.data.frame(t(aux)))
          if(object@factor.population){
            # We must return the same type of population which
                was given before
      for(i in 1:length(aux)){
        for(j in 1:length(aux[[i]])){
          if(is.na(aux[[i]][j])){
            aux[[i]][j] <- sample(levels(aux[[i]]), 1)
          }
        }
      }
          }else{
            aux <- lapply(aux, FUN=cat2bin)
            for(i in 1:length(aux)){
              for(j in 1:length(aux[[i]])){
                if(is.na(aux[[i]][j])){
                  aux[[i]][j] <- runif(1) >= 0.5
                }
              }
            }
          }

          return(aux)
        })

# CONSTRUCTOR
    ------------------------------------------------------------


#' Constructor of EBNA model
#'
#' This function creates an object of class \code{\linkS4class
    {BayesianNetwork}}
#'
#' @family EDA
#' @param data The dataframe containing the initial-population
     to construct the bayesian network
#' @param ... Ignored
```

```
#' @return An object of class \code{\linkS4class{
    BayesianNetwork}} that includes the Bayesian Network of
    the given data
#'
bayesianNetwork <- function(data, ...) {
        if(class(data) == "list"){
          if(class(data[[1]]) == "logical"){
            # The model is going to learn from a logical
                population
            factor.population <- F # is not a factor
                population
            aux <- lapply(data, FUN=bin2cat)
            pop.cat.df <- data.frame(t(matrix(unlist(aux),
                nrow = length(aux[[1]]), byrow = F)),
                stringsAsFactors = T)
            names(pop.cat.df) <- paste("X", 1:length(pop.cat.
                df), sep = "")
            for(i in 1:length(pop.cat.df)){
              pop.cat.df[[i]] <- factor(x=pop.cat.df[[i]],
                  levels=c("T","F"))
            }
          }else if(class(data[[1]]) == "factor"){
            factor.population <- T
            pop.cat.df <- data.frame(t(matrix(unlist(data),
                nrow = length(data[[1]]), byrow=F)),
                stringsAsFactors = T)
          }
                network <- hc(x = pop.cat.df)
                network <- bn.fit(network, data=pop.cat.df)
                obj <- new("BayesianNetwork", network =
                    network, factor.population = factor.
                    population)
        }else{
                stop("The data must be a list")
        }
}

#' Converts a vector from categorical to logical.
#' @param catVector categorical vector to be converted.
#' @return a logical vector
cat2bin <- function(catVector){
  return(catVector == "T")
}

#' Converts a vector from logical to categorical.
#' @param bin.v logical vector to be converted.
#' @return a categorical vector
bin2cat <- function(bin.v){
  chr.v <- rep("F", length(bin.v))
  chr.v[bin.v] <- "T"
  catVector <- factor(chr.v, levels=c("T", "F"))
  return(catVector)
}

#' Calculates the probabilities of the given model for the
    especified population.
#' @param object object of the class BayesianNetwork.
```

```
#' @param data population of individuals whoose probabilities
    are desired to know.
#' @return a vector containing the probabilities.
#'
calc.prob <- function(object, data){
  if(class(object) == "BayesianNetwork"){
    aux <- lapply(data, FUN=bin2cat)
    pop.cat.df <- data.frame(t(matrix(unlist(aux), nrow =
        length(aux[[1]]), byrow = F)), stringsAsFactors = T)
    names(pop.cat.df) <- paste("X", 1:length(pop.cat.df), sep
        = "")
    for(i in 1:length(pop.cat.df)){
      pop.cat.df[[i]] <- factor(x=pop.cat.df[[i]], levels=c("T
          ","F"))
    }
    logverosimilitud <- logLik(object@network, pop.cat.df, by.
        sample=TRUE)
    return(exp(logverosimilitud))
  }else{
    stop("The model must be of the class BayesianNetwork")
  }
}
```

B.1.2 EHBSA

Esta clase es la encargada del modelo probabilístico EHBSA que se ha descrito a lo largo del documento.

Código 9: EHBSA.R

```
#' An S4 class to represent distributions based on Edge
    Histrogam Based Sampling Algorithm.
#'
#' @slot adjacencyMatrix A matrix containing the second order
    marginal probabilities from the data.
#'
setClass(
        Class="EHBSA",
        representation=representation(adjacencyMatrix="matrix"
            )
)

# GENERIC METHODS
    ------------------------------------------------------------
setMethod(
  f="simulate",
  signature="EHBSA",
  definition=function(object, nsim=1, seed=NULL, ...) {
    # we create a roullette wheel
    new.population <- matrix(, nrow=0, ncol = length(
        object@adjacencyMatrix[1,]))
    for(q in 1:nsim){
      l.matrix <- object@adjacencyMatrix
      l <- c(sample(0, length(l.matrix[1,]), replace=TRUE)) #
          new list with numbers
      l[1] <- round(runif(1, 1, length(l.matrix[1,])), 0)
```

```
      l.matrix[, l[1]] <- c(0)
      for(i in 2:length(l.matrix[1,])){ # iterate positions of
          the new individual
        val <- runif(1, 0, sum(l.matrix[l[i-1],]))
        j <- 1
        acc <- l.matrix[l[i-1], j]
        #acc <- 0
        while(val > acc){
          j <- j + 1
          acc <- acc + l.matrix[l[i-1],j]
        }
        l[i] <- j
        l.matrix[, j] <- c(0)
      }
      l <- permutation(l)
      new.population <- c(new.population, l)
    }
    return(new.population)
})

