UNIVERSIDAD REY JUAN CARLOS

INGENIERÍA INFÓRMATICA

Curso Académico 2003/2004

Proyecto de Fin de Carrera

Algunos Algoritmos Evolutivos Basados en Clases de Equivalencia para la Inducción de Redes Bayesianas a partir de Datos

Autor: Enrique López Ponce
Tutor: Jorge Muruzábal

10 de Septiembre del 2004

Índice

Resumen

Las *Redes Bayesianas (RB)* constituyen una clase de modelos gráficos muy utilizada para reflejar la estructura de dependencia en un conjunto de variables discretas. Uno de los problemas básicos asociados a este tipo de modelos consiste en la inducción de la estructura gráfica óptima a partir de una matriz de observaciones de las variables de interés. Es sabido que la solución exacta de este problema es computacionalmente difícil y a menudo se emplean técnicas heurísticas para su resolución. Tales técnicas difieren, entre otros aspectos, en la representación que se elija para las soluciones del problema. La representación más frecuente está basada en *Grafos Dirigidos Acíclicos*(o *DAGs* en inglés). Como dos *DAGs* diferentes pueden codificar en realidad el mismo modelo, una representación alternativa se basa en *Grafos Parcialmente Dirigidos Acíclicos Completados* (o *CPDAGs*).

Los Algoritmos Evolutivos constituyen un método general de optimización que requiere mínimas hipótesis sobre el espacio de búsqueda. En este proyecto se consideran Algoritmos Evolutivos basados en CPDAGs para atacar el problema de inducción de estructura. Se implementan para ello operadores de mutación disponibles en la literatura así como algunos operadores novedosos. En el proyecto se evalúa el mérito de estos nuevos operadores y se contrastan los resultados obtenidos con los nuevos Algoritmos Evolutivos con relación a otros métodos inductivos. En nuestro caso utilizaremos el método K2.

Capítulo 1

Introducción

1.1 Redes Bayesianas

El formalismo de las Redes Bayesianas es un modelo teórico procedente de la teoría de probabilidades que pretende modelizar el conocimiento bajo incertidumbre que caracteriza el razonamiento humano en forma gráfica, crear modelos gráficos que intenten representar ese conocimiento a través de la estructura que podrían tener.

Se representará el dominio del problema como un conjunto de variables aleatorias con dominios discretos y predefinidos. De esta forma cada variable aleatoria tendrá asociada una distribución de probabilidad, que asigna probabilidades a cada uno de los posibles valores de dicha variable. Además existirá información acerca de cómo influyen unas variables en otras. Por este motivo las Redes Bayesianas también son conocidas como *Redes Causales*. Para el estudio de esta primera parte se ha consultado [1].

Un ejemplo de una pequeña Red Bayesiana puede verse en la figura 1.1 (Pág. 5), en la cual se consideran las variables aleatorias que describen un problema relacionado con el estudio de la posibilidad de que la hierba esté mojada o no:

- *Nublado*: Puede ser verdad o no. Dominio[F,T].
- *Aspersor*: Puede estar encendido o no. Dominio[F,T].
- *Lluvia*: Puede caer o no. Dominio[F,T].
- *Hierba Mojada*: Puede estarlo o no. Dominio[F,T].

Se estima que existe una relación de dependencia entre las variables Nublado y Aspersor, Nublado y Lluvia, Asperor y Hierba Mojada y Lluvia y Hierba Mojada, y a su vez Hierba Mojada dependerá a través de Aspersor y Lluvia del valor que posea Nublado. Para expresar dicha relación de dependencia se tiene que representar la información de los elementos de la red: *nodos y arcos.*

En una Red Bayesiana, cada nodo se corresponde con una variable, que a su vez representa una entidad del mundo real. Los arcos que unen los nodos indican

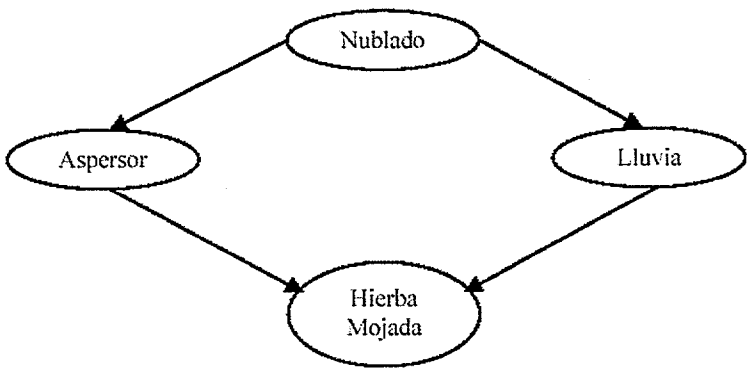

Figura 1.1: Red Bayesiana Simple.

las relaciones de influencia causal entre dichas variables. Estas relaciones entre nodos nos conducen a los conceptos de *padre e hijo* en la red.

Es decir, todos aquellos nodos a los que no les llegue ningún arco serán nodos padre y tendrán asociados a ellos un vector fila de **probabilidades marginales**, que asignará una probabilidad a cada uno de los posibles valores que puede tomar la variable.

Las probabilidades marginales de Nublado aparecen en la tabla 1.1 (Pág. 5).

Nublado	**False**	**True**
Conj. Carac	0.5	0.5

Tabla 1.1: Probabilidad Marginal de Nublado.

Es decir, existe la misma posibilidad de que esté nublado como de que no lo esté. En este caso queda clara la absoluta ignorancia acerca del hecho, ya que se asignan probabilidades idénticas a todos los posibles valores de la distribución. La suma de todos los valores de dicho vector fila debe ser 1. Los nodos hijos, es decir, todos aquellos a los que les lleguen arcos de uno o más padres, tendrán asociadas a ellos **matrices de probabilidad condicionada**.

Como ejemplo se tienen las matrices de probabilidad condicionada de las variables Aspersor, Lluvia y Hierba Mojada, representadas en las tablas 1.2, 1.3 y 1.4 respectivamente (Pág. 6).

En general se puede decir que los nodos de una red pueden tener uno o más padres y a su vez ser padres de uno o más hijos. En el caso de un nodo con más de un padre, cada matriz de probabilidad condicionada representará la relación de todos los padres con ese nodo. Por ejemplo:

La matriz de probabilidades condicionadas relativa a cada hijo tendría las siguientes propiedades:

Aspersor	**False**	**True**
False	0.5	0.5
True	0.9	0.1

Tabla 1.2: Probabilidad Marginal de Aspersor.

Lluvia	**False**	**True**
False	0.8	0.2
True	0.2	0.8

Tabla 1.3: Probabilidad Marginal de Lluvia.

Hierba mojada	**False**	**True**
False False	1	0
True False	0.1	0.9
False True	0.1	0.9
True True	0.01	0.99

Tabla 1.4: Probabilidad Marginal de Hierba Mojada.

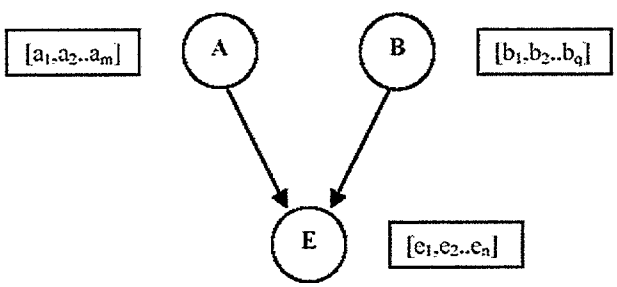

Figura 1.2: Esquema de un nodo con más de un padre.

- *Tantas filas como posibles valores combinados tengan las causas.* Es decir, el producto del número de valores o estados de cada uno de sus padres. En el ejemplo anterior, la matriz de probabilidad condicionada de Hierba Mojada tendrá tantas filas como valores de Aspersor multiplicado por los valores de su otro padre, Lluvia. Es decir $2 * 2 = 4$ filas.
- *Tantas columnas como posibles valores o estados tiene el efecto.*
- *La suma de los elementos de cada fila será 1.*

La matriz de probabilidad condicionada del ejemplo de la figura 1.2 se podría expresar como:

$$\mathbf{M_{(E/A,B)}} = \begin{pmatrix} P(E_1/A_1,B_1) & P(E_2/A_1,B_1) & \dots & P(E_n/A_1,B_1) \\ P(E_1/A_1,B_2) & P(E_2/A_1,B_2) & \dots & P(E_n/A_1,B_2) \\ \vdots & \vdots & \ddots & \vdots \\ P(E_1/A_m,B_q) & P(E_2/A_m,B_q) & \dots & P(E_n/A_m,B_q) \end{pmatrix}$$

Dos sucesos son independientes sí y sólo sí $P(A/B) = P(A)$. De manera análoga, indicamos que dos sucesos son dependientes si no se da la igualdad.

Los arcos de una Red Bayesiana expresan precisamente este tipo de dependencia entre las dos variables que conectan.

La *independencia condicional* (o separabilidad lineal) entre dos sucesos extiende este concepto para explicar la relación entre dos sucesos que, si bien no son independientes, pueden expresar su condicionamiento basándose en un tercer suceso. Decimos que X es condicionalmente independiente de W dado Y si se cumple la igualdad:

$$P(X/Y,W) = P(X/Y) \tag{1.1}$$

Esta igualdad implica que toda la influencia que pueda ejercer W sobre X se canaliza a través de la variable Y. En la figura 1.3 (Pág. 8) se observa cómo se puede representar dicha influencia gráficamente.

Esta figura expresa que cualquiera de los nodos W_i es *linealmente separable* de X mediante Y. O dicho de otra forma, que X es *condicionalmente independiente* de cualquiera de los nodos W_i respecto de Y. En una Red Bayesiana, cualquier pareja de nodos que tenga un camino entre ellos es condicionalmente independiente respecto de algún nodo intermedio, aunque no se puede decir que sean independientes.

1.1.1 Definiciones

A la luz de los conceptos generales presentados informalmente en el apartado anterior, se procede a formularlos de manera más precisa a continuación.

Un **arco dirigido** es un par ordenado (x, y) que se escribe como $x \rightarrow y$. Por el contrario, un **arco no dirigido** o no orientado es un par no ordenado (x, y) cuya notación es $x - y$.

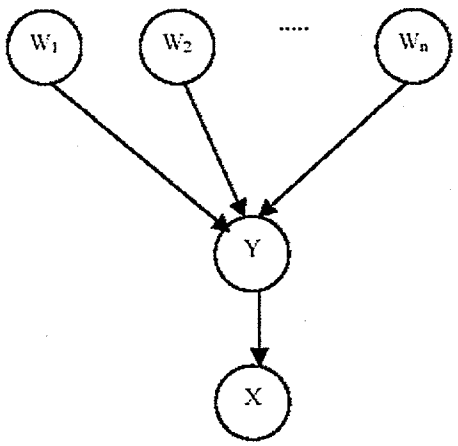

Figura 1.3: X es independientemente condicional de W_i dado Y.

Sea el par $G = (N, A)$ tal que N es un conjunto de nodos y A es el conjunto de arcos dirigidos definidos sobre dichos nodos, se dice que G es un **grafo dirigido**.

Sea el par $\bar{G} = (\bar{N}, \bar{A})$ en el cual $\bar{N}$ es un conjunto de nodos y $\bar{A}$ el conjunto de arcos no orientados definidos sobre tales nodos, se dice que $\bar{G}$ es un **grafo no dirigido**.

Dado un grafo $G = (N, A)$ se define un **camino** en G como una secuencia ordenada de nodos $(x_1, \cdots x_r), x_i \in N$ tal que $\forall j, 1 \leq j < r$, se verifica que el arco $x_j \rightarrow x_{j+1} \in A$ o el arco $x_{j+1} \rightarrow x_j \in A$.

Se habla de **camino dirigido** cuando dado G se tenga una secuencia ordenada de nodos $(x_1, \ldots, x_r), x_i \in N$ tal que $\forall j, 1 \leq j < r$ se verifica que el arco $x_j \rightarrow x_{j+1} \in A$.

Un **camino no dirigido** en G será una secuencia ordenada de nodos como $(x_1, \ldots, x_r)$, $x_i \in N$ tal que $\forall j, 1 \leq j < r$ el arco $x_j - x_{j+1} \in A$.

Sea un grafo $G = (N, A)$, se puede decir que **ciclo** es un camino $(x_1, \ldots, x_r)$, $x_i \in N$, tal que $x_1 = x_r$. Un **ciclo dirigido** es un camino dirigido $(x_1, \ldots, x_r)$, $x_i \in N$, tal que $x_1 = x_r$.

Un **grafo dirigido acíclico** o *DAG* (Directed Acyclic Graph) es un grafo dirigido que no contiene ciclos.

Sea $G = (N, A)$ un grafo dirigido, donde x e $y \in N$, decimos que x es **padre** de y o y es **hijo** de x sí y sólo sí existe un arco dirigido de x a y, es decir, $x \rightarrow y$.

Si además $z \in N$, entonces x será **antepasado** de z sí y sólo sí existe un camino dirigido de x a z.

Del mismo modo, z será un **descendiente** de x sí y sólo sí x es antepasado de z.

Dado un grafo no dirigido $\bar{G} = (\bar{N}, \bar{A})$ tal que x e $y \in N$, x será **vecino** de y

o y vecino de x sí y sólo sí existe un arco no dirigido entre ellos, $x - y$ o $y - x$.

Sea G un grafo dirigido acíclico (DAG), definimos el **orden topológico** de los nodos de G como cualquier orden parcial de dichos nodos tal que para cualquier par x e $y \in N$ si x es antecesor de y entonces x debe preceder a y en el orden.

Dadas tres variables X, Y e Z, diremos que Z **separa condicionalmente** a X e Y si X e Y son independientes dado Z.

Un **ejemplo** o **muestra** es un vector que contiene el valor o estado de cada nodo de la Red Bayesiana en un momento dado. Por tanto la longitud del vector será igual que el número de nodos de la red.

Definición formal de Red Bayesiana

Una Red Bayesiana es:

- Un conjunto de variables V.
- Un conjunto E de relaciones binarias definidas sobre las variables de V.
- Una distribución de probabilidad conjunta θ definida sobre las variables de V, tales que:
  - (V, E) es un grafo acíclico y dirigido G.
  - (G, θ) cumple las hipótesis de independencia condicional, también llamadas de *separación direccional.*

Inferencia

La utilidad de este modelo radica en que se puede realizar inferencia sobre la red. Su base teórica principal es la probabilidad condicionada y los Teoremas de Bayes y de la Probabilidad Total.

A continuación se pasará a definir brevemente estos conceptos antes de introducirse en los métodos de inferencia.

- *Probabilidad*
  Dado un experimento aleatorio, con espacio muestral Ω y álgebra de sucesos asociada Q, se define la *probabilidad* como una aplicación del álgebra de sucesos Q en el intervalo $[0,1]$, es decir, $P : Q \rightarrow [0,1]$, que satisface los tres axiomas siguientes:
  1. $P(A) \geq 0, \forall A \in Q$.
  2. $P(\Omega) = 1$.
  3. $P(A \cup B) = P(A) + P(B)$ sí $A, B \in Q$ con $A \cap B = \emptyset$.
- *Probabilidad Condicionada*
  Dado un suceso $A \in Q$ con $P(A) > 0$, para cualquier otro suceso $B \in Q$, definimos la *probabilidad del suceso B condicionada al suceso A* como:

$$P(B/A) = P(A \cap B)/P(A) \tag{1.2}$$

Dados dos sucesos $A, B \in Q$ dirán que son independientes sí:

$$P(B/A) = P(B) \rightarrow P(A \cap B) = P(A) * P(B) \tag{1.3}$$

- *Teorema de Probabilidad Total*
  Considera un conjunto de sucesos $\{A_i\}_i = 1 \ldots n, A_i \in Q$ tal que se cumple:

  1. $\bigcup_{i=1}^{n} A_i = \Omega$
  2. $A_i \cap A_j = \emptyset, \forall (i,j), 1 \leq i,j \leq n, i \neq j$

  Es decir, la unión de todos ellos es el suceso seguro y son incompatibles dos a dos. Un conjunto de sucesos con estas dos probabilidades recibe el nombre de *sistema completo de sucesos.*

  Sea un suceso cualesquiera $B \in Q$ y un sistema completo de sucesos $\{A_i\}_i = 1 \ldots n$ tal que $P(A_i) > 0 \forall i$, el Teorema de Probabilidad Total establece que:

$$P(B) = \sum_{i=1}^{n} P(A_i * P(B/A_i) \tag{1.4}$$

  Es decir, si el suceso B puede ocurrir por alguna de las causas A_i, la probabilidad de que ocurra es la suma de las probabilidades de las causas $(P(A_i))$ por la probabilidad del suceso B condicionado a la causa A_i, $(P(B/A_i))$.

$$\forall i, P(B \cap A_i) = P(A_i) * P(B/A_i) \tag{1.5}$$

- *Teorema de Bayes*
  El Teorema de Bayes se apoya en el proceso inverso al que hemos visto en el Teorema de la Probabilidad Total. Es decir, en el Teorema de la Probabilidad Total a partir de las probabilidades del suceso A deduce la probabilidad del suceso B. Mientras que en el Teorema de Bayes a partir de que ha ocurrido el suceso B deducimos las probabilidades del suceso A.