# CONSTRUCTOR
    ----------------------------------------------------------------

#' Constructor of EHBSA model
#'
#' This function creates an object of class \code{\linkS4class
    {EHBSA}}
#'
#' @family EDA
#' @param data The dataframe containing the initial-population
     to construct the EHBSA.
#' @param bias ratio to control the preassure of the learned
    probabilities. By default it is set to 0.5
#' @param ... Ignored
#' @return An object of class \code{\linkS4class{EHBSA}} that
    includes the adjacency matrix of the given data
#'
ehbsa <- function(data, bratio, ...) {
        if(class(data) == "list"){
          if(missing(bratio)){
            bratio <- 0.5
          }
                # We can create the object
          # Learn matrix definition
          adjacencyMatrix <- matrix(c(0) , nrow=length(data
              [[1]]), ncol=length(data[[1]])) #empty matrix
            for(i in 1:length(data)){
              for(j in 2:length(data[[i]]@permutation)){
                adjacencyMatrix[ data[[i]]@permutation[j
                    -1], data[[i]]@permutation[j] ] <-
                    adjacencyMatrix[ data[[i]]@permutation
                    [j-1], data[[i]]@permutation[j] ] + 1
              }
            }
          colnames(adjacencyMatrix) <- paste("X", 1:length(
              adjacencyMatrix[1,]), sep="")
```

```
            rownames(adjacencyMatrix) <- paste("X", 1:length(
                adjacencyMatrix[1,]), sep="")
            adjacencyMatrix <- adjacencyMatrix + bratio
            diag(adjacencyMatrix) <- 0
            obj <- new("EHBSA", adjacencyMatrix =
                adjacencyMatrix)
          }else{
                stop("The data must be a list")
          }
}
```

B.1.3 PlackettLuce

Esta clase es la encargada del modelo probabilístico PLEDA que se ha descrito a lo largo del documento.

Código 10: PlackettLuce.R

```
#' An S4 class to represent distributions based on Plackett
    Luce distribution model.
#'
#' @slot parameters The weight parameters of the learned
    population.
#'

setClass(
  Class="PlackettLuce",
  representation=representation(parameters="numeric")
)
# GENERIC METHODS
    ------------------------------------------------------------
setMethod(
  f="simulate",
  signature="PlackettLuce",
  definition=function(object, nsim=1, seed=NULL, ...) {
    new.population <- c()
    parameters <- object@parameters # create a copy to
        manipulate it
    # Iterate through nsim
    for (k in 1:nsim){
      # Create a random number for roulette wheel
      ac <- 0
      rnd <- 0
      parameters <- object@parameters # create a copy to
          manipulate it
      l <- sample(x=0, size=length(parameters), replace = T)
      for(i in 1:length(parameters)){
        rnd <- runif(1, 0, sum(parameters))
        ac <- parameters[1] # get the first position
        j <- 1 # position of the parameters vector
        while(ac <= rnd){
          j <- j + 1
          ac <- ac + parameters[j]
        }
        l[i] <- j
        parameters[j] <- 0
```

```
      }
      l <- permutation(l)
      new.population <- c(new.population, l)
    }
    return(new.population)
  })

# CONSTRUCTOR
    ----------------------------------------------------------------


#' This function creates an object of class \code{\linkS4class
    {PlackettLuce}}
#'
#' @family EDA
#' @param data The dataframe containing the initial-population
     to contruct the Plackett-Luce model.
#' @param maxIter Optional. Maximum iterations for the MM
    algorithm.
#' @param ... Ignored
#' @return An object of class \code{\linkS4class{PlackettLuce
    }} that includes the weight parameter vector of the given
    data
#'
plackettLuce <- function(data, maxIter, ...) {


  # This code is based on Hunter's script of Matlab
  if(missing(maxIter)){
    maxIter <- 10000
  }
  if(class(data) == "list"){
    data <- c(data, permutation(rev(data[[1]]@permutation)))
     # We can create the object
    # Transform data to a matrix
    P <- length(data[[1]]@permutation) # Problem size
    N <- length(data) # Pop size
    M <- P # Problem size
    f <- matrix(data = 0, nrow=P, ncol=N) # f(i,j) = one who
        placed i in contenst j
    r <- matrix(data = 0, nrow=M, ncol=N) # r(i,j) = place of
        i in contest j, modified so that

    for(i in 1:N){
      ind <- data[[i]]@permutation
      for(j in 1:M){
        f[j,i] <- ind[j]
        r[ind[j], i] <- j + P*(i-1)
      }
    }
    r2 <- r

    w <- matrix(data = 0, nrow = M, ncol = 1) # w(i) = # times
         i placed higher than last
    pp <- rep(x=M, N)
    for(i in 1:N){
      tmp <- f[1:(pp[i]-1), i]
```

```
      w[tmp] <- 1 + w[tmp]
    }
    pp<- pp + seq(0,((N-1)*P),by = P)
    gamma <- matrix(1,M,1)
    dgamma <- matrix(1,M,1)
    iterations <- 0
    while((norm(x=dgamma ,type="2") > 1e-09) && iterations <=
        maxIter){
      iterations <- iterations + 1
      g <- f
      g[f>0] <- gamma[f[f>0]]
      aux <- g
      for(i in (P-1):1){
        for(j in 1:N){
          aux[i,j] <- g[i,j]+aux[i+1,j]
        }
      }
      g <- aux

      g[P,] <- 0

      g[g>0] <- 1./g[g>0]
      # At this point, g(i,j) should be the reciprocal of the
          sum of gamma's
      # for places i and higher in the  jth contest exceppt
          for i=lastplace.

      g <- getColCumsum(g)
      # Now g(i,j) should be the sum of all the denominators
          for ith place in the jth contest

      r2[r>0] <- g[r[r>0]]
      sr2 <- c(unlist(lapply(1:length(r2[,1]), FUN=function(i)
          {
                return(sum(r2[i,]))
      })))
      newgamma <- w/sr2
      dgamma <- newgamma - gamma
      gamma <- newgamma
    }
    params <- gamma/sum(gamma)
    obj <- new("PlackettLuce", parameters = c(params))
  }else{
    stop("The data must be a list")
  }
}

#' Acumulate the matrix values by columns
#' @param m matrix
#' @return acumulated matrix
getColCumsum <- function(m){
  for(i in 2:(length(m[,1]))){
    m[i,] <- m[(i-1),] + m[i,]
  }
  return(m)
}
```

```
#' Calculates the probabilities of the given model for the
    population especified.
#' @param model object of the class PlackettLuce.
#' @param data population whoose probabilities are desired to
    know.
calc.prob <- function(model, data){
  if(class(model) != "PlackettLuce"){
    stop("The model must be of class PlackettLuce")
  }else{
    result <- list()
    for(i in 1:length(data)){
      individual <- data[[i]]@permutation
      prob <- 1
      for(j in 1:(length(individual)-1)){
        if(j > 1){
          prob <- prob * (model@parameters[individual[j]] /
              sum(model@parameters[-appeared]))
          appeared <- c(appeared, individual[j])
        }else{
          prob <- prob * (model@parameters[individual[j]] /
              sum(model@parameters))
          appeared <- individual[j]
        }
      }
      appeared <- 0
      result <- c(result, prob)
    }
  }
  return(unlist(result))
}
```

B.2 EXPERIMENTOS REALIZADOS

Esta sección expone el código de los experimentos realizados durante el Capítulo 7.

B.2.1 Experimentos EBNA

Código 11: Experimento 1 - EBNA

```
cat("*
    ----------------------------------------------------------
    *\n")
cat("*                    Experiment 1: EBNA
                        *\n")
cat("*
    ----------------------------------------------------------
    *\n")
cat("Description: Checking if population data structure \n")
cat("interferes in learned Bayesian network:\n")
```

```
cat("
    ------------------------------------------------------------\
    n")
cat("Author: Ander Carreño López - ander.carreno@ikasle.ehu.
    eus\n")
cat("
    ------------------------------------------------------------\
    n\n\n")
# We create a random population
# Random knapsack problem
library(metaheuR)

n <- 10
w <- runif(n)
l <- sum(w[runif(n) > 0.66])
v <- w*3 + runif(n)

knap.problem <- knapsackProblem(weight=w, limit=l, value=v)

# Initial population
pop.size <- 10
pop.1 <- lapply(1:pop.size,
                FUN=function(i){
                  return(knap.problem$correct(runif(n)>0.5))
                })
pop.2 <- lapply(pop.1, FUN=bin2cat)

# Learn Bayesian Networks of each population
network.1 <- bayesianNetwork(pop.1)
network.2 <- bayesianNetwork(pop.2)

cat("\nFirst Bayesian Network:\n")
cat("
    ------------------------------------------------------------\
    n")
print(network.1@network)
cat("\n\nSecond Bayesian Network:\n")
cat("
    ------------------------------------------------------------\
    n")
print(network.2@network)
```

Código 12: Experimento 2 - EBNA

```
cat("*
    ------------------------------------------------------------
    *\n")
cat("*                    Experiment 2: EBNA
                        *\n")
cat("*
    ------------------------------------------------------------
    *\n")
cat("Description: Checking if learning probabilities are
    correct \n")
cat("
    ------------------------------------------------------------\
    n")
```

```
cat("Author: Ander Carreno Lopez - ander.carreno@ikasle.ehu.
    eus\n")
cat("
    ------------------------------------------------------------\
    n\n\n")
# We create a random population
# Random knapsack problem
library(metaheuR)

n <- 10
w <- runif(n)
l <- sum(w[runif(n) > 0.66])
v <- w*3 + runif(n)

pop.size <- 1000
percentage.of.eq <- 0.75

pop<- lapply(1:(pop.size*percentage.of.eq),
             FUN=function(i){
               return(rep(FALSE, n))
             })

# We add random permutations to the initial population
pop <- c(pop,
         lapply(1:(pop.size*(1 - percentage.of.eq)),
                FUN=function(i){
                  return(knap.problem$correct(runif(n)
                      >0.5))
                }
         )
)

# Build the Bayesian Network
network <- bayesianNetwork(pop)

print(network@network)
```

Código 13: Experimento 3 - EBNA

```
# Experiment 3: EBNA
# author: Ander Carreño López - ander.carreno@ikasle.ehu.eus