  La fórmula del Teorema de Bayes es:

$$P(A_j/B) = \frac{(P(A_j) * P(B/A_j))}{(\sum P(A_i) * P(B/A_i)) = P(A_j \cap B)/P(B)} \tag{1.6}$$

  Pero por otra parte tenemos que:

$$P(B/A_j) = \frac{P(A_j \cap B)}{P(A_j)} \rightarrow P(A_j \cap B) = P(B/A_j) * P(A_j) \tag{1.7}$$

$$P(A_j/B) = \frac{P(A_j \cap B)}{P(B)} = P(B/A_j) * P(A_j)/P(B) \qquad (1.8)$$

Los métodos de inferencia en una Red Bayesiana más importantes son:

- *Aprendizaje*
  Con este método se pretende que la red aprenda o establezca las probabilidades de sus nodos, ya sean marginales (si son nodos raiz) o condicionadas (si son nodos hoja). Se parte de que se conoce la estructura de la red y además una lista de patrones o muestras. En esta lista, cada patrón tendrá como longitud el número de nodos de la red, y para cada nodo se tendrá asignado un valor o estado, que va a ser el que determine las probabilidades de la red en dicho aprendizaje. La estimación de los parámetros dado el grafo es muy sencilla.

- *Verosimilitud*
  La verosimilitud o *likelihood* de un ejemplo para una Red Bayesiana es la probabilidad de que cada nodo de la Red Bayesiana tome el valor que se indica en el ejemplo. La probabilidad condicionada de un nodo estará influenciada por sus padres. La verosimilitud dependerá, por tanto, de la estructura de la red y del contenido de la lista de patrones.

- *Verosimilitud Marginal*
  La verosimilitud marginal o *Marginal likelihood*, es el producto de las verosimilitudes marginales de cada nodo. La diferencia principal con la anterior verosimilitud consiste en que ahora se integran los parámetros.

- *Criterio de Información Bayesiano (BIC)*
  Este criterio es lo que comúnmente se conoce como *BIC* [2] y se define como:
  $$BIC = L - 0.5 * d * \lg(N) \qquad (1.9)$$
  Donde L es la verosimilitud de la red en función de la lista de patrones, d es el número de parámetros de la red y N es el número total de patrones. El método *BIC* no requiere distribuciones a priori.

- *Muestreo*
  El muestreo es la obtención de ejemplos a partir de una Red Bayesiana. Es decir, poseemos la estructura de la red y conocemos las probabilidades tanto marginales como condicionada de los nodos, y a partir de esa información queremos hallar la posible lista de patrones a través de la cual se puede obtener dicha red por medio del aprendizaje.

Posibles Algoritmos de Búsqueda

Dentro de las diferentes posibilidades que presentes para afrontar el problema, el trabajo se centrará en los algoritmos con los que se harán las pruebas, el algoritmo K2 [5] y los Algoritmos Evolutivos. Más adelante se explicará el funcionamiento de cada uno, estudiando y comparando pros y contras de ambos enfoques.

1.2 Clases de Equivalencia

Las *clases de equivalencia*, como se explica más adelante, representan la posibilidad de equiparar varias redes en función de la distribución de probabilidad sobre las variables de las mismas. Esto supone la posibilidad de asociar varias redes, reduciendo la búsqueda de un individuo cualquiera a la busqueda de la clase a la que pertenece ese individuo.

1.2.1 Introducción

Podemos afirmar que existen dos problemas generales para las *clases de equivalencia:*

1. El criterio de puntuación a la hora de evaluar las redes.
2. El problema de reconocer la estructura de dichas redes.

En el primer problema se necesitará un criterio de puntuación para una red dados unos datos. Es decir, dada una estructura y un conjunto de datos, ¿cómo asignar la puntuación que cuantifique la bondad de la estructura para esos datos?

En el segundo problema, se buscarán una o más redes que produzcan el mismo valor para el criterio de evaluación.

Es importante en ambos problemas la noción de equivalencia:

Dos redes son ***equivalentes*** *si la distribución de probabilidad sobre las variables que representa una de esas redes es idéntica a la distribución que representa la otra.*

Es decir, diferentes redes con diferentes topologías pueden representar la misma distribución. Como es evidente, la equivalencia es reflexiva, simétrica y transitiva.

Muchos criterios de puntuación usados para el aprendizaje de estructuras de Redes Bayesianas son de puntuación equivalente. Es decir, estos criterios no distinguen entre las redes que son equivalentes, ya que asignan la misma puntuación a estructuras equivalentes. En este proyecto se utilizará un criterio de puntuación equivalente junto con un algoritmo heurístico de búsqueda *greedy* o voraz, lo cual producirá una mejor búsqueda que utilizando otros métodos.

Dos grafos $G = (V, E)$ y $\bar{G} = (\bar{V}, \bar{E})$ son equivalentes si para cada Red Bayesiana $B = (G, \theta)$ existe otra red $\bar{B} = (\bar{G}, \bar{\theta})$ tal que B y $\bar{B}$ definen la misma distribución de probabilidad y viceversa.

Los métodos típicos de aprendizaje de estructuras de Redes Bayesianas se dividen en dos componentes.

El primer componente es un *criterio de puntuación* que tiene como entrada una estructura de Red Bayesiana, un conjunto de datos y posiblemente algún dominio conocido. El criterio de puntuación devuelve un valor indicando en que medida se ajusta a la estructura de datos.

El segundo componente es un *algoritmo de búsqueda* que identifica a una o más estructuras con la mayor puntuación. Una vez que la estructura de la Red Bayesiana ha sido identificada es usual propagar los parámetros de ese modelo. Dado un criterio de puntuación, la finalidad del procedimiento de búsqueda es identificar uno o más modelos con la mejor puntuación.

Uno de los principales problemas que se nos presenta es que muchos operadores usados por los algoritmos de búsqueda están definidos para moverse entre *DAGs* sin considerar las clases de equivalencia a las que pertenecen.

Por ejemplo, vamos a considerar la figura 1.4, en la cual se muestran algunos *DAGs* definidos sobre un dominio de tres variables. Se asume que se ha aplicado un algoritmo de búsqueda *greedy* a este espacio, comenzando por el grafo vacío de la figura 1.4a y establecemos que los operadores usados por el algoritmo van a ser: insertar, borrar e invertir arco. Podemos observar que los grafos de las figuras 1.4b y 1.4c son equivalentes.

Sin embargo, un algoritmo *greedy* no distinguiría un grafo de otro, por lo que el algoritmo elegiría uno arbitrariamente. Si se opta por la figura 1.4b, el algoritmo en un paso (insertar arco $z \rightarrow y$) llega al grafo de la figura 1.4d, mientras que si se elige la figura 1.4c, se necesitan dos movimientos - invertir arco $y \rightarrow x$ e insertar arco $z \rightarrow y$ - para alcanzar la figura 1.4d.

Desafortunadamente, la puntuación del grafo dada por un algoritmo de búsqueda greedy puede ser sensible a estas preferencias arbitrarias.