# We create a random population
# Random knapsack problem
cat("*
    ------------------------------------------------------------
    *\n")
cat("*                    Experiment 3: EBNA
                      *\n")
cat("*
    ------------------------------------------------------------
    *\n")
cat("Description: Checking extraction of probabilities from \n
    ")
cat("learned Bayesian network: \n")
```

```
cat("
    ------------------------------------------------------------\
    n")
cat("Author: Ander Carreno Lopez - ander.carreno@ikasle.ehu.
    eus\n")
cat("
    ------------------------------------------------------------\
    n\n\n")
library(metaheuR)
n <- 10
w <- runif(n)
l <- sum(w[runif(n) > 0.66])
v <- w*3 + runif(n)

pop.size <- 1000
knap.problem <- knapsackProblem(w,v,l)
pop<- lapply(1:pop.size,
             FUN=function(i){
               return(rep(FALSE, n))
             })

# We add random permutations to the initial population
pop <- c(pop,
         lapply(1:(pop.size),
                FUN=function(i){
                  return(knap.problem$correct(runif(n)>0.5))
                }
         )
)

# Build the Bayesian Network
network <- bayesianNetwork(pop)

# Get the probabilities
unique.cases <- unique(pop)
probabilities <- calc.prob(network, unique.cases)

# Just for formatting
p <- lapply(unique.cases, FUN=bin2cat)
p <- matrix(unlist(p), ncol = n, byrow = T)
aux <- apply(p, 1, paste, collapse="-")
df <- data.frame('Individuals'=aux, 'Probabilities'=
    probabilities)

print(df)
```

Código 14: Experimento 4 - EBNA

```
cat("*
    ------------------------------------------------------------
    *\n")
cat("*                    Experiment 4: EBNA
                        *\n")
cat("*
    ------------------------------------------------------------
    *\n")
```

```
cat("Description: Checking Kullback-Leibler divergence between
    \n")
cat("two learned Bayesian networks:\n")
cat("
    ------------------------------------------------------------\
    n")
cat("Author: Ander Carreno Lopez - ander.carreno@ikasle.ehu.
    eus\n")
cat("
    ------------------------------------------------------------\
    n\n\n")
# We create a random population
# Random knapsack problem
library(stringr)
library(metaheuR)
n <- 10
w <- runif(n)
l <- sum(w[runif(n) > 0.66])
v <- w*3 + runif(n)
times <- 10
inc.ratio <- 0.60 # increment ratio of each iteration
pop.size <- 50000
pop.sizes <- c(pop.size)

knap.problem <- knapsackProblem(weight=w, limit=l, value=v)

# Get the complete search space
all.space <- lapply(0:((2^n)-1), FUN=intToBitVect)
all.space <- lapply(all.space, FUN=numberToLogical)

divergences <- list()

#iterate times times
for(t in 1:times){
  cat("-->Starting itertion ", t, "\nCurrent population size:
      ", pop.size, "\n")
  # Initial population
  i.pop <- lapply(1:pop.size,
                  FUN=function(i){
                    return(knap.problem$correct(runif(n)>0.5))
                  })

  # Build the initial Bayesian Network
  i.network <- bayesianNetwork(i.pop)

  # We simulate once and learn from that simulation
  f.pop <- simulate(i.network, nsim=pop.size)
  f.network <- bayesianNetwork(f.pop)

  # Get the divergence
  divergence <- divergencia.kullbackLeibler(i.network, f.
      network, all.space)

  # Add to the divergences array
  divergences <- c(divergences, divergence)
  cat("The divergence between two models is: ", divergence, "\
      n\n")
```

```
  # increment the pop.size for the next iteration
  pop.size <- round(pop.size + (pop.size * inc.ratio),0)
  # Store it
  pop.sizes <- c(pop.sizes, pop.size)
}

results.df <- data.frame('Size of the population'=pop.sizes[-
    length(pop.sizes)], 'Divergences'=unlist(divergences,
    recursive=T))
results.df <- data.frame(num=1, results.df)
# Export to csv
write.table(results.df, file="Experimentos/resultados.csv",
    append = T, sep = ",")
cat(results.df)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Auxiliary functions
library(stringr)
intToBitVect <- function(x){
  options(scipen=999)
  bsum<-0
  bexp<-1
  while (x > 0) {
    digit<-x %%2
    x<-floor(x / 2)
    bsum<-bsum + digit * bexp
    bexp<-bexp * 10
  }
  return(bsum)
}

numberToLogical <- function(x){
  x.string <- toString(x)
  x.string <- str_pad(x.string, width=n, side="left", pad="0")
  x.string <- strsplit(x.string, "")
  ls <- list()
  for(number in x.string){
    ls<-c(ls, number!="0")
  }
  return(unlist(ls))
}
```

B.2.2 Experimentos EHBSA

Código 15: Experimento 1 - EHBSA

```
cat("*
    ----------------------------------------------------------
    *\n")
cat("*                    Experiment 1: EHBSA
                        *\n")
cat("*
    ----------------------------------------------------------
    *\n")
cat("Description: Checking learning of the algorithm \n")
```

```
cat("
    ------------------------------------------------------------------\
    n")
cat("Author: Ander Carreno Lopez - ander.carreno@ikasle.ehu.
    eus\n")
cat("
    ------------------------------------------------------------------\
    n\n\n")