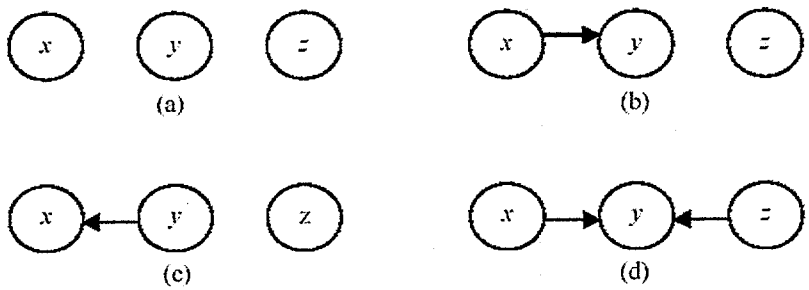

Figura 1.4: Ejemplo de un algoritmo de búsqueda *greedy* aplicado a un *DAG*

1.2.2 Nociones Generales

A continuacion se definirán algunos conceptos necesarios para concretar nuestro espacio de búsqueda.

El **esqueleto** de cualquier *DAG* es un grafo no dirigido resultante de ignorar las direcciones de cada arco. Una **V estructura** en un *DAG* G es una tupla ordenada de nodos (x, y, z) tal que:

1. G contiene los arcos x — y y z — y

2. (x, z) no son adyacentes en G.

Teorema 1 *Dos DAGs son equivalentes sí y sólo sí tienen el mismo esqueleto y las mismas* V estructuras.

Un arco dirigido $x \rightarrow y$ es **forzado** en G si para cada *DAG* $\bar{G}$ equivalente a G, $x \rightarrow y$ existe en $\bar{G}$. Para cualquier arco e en G, si e no es forzado en G, entonces es **reversible** en G, es decir, existen algunos *DAGs* $\bar{G}$ equivalentes a G en los cuales un arco e puede tener direcciones opuestas.

Una consecuencia del Teorema 1 es que para cada arco e participante en una V estructura en algún *DAG* G, si ese arco es reversible en algún otro *DAG* $\bar{G}$, entonces G y $\bar{G}$ no son equivalentes. De este modo, cualquier arco participante en una V estructura es forzado. Sin embargo, no todos los arcos forzados participan necesariamente en una V estructura. Por ejemplo, en la figura 1.5, el arco $z \rightarrow w$ es forzado y no participa en la V estructura (x, z, y).

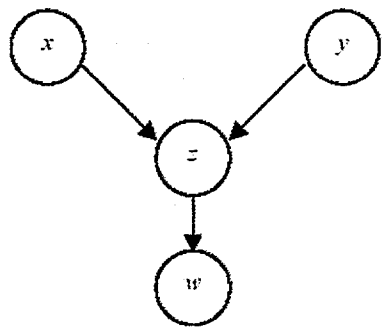

Figura 1.5: Ejemplo de *DAG* que contiene un arco forzado y no participa en la V estructura.

Un **Grafo Acíclico Parcialmente Dirigido**, conocido como **PDAG**, es un grafo que contiene arcos dirigidos y no dirigidos. Se utilizarán los *PDAGs* para representar las clases de equivalencias de una Red Bayesiana.

Se denota P como un *PDAG* arbitrario, y de define la clase de equivalencia del *DAG* correspondiente a P (denotado como *Class(P)*) como sigue:

$G \in$ *Class(P)* sí y sólo sí G y P tienen el mismo esqueleto y las mismas V estructuras.

Por el Teorema 1 decimos que un *PDAG* que contiene un arco dirigido para cada arco participante en una V estructura, y un arco no dirigido para cada otro arco, identifica una única clase de equivalencia del *DAG*.

Si un *DAG* G tiene el mismo esqueleto y el mismo conjunto de V estructuras que un *PDAG* P, y si cada arco dirigido en P tiene la misma orientación en G, decimos que G es una **extensión consistente** de P. Obsérvese que todos los *DAGs* que son extensión consistente de P están contenidos en *Class(P)*, pero no todos los *DAGs* de *Class(P)* son extensiones consistentes de P.

Si existe al menos una extensión consistente del *PDAG* P, decimos que P admite una *extensión consistente*. La figura 1.6.a muestra un *PDAG* que admite una extensión consistente, la figura 1.6.b muestra una posible extensión consistente de la anterior y la figura 1.6.c muestra un *PDAG* que no admite una extensión consistente.

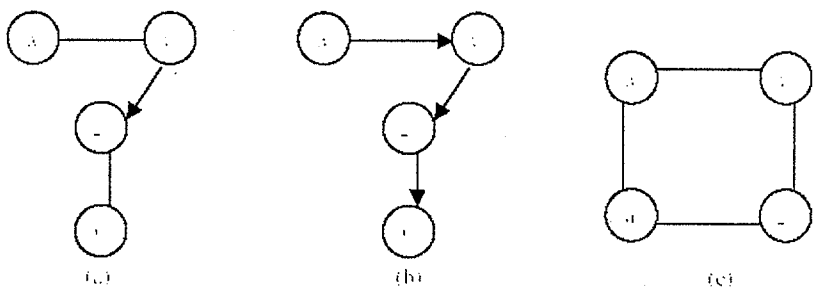

Figura 1.6: (a) *PDAG* que admite una extensión consistente, (b) es una extensión consistente del *PDAG* de (a), y (c) *PDAG* que no admite una extensión consistente.

Un **Grafo Acíclico Parcialmente Dirigido completado, CPDAG**, que corresponde a una clase de equivalencia es el *PDAG* que contiene un arco dirigido por cada arco forzado de la clase de equivalencia, y un arco no dirigido por cada arco reversible de la clase de equivalencia. Observar que para un *PDAG* completado P^c, distinto de un *PDAG*, cada *DAG* contenido en *Class*(P^c) es una extensión consistente de P^c. En la figura 1.7 (página 15) se muestra un ejemplo.

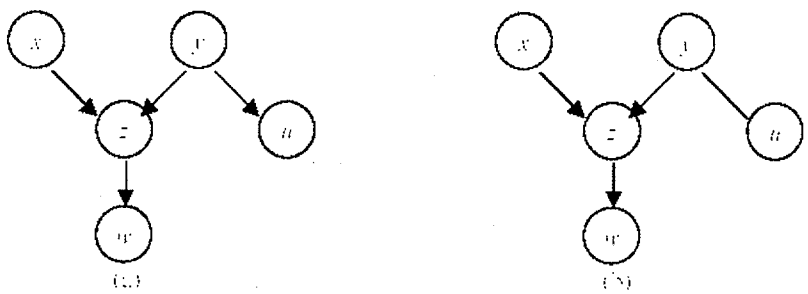

Figura 1.7: (a) *DAG* G y (b) *PDAG* completado de *Class(G)*.

Dada una clase de equivalencia para una Red Bayesiana, el *PDAG comple-*

tado para esa clase de equivalencia es único, lo cual conduce al siguiente lema.

Lema 1 *Sean* P_1^c *y* P_2^c *dos* PDAGs completados *que admiten una extensión consistente. Por lo tanto,* $P_1^c = P_2^c$ *sí y sólo sí* $\text{Class}(P_1^c) = \text{Class}(P_2^c)$.

Un **cliqué** en un *DAG* o un *PDAG* es un conjunto de nodos en el cual cada par de nodos es adyacente. Es decir, si x e y pertenecen al *cliqué*, entonces x es adyacente a y e y es adyacente a x.

También se puede hablar de **cliqué de arcos no dirigidos**, en el cual, los nodos son adyacentes en alguna dirección. Es decir, si x e y pertenecen al *cliqué*, entonces x es adyacente a y o y es adyacente a x.

Se utilizará la notación Π_x para referirse al conjunto de **padres** del nodo x, tanto para *DAG* como para *PDAG*. Se usará N_x para nombrar al conjunto de **vecinos** del nodo x en un *PDAG*. Además se llamará $N_{x,y} = N_x \cap N_y$ al conjunto de **vecinos comunes** de x e y en un *PDAG* y se dice que $\Omega_{x,y} = \Pi_x \cap N_y$ es el conjunto de **padres del nodo x que son vecinos del nodo y** en un *PDAG*.

1.2.3 Algoritmos de Conversión

En este apartado se describen los algoritmos *Grafo Acíclico Dirigido - Grafo Acíclico Parcialmente Dirigido completado (DAG-CPDAG)* y *Grafo Acíclico Parcialmente Dirigido - Grafo Acíclico Dirigido (PDAG-DAG)*, que son necesarios para aplicar los operadores de búsqueda que se presentarán en la siguiente sección.

Grafo Acíclico Dirigido - Grafo Acíclico Parcialmente Dirigido Completado (DAG-CPDAG)

Este algoritmo toma como entrada una Red Bayesiana y produce un *PDAG* completado representante de la clase de equivalencia a la que pertenecía el *DAG*.

Se han realizado varios estudios que se pueden usar para la implementación. Una de las alternativas es la ofrecida por Chickering [3], la cual es computacionalmente eficiente y asintóticamente óptima. El algoritmo etiqueta todos los arcos del *DAG* como forzados o reversibles. Una vez realizado el etiquetado, es trivial la construcción del correspondiente *PDAG* completado.

El primer paso del algoritmo será realizar una ordenación total de los arcos del *DAG* (ORDER_EDGES(G)). A continuación se realizará el etiquetado sobre los arcos ya ordenados (*LABEL_EDGES(G)*). Por último, recorremos el *DAG* y cambiamos los arcos forzados por arcos dirigidos y los reversibles por no dirigidos en el *PDAG* resultante.

Algoritmo ORDER_EDGES(G)
Entrada: *DAG G*
Salida: *DAG G* con un orden total sobre los arcos

1. Ordenar topológicamente los nodos en G

2. Asignar $i = 0$
3. **Mientras** existan arcos desordenados en G **hacer**
   (a) Seleccionar y que será el menor nodo ordenado que tenga un arco desordenado en él
   (b) Seleccionar el mayor nodo ordenado, x, para el cual $x \rightarrow y$ no esté ordenado
   (c) Etiquetar $x \rightarrow y$ con orden i
   (d) $i = i + 1$

Algoritmo LABEL_EDGES(G)
Entrada: *DAG G*
Salida: *DAG G* en el cual cada arco está etiquetado como *forzado* o *reversible*

1. Utilizar el algoritmo **Order_Edges***(G)* para ordenar los arcos
2. Etiquetar cada arco en G como *desconocido*
3. **Mientras** existan arcos etiquetados como *desconocido* en G **hacer**
4. Tomar el menor arco ordenado etiquetado como *desconocido* $x \rightarrow y$
5. **Para** cada arco $w \rightarrow x$ etiquetado *forzado* **hacer**
6. **Si** w no es padre de y **entonces**
7. Etiquetar $x \rightarrow y$ y todos los arcos incidentes en y con *forzado*
8. **Goto** 3
9. **En otro caso**
10. Etiquetar $w \rightarrow y$ con *forzado*
11. **Si** existe un arco $z \rightarrow y$ tal que $z \neq x$ y z no sea padre de x **entonces**
12. Etiquetar $x \rightarrow y$ y todos los arcos *desconocidos* incidentes en y con *forzado*
13. **En otro caso**
14. Etiquetar $x \rightarrow y$ y todos los arcos *desconocidos* incidentes en y con *reversible*

Grafo Acíclico Parcialmente Dirigido - Grafo Acíclico Dirigido (PDAG-DAG)

Este algoritmo tiene como entrada un *PDAG* representante de una clase de equivalencia y como salida un *DAG* miembro de esa clase de equivalencia. Una implementación simple de este algoritmo es debida a Dor y Tarsi (1992) y que se explica a continuación.

Algoritmo PDAG_A_DAG
Entrada: *PDAG P*
Salida: *DAG G*

1. Crear el *DAG G* y añadir sólo y exclusivamente los arcos dirigidos de P
2. **Mientras** existan nodos en P **hacer**
3. Seleccionar un nodo x de P, tal que:
   (a) x no tenga arcos de salida, es decir, no tenga hijos.
   (b) **Si** N_x no es vacío, **entonces** N_x será un cliqué y cada uno de los nodos del conjunto de los vecinos estará conectado con todos los padres de x.
   (c) Si P admite una extensión consistente, el nodo x garantiza su existencia.
4. **Para** cada arco $y - x$ que incida en x de P **hacer**
5. Insertar un arco dirigido $y \rightarrow x$ en G
6. Eliminar el nodo x y todos los arcos que incidan en él de P

1.2.4 Operadores sobre Grafos Acíclicos Parcialmente Dirigidos completados (CPDAG)

Dado un criterio de puntuación para evaluar Redes Bayesianas, un algoritmo de aprendizaje intentaría identificar una o más estructuras que tengan la mejor puntuación al aplicar el algoritmo de búsqueda heurístico. Se formulará un espacio de búsqueda que pueda ser usado por el algoritmo de búsqueda heurístico (en unión con un criterio de puntuación) para poder moverse dentro de las clases de equivalencias de una Red Bayesiana.

Un espacio de búsqueda tendrá tres componentes:

1. Un conjunto de estados.
2. Un esquema de representación para los estados.
3. Un conjunto de operadores.

El conjunto de estados representa un conjunto lógico de soluciones para el problema de búsqueda, el esquema de representación un camino eficiente para representar los estados y el conjunto de operadores es usado por el algoritmo de búsqueda para trasformar las representaciones de un estado a otro.

En una formulación simple de un espacio de búsqueda para el aprendizaje de Redes Bayesianas, los estados de la búsqueda podrían ser definidos como una Red Bayesiana individual, la representación de un estado como un simple grafo dirigido acíclico y los operadores estarían definidos para realizar cambios locales en ese grafo. Por ejemplo, Chickering, Geiger y Heckerman [16] compararon varios procedimientos de búsqueda en el espacio de búsqueda de una Red Bayesiana, usando los siguientes operadores:

Para cualquier par (x, y) si x e y son adyacentes, el arco que los conecta puede ser borrado o invertido.

Si x e y no son adyacentes, se puede añadir un arco entre ellos en cualquier dirección. Todos los operadores son sometidos a la restricción de que no se pueden formar ciclos en la red.

Al espacio de Red Bayesiana se le llamará *B-espacio.*

En la figura 1.8 se puede ver un ejemplo de cada operador en el *B-espacio.*

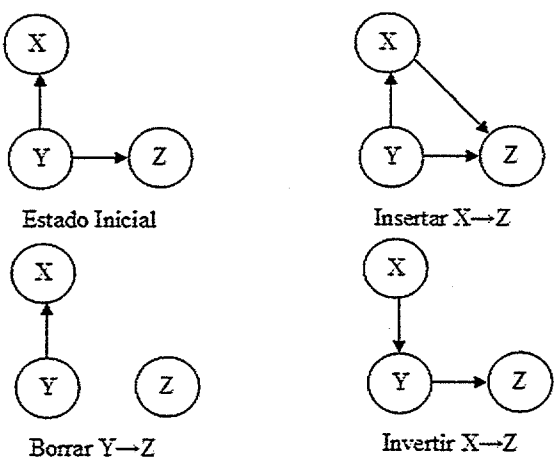

Figura 1.8: Resultados de aplicar un operador en el *B-espacio.* Todas las operaciones están realizadas a partir del estado inicial.

Definición 1 *Un criterio de puntuación s para una Red Bayesiana es* descomponible *sí éste puede ser interpretado como la suma de sus partes o unidades, cada una de las cuales está en función de un solo nodo y sus padres.*

Esto se puede expresar como:

$$S(G) = \sum_{i=1}^{n} s(x_i, \Pi_{x_i}) \tag{1.10}$$

Donde n es el número de nodos de DAG G *y* $S(x_i, \Pi_{x_i})$ *está en función del nodo* x_i *y sus padres en G.*

Dado un criterio de puntuación descomponible, el *B-espacio* es especialmente importante dado que los cambios en las puntuaciones por la aplicación de un operador se pueden calcular localmente.

Se define también un espacio de búsqueda para las clases de equivalencia, al cual se llamará *E-espacio*. Será necesario que los *PDAGs* que se utilicen en el *E-espacio* sean completados, eliminando así el problema de tener múltiples representaciones para la misma clase de equivalencia. Para completar la especificación del *E-espacio* se definirán seis operadores simples que podrán ser aplicados a los *PDAGs* completados. Todos los operadores estarán sujetos a las restricciones de que el *PDAG* resultante sea acíclico y admita una extensión consistente. Los operadores serán los siguientes:

1. **InsertarU**: Para cualquier par de nodos x e y que no sean adyacentes en P^c, se podrá insertar un arco no dirigido entre ellos.
2. **BorrarU**: Para cualquier arco no dirigido $x - y$ de P^c, podemos borrar el arco.
3. **InsertarD**: Para cualquier par de nodos x e y que no sean adyacentes en P^c, se podrá insertar un arco dirigido en cualquier dirección.
4. **BorrarD**: Para cualquier arco dirigido $x \rightarrow y$ en P^c, podemos borrar el arco.
5. **InvertirD**: Para cualquier arco dirigido $x \rightarrow y$ en P^c, podemos invertir el arco.
6. **Construir V Estructura**: Para cualquier tupla de nodos x, y, z en P^c, tal que:
   (a) (x, z) no sean adyacentes,
   (b) P^c tenga el arco no dirigido $x - y$,
   (c) P^c tenga el arco no dirigido $y - z$,

   Podremos insertar la V estructura $x \rightarrow y \leftarrow z$.