# Default parameters for the experiment
library(metaheuR)
n <- 10

# We create a new user defined population
permu.i.pop <- lapply(1:50, FUN=function(i){
                        return(c(1:n))
                      })
aux <- permu.i.pop
permu.i.pop <- c(aux, lapply(1:70,
                      FUN=function(i){
                        return(c(n:1))
                      })
                 )
# Learn
ehbsa.model <- ehbsa(permu.i.pop, bratio = 0)

# We print the learned adjacency matrix
cat("Learned adjacency matrix:\n")
cat("
    ------------------------------------------------------------------\
    n")
print(ehbsa.model@adjacencyMatrix)
```

Código 16: Experimento 2 - EHBSA

```
cat("*
    ------------------------------------------------------------------
    *\n")
cat("*                    Experiment 2: EHBSA
                      *\n")
cat("*
    ------------------------------------------------------------------
    *\n")
cat("Description: Checking sampling of the algorithm \n")
cat("
    ------------------------------------------------------------------\
    n")
cat("Author: Ander Carreño López - ander.carreno@ikasle.ehu.
    eus\n")
cat("
    ------------------------------------------------------------------\
    n\n\n")

# Default parameters
library(metaheuR)
set.seed(10)
```

```
n <- 10
pop.size <- 1000

# We generate a random population
permu.i.pop <- lapply(1:pop.size,
                      FUN=function(i){
                        return(sample(1:n))
                      })
# Learn a model
ehbsa.model <- ehbsa(permu.i.pop)
# We overwrite existing matrix with one we have chosen
m <- matrix(c(0,0,0,100,0,0,0,0,0,0,
              100,0,0,0,0,0,0,0,0,0,
              0,100,0,0,0,0,0,0,0,0,
              0,0,0,0,100,0,0,0,0,0,
              0,0,0,0,0,100,0,0,0,0,
              0,0,0,0,0,0,100,0,0,0,
              0,0,0,0,0,0,0,100,0,0,
              0,0,0,0,0,0,0,0,100,0,
              0,0,0,0,0,0,0,0,0,100,
              0,0,0,0,0,0,0,0,0, 100), ncol=10, nrow=10, byrow
                  =T)
ehbsa.model@adjacencyMatrix <- m
# We sample a 100 cases and we check if match with desired
    permutation
chk.permu <- c(3,2,1,4,5,6,7,8,9,10)
chk.permu.collapsed <- paste(chk.permu, collapse="")
ok <- 0
for(i in 1:100){
  aux <- simulate(ehbsa.model)
  if(paste(aux, collapse ="") == chk.permu.collapsed){
    ok <- ok + 1
  }else{
    if(sum(aux)!=55)
      cat(aux , "\t", i , "\n")
  }
}

cat("Checked permutation is: ", chk.permu, "\n")
cat("Percentaje of accuracy: ", ok, "%")
```

Código 17: Experimento 3 - EHBSA

```
cat("*
    ------------------------------------------------------------
    *\n")
cat("*                    Experiment 3: EHBSA
                        *\n")
cat("*
    ------------------------------------------------------------
    *\n")
cat("Description: Checking sampling of the algorithm \n")
cat("
    ------------------------------------------------------------\
    n")
cat("Author: Ander Carreno Lopez - ander.carreno@ikasle.ehu.
    eus\n")
```

```
cat("
    ------------------------------------------------------------\
    n\n\n")

# Default parameters
library(metaheuR)

n <- 10
pop.size <- 1000

# We generate a random population
permu.i.pop <- lapply(1:pop.size,
                      FUN=function(i){
                        return(sample(1:n))
                      })

# Learn a model
ehbsa.model <- ehbsa(permu.i.pop)

# We overwrite existing matrix with one we have chosen
m <- matrix(c(0,0,5,70,5,1,1,3,5,10,
              85,0,0,0,0,0,0,0,0,15,
              2,90,0,0,0,2,2,2,2,0,
              0,0,0,0,100,0,0,0,0,0,
              0,0,0,1,0,99,0,0,0,0,
              3,0,0,0,0,0,97,0,0,0,
              0,0,10,0,0,0,0,90,0,0,
              0,0,3,0,0,2,0,0,95,0,
              0,0,1,0,0,0,0,0,0,99,
              97,0,3,0,0,0,0,0,0, 0), ncol=10, nrow=10, byrow=
                  T)


# We manualy add the bias ratio
m <- m + 0.5

ehbsa.model@adjacencyMatrix <- m

# We sample a 100 cases and we check if match with desired
    permutation
chk.permu <- c(3,2,1,4,5,6,7,8,9,10)
chk.permu.collapsed <- paste(chk.permu, collapse="")
ok <- 0
for(i in 1:100){
  aux <- simulate(ehbsa.model)
  if(paste(aux, collapse ="") == chk.permu.collapsed){
    ok <- ok + 1
  }else{
    if(sum(aux)!=55)
      cat(aux , "\t", i , "\n")
  }
}

cat("Checked permutation is: ", chk.permu, "\n")
cat("Percentaje of accuracy: ", ok, "%")
```

B.2.3 Experimentos Plackett-Luce

Código 18: Experimento 1 - Plackett-Luce