Después de aplicar un operador a un *PDAG* completado, el *PDAG* resultante no tiene porque ser necesariamente completado. Para que el *PDAG* sea completado tendremos que utilizar los algoritmos de transformación descritos anteriormente.

Los pasos para obtener el *PDAG* completado son los siguientes:

1. Llamar al algoritmo **PDAG_A_DAG** con entrada P para extraer una extensión consistente G. Si P no admite una extensión consistente, entonces el operador dado no es válido.

2. El algoritmo **DAG_A_CPDAG** es llamado con entrada G y como salida tendrá un *PDAG* completado.

El proceso anterior queda reflejado en la figura 1.9:

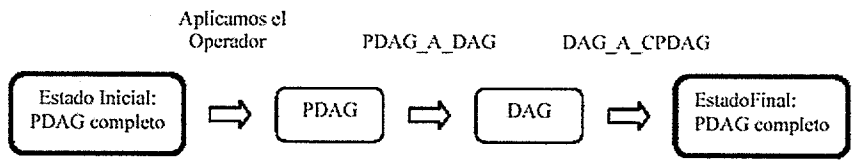

Figura 1.9: Proceso para aplicar un operador.

En la siguiente figura (fig.1.10, pág.22) se ilustra la aplicación de cada tipo de operador:

Otra restricción que se debe tener en cuenta es que sólo se podrá insertar un arco en un *PDAG completado* si ese arco tiene la misma direccionalidad en el *PDAG completado resultante.*

Y con esto se refiere a que no se puede puede insertar un arco no dirigido si ese arco pasa a ser dirigido en el *PDAG* resultante, y viceversa. Por ejemplo, en la figura 1.10 no está permitido insertar el arco dirigido $v \rightarrow u$ porque al pasar el *PDAG* a *DAG* y después a *CPDAG* la dirección del arco dirigido insertado se convierte en un arco no dirigido, rompiendo la regla anterior.

Como establece el siguiente teorema, los operadores son completados para el espacio de búsqueda, es decir, sean P_1^c y P_2^c dos *PDAGs* completados que admiten una extensión consistente, entonces existe una secuencia de operadores legales que pasan de P_1^c a P_2^c.

Teorema 2 *Los operadores InsertarU, InsertarD, BorrarU, BorrarD y Construir V Estructura son completados.*

1.2.5 Puntuación

En este apartado se describirá como se puntúan localmente todos los operadores del *E-espacio* nombrados anteriormente.

En particular, se mostrará que dado un criterio de puntuación descomponible para *DAGs*, el incremento en la puntuación que resulta de aplicar cada uno de los operadores puede ser calculado evaluando a lo sumo cuatro términos en la suma de la ecuación 1.10 (pág.20).

Para la ejecución de las operaciones se necesita que cada una de ellas cumpla una serie de condiciones. Por ejemplo, para que la operación Borrar arco dirigido

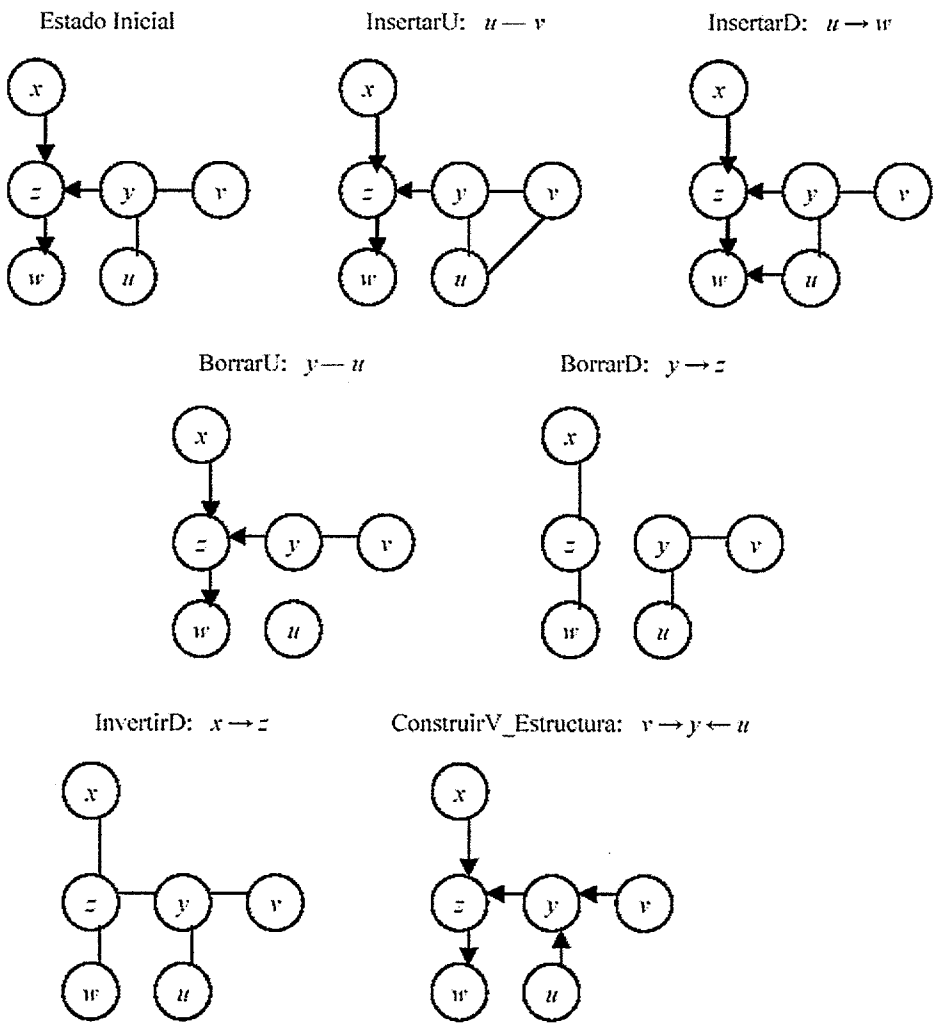

Figura 1.10: Ejemplo de cada tipo de operador.

$x \rightarrow y$ de un *PDAG completado* P^c es necesario que N_y sea un cliqué de arcos no dirigidos, siendo a N_y el conjunto de vecinos del nodo y.

Para una información más profunda acerca de las condiciones necesarias para que se puedan efectuar las operaciones y de los metodos de puntuación se puede consultar a Chickering [3].

1.3 Generalidades sobre Algoritmos Evolutivos

La principal aportación de la computación evolutiva a la metodología de resolución de problemas consiste en el uso de mecanismos de selección de soluciones potenciales y de construcción de nuevos candidatos por recombinación de características de otros ya presentes, de modo parecido a como ocurre en la evolución de los organismos naturales.

Cuando surgen varios candidatos como solución de un problema aparece la necesidad de establecer criterios de calidad y de selección y surge la idea de combinar características de buenas soluciones para obtener otras mejores. No se persigue una simulación de los procesos naturales, sino más bien una emulación de dichos procesos.

Un *Algoritmo Evolutivo* será cualquier procedimiento estocástico de búsqueda basado en el principio de la evolución. Es decir, los más aptos tienen más posibilidades de sobrevivir y, como resultado, más oportunidades de transmitir sus características a las generaciones siguientes.

Al ejecutar un Algoritmo Evolutivo una población de individuos, que representa a un conjunto de candidatos a soluciones de un problema, es sometida a una serie de transformaciones con las que se actualiza la búsqueda y después a un proceso de selección que favorece a los mejores individuos.

Cada ciclo de transformación + selección constituye una *generación.* Se espera de dicho algoritmo que tras un cierto número de generaciones (*iteraciones*) el mejor individuo esté razonablemente próximo al objetivo.

Estructura y funcionamiento de un algoritmo evolutivo.

En la figura 1.11 podemos ver la estructura genérica del bucle básico de un algoritmo evolutivo:

Una población que consta de n miembros se somete a un proceso de selección para constituir una población intermedia, llamada *población auxiliar*, de m criadores. De dicha población intermedia se extrae un grupo reducido de individuos llamados *progenitores*, que serán los encargados de crear la siguiente generación. Utilizando los operadores genéticos, los progenitores son sometidos a ciertas transformaciones de alteración y recombinación en la fase de reproducción, en virtud de las cuales se generan s nuevos individuos que constituyen la descendencia.

Para formar la población de la siguiente generación $[t+1]$ se deben seleccionar n supervivientes de entre los $m + s$ de la población auxiliar y la descendencia. A esa operación de la llama *fase de reemplazo.*

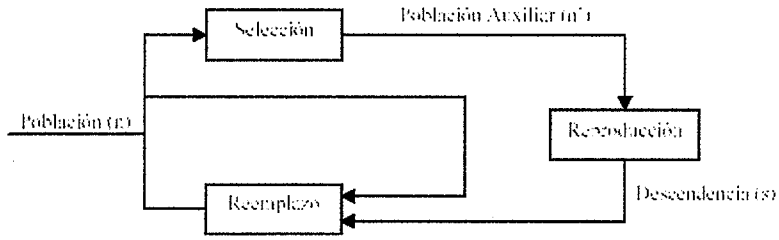

Figura 1.11: Bucle básico del funcionamiento de un Algoritmo Evolutivo.

Existen muchos tipos de Algoritmos Evolutivos. El trabajo se centra en los tipos basados en *cruce* y en la *mutación*. Para nuestro algoritmo se utilizó el tipo *mutación*, dejando para futuros trabajos el de cruce(ver sección 4.2.2).

La *técnica del cruce* se centra en el intercambio de las características de los individuos de la población. Consiste en la obtención de nuevos individuos a partir de la unión de fragmentos de otros individuos. Normalmente se toman dos elementos cualesquiera de la población, se decide la frontera por donde fragmentar y de cada uno de los individuo del cruce se toma una parte, que a continuación se une a la parte tomada del otro individuo para crear el nuevo.

Esta técnica tiene como ventaja el rápido paso de información entre individuos. El hecho de que se cruce a dos buenos individuos para obtener uno nuevo da una alta esperanza de que el nuevo tenga características buenas de ambos padres y por tanto mejore la población con cierta facilidad. El problema de esta técnica es la posible movilidad de los elementos que componen a los individuos de la población.

Imaginemos que el objetivo fuera encontrar un vector de 10 posiciones mediante cruce.

En el caso de que las posiciones fueran fijas, entonces solo nos tendríamos que preocupar del valor de cada una, con lo que no cabe la posibilidad de repetir ninguno de los elementos que forman a los vectores hijos y las fronteras serían muy facíl de delimitar.

Si en cambio el orden de los elementos que forman el vector pudiera ser intercambiable, habría un grave problema. Llegaría un momento en que alguno de los hijos creados podría tener elementos repetidos en su interior, con el problema de inconsistencia que esto acarrearía. A partir de ese momento sería imposible llegar a una solución satisfactoria del problema. La complejidad de encontrar en este caso fronteras válidas para realizar el cruce entre dos elementos es mucho mayor que en el caso de que los elementos que conforman los individuos estén fijos en la estructura de estos.

De ahí que la solución del cruce sea más difícil de controlar e implementar que la de la mutación.

La *técnica de la mutación* se centra en el cambio puntual en elementos que forman a los individuos de la población.
Consiste en el cambio de características de los elementos de los individuos de manera aleatoria. La ventaja principal de esta técnica es la facilidad de aplicación de la misma y el más fácil control de las inconsistencias que pudiera provocar.

Por ejemplo, en nuestro programa el único control que se ejecuta es el de que la operación sea factible (que exista un arco para borrar o no exista un arco para poder insertar uno en esa posición) y el que no cree ciclos. Una vez controlados esos dos aspectos, la operación tiene el visto bueno para ser ejecutada, y se tiene certeza de que el individuo creado a partir de ella tendrá una estructura correcta.

El problema de la mutación viene dado por la poca efectividad que presenta. Son la aleatoriedad y el hecho de trabajar cada vez en un par de elementos por individuo las razones de la lentitud de su progreso. Aunque también se hacen intentos para ganar optimización en este sentido. Por ejemplo, la operación crear V estructura trabaja directamente sobre tres nodos, y varía su estructura de manera considerable cada vez que se puede ejecutar. El problema que tiene es que las condiciones que necesita para que se den son dificiles de cumplir, así que el número de operaciones ejecutadas es menor que el de las otras operaciones. También se consideran los operadores *SHAREAD* y *SHAREAND*, con los que se pasa información entre miembros de la población.

Con esto lo que se pretende es suprimir intentos de mutación correctos pero que no aportan mejoras a los individuos, pasando directamente a compartir arcos probados y buenos entre las estructuras.

Con respecto a otros paradigmas, como las técnicas voraces, que serán las que se verán, los Algoritmos Evolutivos presentan ventajas y un desventajas. Una ventaja es que los Algoritmos Evolutivos no precisan conocer el orden de precedencia de los nodos dentro de la estructura. Sin embargo los algoritmos voraces no pueden desarrollar la busqueda si desconocen ese orden de las variables. La desventaja que presentan es el tiempo que se necesita para llegar a una solución. Mientras que el K2 solo trabaja con una red, el algoritmo Evolutivo trabaja con una población mucho mayor, y el tiempo que necesita para poder desarrollar su ejecución es proporcional.

Los algoritmos voraces, para llegar a la solución correcta, tienen que tomar todos los posibles caminos que vayan desde el individuo inicial hasta la solución para asegurarse de que han llegado al objetivo.

Por su parte, y mediante las funciones de puntuación, los Algoritmos Evolutivos avanzan a través de las posibles soluciones seleccionando aquellas que les parecen mejores y desestimando las demás, sin tener siquiera que preocuparse de su existencia.

Los Algoritmos Evolutivos tienen dos problemas principales:

- *La utilización de recursos*
  Para poder llevar a cabo una búsqueda con estos algoritmos es necesaria una cantidad de recursos (sobre todo de memoria) mucho mayor que en

otros tipos de algoritmos, ya que en los evolutivos es necesario mantener de manera permanente una población de n individuos más sus m descendientes, los cuales además pueden aumentar de tamaño en el tiempo.

- *La posibilidad de no llegar a una solución correcta*
  Al igual que en otros tipos de algoritmos, los Algoritmos Evolutivos no siempre llegan a la solución correcta. Es normal quedarse en algún máximo local, próximo a la solución deseada, pero no igual al objetivo inicial.

1.4 Algoritmos Evolutivos para la Inducción de Redes Bayesianas

Para poder aplicar Algoritmos Evolutivos al problema de la inferencia de las Redes Bayesianas es necesario disponer de:

1. *Una población de posibles soluciones representadas a través de individuos.* Para nuestro proyecto se considerará que un individuo contiene los datos necesarios para construir una Red Bayesiana (topología del DAG).
2. *Un procedimiento de selección basado en la aptitud de los individuos.* Es decir, se realizará una evaluación de cada individuo, en la cual se obtendrá un valor según se esté utilizando el tipo *BDeu* (verosimilitud marginal) o el tipo BIC (verosimilitud), y se seleccionará la mejor solución, el valor menor si se minimiza o el mayor si se maximiza.
3. *Un procedimiento de transformación.* Esto es, de construcción de nuevas soluciones a partir de las disponibles actualmente.

Una primera aproximación al problema de la búsqueda de Redes Bayesianas a partir de Algoritmos Genéticos o Evolutivos es dado en el trabajo de Larrañaga y sus colaboradores [13].

En él, partiendo de la búsqueda de la estructura de las redes y no de los parámetros de estas, se hace un acercamiento a la búsqueda de soluciones a partir de *DAGs*. Al igual que en nuestro caso, la búsqueda de la solución se realiza a través de una función objetivo. En su caso la función utilizada es la misma que desarrollaron Cooper y Herskovits [5].