```
cat("*
    ------------------------------------------------------------------
    *\n")
cat("*                        Experiment 2: PlackettLuce
                  *\n")
cat("*
    ------------------------------------------------------------------
    *\n")
cat("Description: Checking Kullback-Leibler sampling \n")
cat("
    ------------------------------------------------------------------\
    n")
cat("Author: Ander Carreño López - ander.carreno@ikasle.ehu.
    eus\n")
cat("
    ------------------------------------------------------------------\
    n\n\n")

require(metaheuR)
require(gtools)

# Initial parameters
set.seed(1)
n <- 5
pop.size <- 100
percentaje.of.equal <- 0.75

pop <- lapply(1:(pop.size*percentaje.of.equal), FUN=function(i
    ){
  return(identityPermutation(n))
})

pop <- c(pop, lapply(1:(pop.size*(1-percentaje.of.equal)), FUN
     = function(i){
 return(randomPermutation(n))
}))

# Learn a PlackettLuce Model
model <- plackettLuce(pop)

print(model@parameters)

# identity.permutation <- paste(n:1, collapse="")
# ok <- 0
#
# # Sample the model once
# for(i in 1:10000){
#
#   new.individual <- simulate(model, 1)
#   if(sum(unlist(new.individual)) != 15){
#     print(i)
#     print(new.individual)
#   }
```

```
#   new.individual <- paste(unlist(new.individual), collapse
    ="")
#   if(new.individual == identity.permutation){
#     ok <- ok + 1
#   }
# }
#
# cat("Percentaje of matched simulations: ", ok/100, "%")
```

Código 19: Experimento 2 - Plackett-Luce

```
# Auxiliar function
divergencia.kullbackLeibler <- function(model.1, model.2, data
    ){
  prob.1 <- calc.prob(model.1, data)
  prob.2 <- calc.prob(model.2, data)
  div<-0
  for (i in 1:length(data)) {
    if(!is.nan(prob.2[i]) & prob.2[i] != 0){
      div <- div + prob.1[i]*log(prob.1[i]/prob.2[i])
    }
  }
  return (div)
}


cat("*
    ------------------------------------------------------------
    *\n")
cat("*                    Experiment 2: PlackettLuce
                *\n")
cat("*
    ------------------------------------------------------------
    *\n")
cat("Description: Checking Kullback-Leibler divergence between
     \n")
cat("two learned models:\n")
cat("
    ------------------------------------------------------------\
    n")
cat("Author: Ander Carreno Lopez - ander.carreno@ikasle.ehu.
    eus\n")
cat("
    ------------------------------------------------------------\
    n\n\n")

require(metaheuR)
require(gtools)

# Initial parameters
set.seed(1)
n <- 5
pop.size <- 1000
times <- 10
inc.ratio <- 0.60 # increment ratio of each iteration
pop.sizes <- c(pop.size)
```

```
for(t in 1:times){
  cat("---->Starting itertion ", t, "\nCurrent population size:
      ", pop.size, "\n")
  # Generate a random permutation based population
  i.pop <- lapply(1:pop.size,
                  FUN=function(i){
                    return(sample(1:n))
                  })
  i.pop <- lapply(1:pop.size,
                  FUN=function(i){
                    return(c(1:n))
                  })
  i.pop <- c(i.pop, lapply(1:10, FUN=function(i){
    return(c(n:1))
  }))

  # Generate all permutation space with gtools
  all.space <- permutations(n=n, r=n, v=c(1:n))
  all.space <- lapply(1:length(all.space[,1]), FUN=function(i)
      {
    return(permutation(all.space[i,]))
  })

  # Learn a PlackettLuce Model
  model <- plackettLuce(i.pop)

  # Sample the model once
  sampled.pop <- simulate(model, nsim=pop.size)

  # Learn another model
  new.model <- plackettLuce(sampled.pop)

  # Get the divergence between both
  divergence <- divergencia.kullbackLeibler(model, new.model,
      all.space)

  cat("The divergence between two models is: ", divergence, "\
      n\n")
  pop.size <- round(pop.size + (pop.size * inc.ratio),0)
  pop.sizes <- c(pop.sizes, pop.size)
}

results.df <- data.frame('Size of the population'=pop.sizes[-
    length(pop.sizes)], 'Divergences'=unlist(divergences,
    recursive=T))
results.df <- data.frame(num=1, results.df)
```

B.2.4 Ejecuciones de los EDAs

Código 20: Ejecuciones sobre problemas de optimización

```
library(metaheuR)
library(igraph)

# Email parameters
```

```
library(mailR)

start.experiment <- date()


# UMDA Experiment
# EBNA Experiments
# Graph Coloring Problem
pop.sizes <- c(20, 40, 60, 80, 100)
times <- 5
umda.results <- list()
ebna.results <- list()
for(k in 1:length(pop.sizes)){
  n <- pop.sizes[k]
  possible.values <- paste("c", 1:n, sep="")
  graph <- random.graph.game(n, p.or.m=0.2)
  mis.problem <- misProblem(graph, penalization = 1/n)


  for(i in 1:times){
    set.seed(i)
    i.pop <- lapply(1:n,
                    FUN=function(i){
                      return(mis.problem$correct(runif(n)>0.5)
                          )
                    })
    parameters <- list()
    parameters$evaluate             <- mis.problem$evaluate
    parameters$initial.population   <- i.pop
    parameters$selectSubpopulation  <- elitistSelection
    parameters$selection.ratio      <- 0.5
    parameters$valid                <- mis.problem$valid
    parameters$resources            <- cResource(time=300)
    parameters$verbose              <- F
    parameters$non.valid            <- "correct"
    parameters$correct              <- mis.problem$correct


    parameters$learn <- univariateMarginals
    res.umda <- do.call(basicEda, parameters)
    umda.results <- c(umda.results, res.umda)
    parameters$learn <- bayesianNetwork
    res.ebna <- do.call(basicEda, parameters)
    ebna.results <- c(ebna.results, res.ebna)
  }
}
date.ebna <- date()


# EHBSA & PLEDA experiments
# TSP Problem from tsplib
problems <- c("Documentos/Minería de Datos/TFG/Ander R/
    Experimentos/Ejecuciones/TSPLib/burma14.xml",
              "Documentos/Minería de Datos/TFG/Ander R/
                  Experimentos/Ejecuciones/TSPLib/ulysses22.
                  xml",
```

```
                "Documentos/Minería de Datos/TFG/Ander R/
                    Experimentos/Ejecuciones/TSPLib/ftv33.xml",
                "Documentos/Minería de Datos/TFG/Ander R/
                    Experimentos/Ejecuciones/TSPLib/dantzig42.
                    xml",
                "Documentos/Minería de Datos/TFG/Ander R/
                    Experimentos/Ejecuciones/TSPLib/ft53.xml")
ehbsa.results <- list()
plEDA.results <- list()
for(k in 1:length(problems)){
  cost.matrix <- tsplibParser(problems[k])
  tsp.problem <- tspProblem(cost.matrix)
  for(i in 1:times){
    set.seed(i)
    i.pop <- lapply(1:length(cost.matrix[1,]), function(s){
      return(randomPermutation(length(cost.matrix[1,])))
    })
    parameters <- list()
    parameters$evaluate            <- tsp.problem$evaluate
    parameters$initial.population  <- i.pop
    parameters$selectSubpopulation <- elitistSelection
    parameters$selection.ratio     <- 0.5
    parameters$valid               <- tsp.problem$valid
    parameters$resources           <- cResource(time=300)
    parameters$verbose             <- F

    parameters$learn <- ehbsa
    res.ehbsa <- do.call(basicEda, parameters)
    ehbsa.results <- c(ehbsa.results, res.ehbsa)
    parameters$learn <- plackettLuce
    res.plEDA <- do.call(basicEda, parameters)
    plEDA.results <- c(plEDA.results, res.plEDA)
  }
}
date.ehbsa <- date()


########################################## EMAIL
    ########################################3
cuerpo <- paste("<HTML><h1>Resultados de las ejecuciones</h1>
    <b>", start.experiment, "</b><br><br><h4>EBNA:</h4><br>\t<
    b>Hora de fin: </b>", date.ebna ," <br>\t<b>Resultados:</b
    ><br>\t",paste(unlist(lapply(1:length(ebna.results), FUN=
    function(i){return(ebna.results[[i]]@evaluation)})), sep="
    ", collapse = "-"), " <br><h4>EHBSA</h4><br> \t<b>Hora de
    fin: </b>", date.ehbsa ," <br>\t<b>Resultados:</b><br>\t",
     paste(unlist(lapply(1:length(ehbsa.results), FUN=function
    (i){return(ehbsa.results[[i]]@evaluation)})), sep="",
    collapse = "-"), " <br><h4>Plackett Luce EDA</h4><br> \t<b
    >Hora de fin: </b><br>\t<b>Resultados:</b><br>\t", paste(
    unlist(lapply(1:length(plEDA.results), FUN=function(i){
    return(plEDA.results[[i]]@evaluation)})), sep="", collapse
     = "-"), " <br></HTML>", sep="")[1]

# Send Email
send.mail(from = "yourEmail@gmail.com",
          to = c("YourName <yourEmail@gmail.com>"),
```

```
        subject = "Resultados de ejecuciones",
        body = cuerpo,
        encoding = "utf-8",
        html = TRUE,
        inline = TRUE,
        smtp = list(host.name = "smtp.gmail.com", port =
            465, user.name = "yourUsername", passwd = "
            yourPassword", ssl = TRUE),
        authenticate = TRUE,
        send = TRUE)
```

B.3 KULLBACK-LEIBLER

En esta sección se expone el código utilizado para calcular la divergencia de Kullback Leibler mencionada en el Capítulo 7.

Código 21: Código divergencia *Kullback-Leibler* en R.

```
divergencia.kullbackLeibler <- function(network.1, network.2,
    data){
  prob.1 <- calc.prob(network.1, data)
        prob.2 <- calc.prob(network.2, data)
        div<-0
        for (i in 1:length(data)) {
          if(!is.nan(prob.2[i]) & prob.2[i] != 0){
            div <- div + prob.1[i]*log(prob.1[i]/prob.2[i])
          }
        }
return (div)
}
```

BIBLIOGRAFÍA

Akaike, Hirotugu

1974 "A new look at the statistical model identification", *Automatic Control, IEEE Transactions on*, 19, 6, págs. 716-723.

Bäck, Thomas

1996 "Evolutionary algorithms in theory and practice".

Bengoetxea, Endika, Pedro Larrañaga, Isabelle Bloch, Aymeric Perchant y Claudia Boeres

2002 "Inexact graph matching by means of estimation of distribution algorithms", *Pattern Recognition*, 35, 12, págs. 2867-2880.

Bradley, Ralph Allan y Milton E Terry

1952 "Rank analysis of incomplete block designs: I. The method of paired comparisons", *Biometrika*, 39, 3/4, págs. 324-345.

Calvo, Borja, Josu Ceberio y Usue Mori

2015 *Bilaketa Heuristikoak Teoria eta Adibide Praktikoak*, Euskal Herriko Unibertsitatea (UPV-EHU), págs. 82-86.

Ceberio, Josu

2014 *Solving Permutation Problems with Estimation of Distribution Algorithms and Extensions Thereof*, Tesis doct., PhD thesis, Faculty of Computer Science, University of the Basque Country.

Ceberio, Josu., Mendiburu, Alexander y Jose A Lozano

2013 "The Plackett-Luce ranking model on permutation-based optimization problems", en *Evolutionary Computation (CEC), 2013 IEEE Congress on*, IEEE, págs. 494-501.

Chambers, John

2008 *Software for data analysis: programming with R*, Springer Science & Business Media.

Chickering, David M, Dan Geiger, David Heckerman y col.

1994 *Learning Bayesian networks is NP-hard*, inf. téc., Citeseer.

Etxeberria, Ramon y Pedro Larrañaga

1999 "Global optimization using Bayesian networks", en *Second Symposium on Artificial Intelligence (CIMAF-99)*, Habana, Cuba, págs. 332-339.

Etxeberria, Ramon, Pedro Larrañaga y Juan M Picaza

1997 "Analysis of the behaviour of genetic algorithms when learning Bayesian network structure from data", *Pattern Recognition Letters*, 18, 11, págs. 1269-1273.

Goldberg, David E

2000 *Genetic Algorithms–In Search, Optimization & Machine Learning, Revised ed*.

Goldberg, David E y Robert Lingle

1985 "Alleles, loci, and the traveling salesman problem", en *Proceedings of the first international conference on genetic algorithms and their applications*, Lawrence Erlbaum Associates, Publishers, págs. 154-159.

Guiver, John y Edward Snelson

2009 "Bayesian inference for Plackett-Luce ranking models", en *proceedings of the 26th annual international conference on machine learning*, ACM, págs. 377-384.

Holland, John H

1975 *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press.

Hunter, David R

2004 "MM algorithms for generalized Bradley-Terry models", *Annals of Statistics*, págs. 384-406.

Koopmans, Tjalling C y Martin Beckmann

1957 "Assignment problems and the location of economic activities", *Econometrica: journal of the Econometric Society*, págs. 53-76.

Kullback, Solomon y Richard A Leibler

1951 "On information and sufficiency", *The annals of mathematical statistics*, 22, 1, págs. 79-86.

Kullback-Leibler divergence s.f. , https://en.wikipedia.org/wiki/Kullback-Leibler_divergence, Accessed: 2016-05-26.

Larrañaga, P, CMH Kuijpers, RH Murga e Y Yurramendi

1996 "Searching for the best ordering in the structure learning of Bayesian networks", *IEEE Transactions on Systems, Man and Cybernetics*, 26, 4, págs. 487-493.

Larrañaga, Pedro y Jose A Lozano

2002 *Estimation of distribution algorithms: A new tool for evolutionary computation*, Springer Science & Business Media, vol. 2.

Larrañaga, Pedro, Mikel Poza, Yosu Yurramendi, Roberto H Murga y Cindy MH Kuijpers

1996 "Structure learning of Bayesian networks by genetic algorithms: A performance analysis of control parameters", *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 18, 9, págs. 912-926.

Leeuwen, Jan

1990 *Handbook of theoretical computer science: algorithms and complexity*, Elsevier, vol. 1.

Lozano, Jose A

2006 *Towards a new evolutionary computation: advances on estimation of distribution algorithms*, Springer Science & Business Media, vol. 192.

Mühlenbein, Heinz y Gerhard Paass

1996 "From recombination of genes to the estimation of distributions I. Binary parameters", en *Parallel Problem Solving from Nature—PPSN IV*, Springer, págs. 178-187.

Pelikan, Martin, David E Goldberg y E Cantú-Paz

2000 "Hierarchical Problem Solving and the Bayesian Optimization Algorithm.", en *GECCO*, págs. 267-274.

Pelikan, Martin y Heinz Mühlenbein

1998 "Marginal distributions in evolutionary algorithms", en *Proceedings of the International Conference on Genetic Algorithms Mendel*, Citeseer, vol. 98, págs. 90-95.

Pelikan, Martin, Kumara Sastry y Erick Cantú-Paz

2007 *Scalable optimization via probabilistic modeling: From algorithms to applications*, Springer, vol. 33.

Scutari, Marco

2009 "Learning Bayesian networks with the bnlearn R package", *arXiv preprint arXiv:0908.3817*.

Tsutsui, Shigeyoshi

2002 "Probabilistic model-building genetic algorithms in permutation representation domain using edge histogram", en *Parallel Problem Solving from Nature—PPSN VII*, Springer, págs. 224-233.

Tsutsui, Shigeyoshi, Martin Pelikan y David E Goldberg

2003 "Using edge histogram models to solve permutation problems with probabilistic model-building genetic algorithms", *IlliGAL Report*, 2003022.