En su trabajo parten de dos posibles aproximaciones a la búsqueda de las redes, dependiendo de si se conoce o no el orden topológico de los nodos que la conforman. En el caso de conocer el orden de los nodos, se asume la corrección de los *DAGs* creados, puesto que no existe la posibilidad de que se creen ciclos. En el caso contrario, al no tener la certeza de la corrección de los *DAGs* creados, se pasa a introducir el concepto de *operador de reparación*, que aseguraría que los individuos creados a partir del algoritmo son correctos.

Wong, Lam y Leung [17] desarrollaron un Algoritmo Evolutivo que ademas de basarse en los operadores geneticos habituales utilizaba un nuevo operador, que a través de una técnica de *Data Mining* obtenía información de la estructura que iba a modificar.

Muruzábal y Cotta [4] [7] introducen la posibilidad de estimar las Redes Bayesianas desde una visión no puramente genética. Entra aquí en juego el concepto de *fenotipo*. En el caso de las Redes Bayesianas, la información fenotípica sería aquella que nos informa acerca de si un determi- nado elemento de un red está en una posición idonea para él, o mejor dicho, como de idonea es la posición que ocupa en la actualidad.

En el trabajo de Chickering [3] se aborda el concepto de clases de equivalencia, anteriormente explicado, aplicado a la búsqueda de la estructura de las redes. El resultado es que cada uno de los posibles operadores debe producir resultados consecuentes con los principios de las clases de equivalencia y correctos con respecto a la red buscada.

La finalidad de este proyecto es la implementación de un nuevo Algoritmo Evolutivo que tenga las características antes mencionadas e introducir dos nuevos operadores, *CompartirArcoDirigido* y *CompartirArcoNoDirigido*. La introducción de estos nuevos operadores busca el intercambio de información válida a priori entre los elementos de la población. La base es que si hay un arco que es bueno para un individuo en cuestión, posiblemente también será válido para otros individuos de la población, que tal vez no lo tengan en su estructura.

Capítulo 2

Objetivos

2.1 Descripción del Problema

La inducción y visualización de Redes Bayesianas a partir de datos es una labor que requiere de una carga de trabajo bastante considerable.

Las técnicas utilizadas, como se explicará más adelante con el algoritmo K2 [5], normalmente son de carácter voraz.

Esto es, partiendo de una estructura bayesiana inicial aleatoria se busca mediante combinatoria la estructura que mejor represente los datos existentes. En el caso de los algoritmos voraces la búsqueda se realiza en un modo secuencial. Se recorren todas las posibles estructuras de la red, con lo que el resultado óptimo está asegurado. El problema de este enfoque la encontramos en la necesidad de conocer con anterioridad el orden topológico de los nodos que componen la red. Sin esta información de partida, no es posible utilizar este algoritmo para solucionar el problema.

Los algoritmos voraces toman una única estructura de partida y hacen todas las combinaciones posibles de arcos para cada uno de los nodos de la estructura hasta encontrar la que mejor se adapte a cada uno. Esto no quiere decir que se consiga llegar en cada ejecución al objetivo buscado. Tambien se pueden quedar en máximos locales, como le sucede a los Algoritmos Evolutivos.

El enfoque evolutivo cambia totalmente esta perspectiva. En este caso en lugar de partir de una única red inicial para la solución del problema, lo que haremos será partir de una población de n elementos aleatoria. A partir de aquí se comienza a *mutar* a todos los individuos de la población en cada *generación* (cada iteración del algoritmo principal). Lo que se busca con esto es una diversificación de la búsqueda, una amplitud de soluciones posibles.

A cada uno de los elementos de la población, con la intención de buscar un elemento que mejore al anterior, se le modifican los arcos que lo forman. Tras la modificación, se calcula la puntuación del individuo creado con el cambio efectuado. Después de hacer esto con cada uno de los n nuevos individuos crea-

dos a partir de los individuos iniciales, formamos un único grupo de individuos (padres e hijos), del cual nos quedaremos con sus mejores representantes. Esto es, con las n mejores redes que haya en el grupo.

Otra de las ventajas que tiene este enfoque es que en lugar de trabajar directamente con Redes Bayesianas lo hacemos con clases de equivalencia (explicadas en anteriores apartados). Con esto conseguimos evitar la redundancia de las redes halladas. Ahora en lugar de buscar red por red la que más se asemeje gráficamente a la red objetivo, lo que haremos será trabajar con grupos de redes que por semejanza pueden ser agrupadas en una única clase que las representa, evitando así la búsqueda individualizada.

Lo que se pretende es ahorrar todas las combinaciones no necesarias que tiene el enfoque voraz. La problema está en la aleatoriedad que se tiene a la hora de la mutación de los arcos, la posible repetición de estructuras y la formación de ciclos.

Ante estas problemas apareció un trabajo que sentaba las bases teóricas para afrontar el problema. Chickering [3] desarrolló un conjunto de 6 operadores de mutación(ver pág. 20).

Con el objetivo de evitar también en parte la perdida de efectividad a la hora de aplicar los operadores y la posible repetición de las operaciones, cada una tiene un *algoritmo de búsqueda de posibles nodos.*

Por ejemplo, en el caso de la inserción de un arco dirigido en la estructura de turno, se selecciona aleatoriamente un par de nodos para la inserción, pero si entre esos nodos existiese ya un arco de algún tipo, entonces en lugar de desperdiciar la iteración con un par de nodos sobre el que no se va a poder desarrollar ninguna mejora, lo que hacemos es buscar a partir de la posición de estos un par de nodos pertenecientes a la estructura en los que se pueda insertar dicho arco dirigido, sin detenernos a probar que el cambio mejore o no la estructura.

Como ampliación del grupo de operadores de Chickering, y siempre teniendo en cuenta el cumplimiento de las condiciones para que se mantenga la estructura de las clases de equivalencia, se ha llevado a cabo la implementación de otros dos nuevos operadores, *SHAREAD* y *SHAREAND.*

En el caso del primero, *SHAREAD* busca en un individuo donador en la población, que en nuestro caso será el mejor individuo, un arco dirigido entre dos nodos cualesquiera de este y lo comparte con otro individuo de la población. Lo que se busca es que partiendo de que el donador es un buen individuo, esto es, tiene una buena puntuación, suponemos que su estructura será buena, y por lo tanto el arco que donará será bueno y mejorará la estructura del individuo que recibe la donación. Una vez que se ha encontrado un arco dirigido en el individuo donador que pueda ser insertado en el individuo receptor (comprobamos que no existe ningún tipo de arco entre ambos nodos en este) procedemos a la inserción del mismo en la estructura(siempre y cuando cumpla la condición de no formar ciclos), a la inserción de su puntuación en el vector de puntuaciones y a la estructura resultante en la población.

En el caso de la segunda operación, *SHAREAND*, es exactamente igual, solo que en lugar de compartir un arco dirigido se comparte uno no dirigido.

Lo que conseguimos con estos dos operadores es una nueva forma de introducir información en la estructura, no tan ligada a la aleatoriedad, aunque esta siga presente, en cuanto a los nodos que se eligen de partida en el individuo donador para comenzar a buscar el arco que se quiere compartir, pero ahora también tenemos en cuenta la transmisión de información entre los elementos de la población.

Esta nueva llegada de información puede ser importante a la hora de transmitir estructuras buenas entre individuos, y es lo que trataremos de comprobar con las diferentes pruebas a las que someteremos a las operaciones.

Más adelante, en la definición de las operaciones, se verá más claramente como funcionan estos operadores.

2.2 Estudio de Alternativas

Dentro de las técnicas algorítmicas válidas para afrontar este problema nos centraremos en aquella con la que hemos confrontado los resultados del proyecto, el algoritmo voraz K2.
Si conocemos un orden topológico de los nodos, podemos intentar encontrar una estructura que se ajuste al mejor, en un sentido local, conjunto de padres para cada nodo independientemente del resto. Esto es lo que hace realmente el algoritmo K2, desarrollado por Cooper y Herskovits [5]. Para poder emplear este algoritmo se debe especificar una función de puntuación. Existen dos funciones típicas:

- *La métrica BDeu*, la cual integra sus parámetros, es decir, es la verosimilitud marginal del modelo. Utiliza distribuciones a priori Dirichlet.
- *La métrica BIC* (Sección 1.1.1).

K2 [5] es un algoritmo de búsqueda *greedy* o voraz que trabaja de la siguiente forma: inicialmente cada nodo no posee padres y se le van añadiendo aquellos cuya unión supongan una mejora en la puntuación, y, por tanto, una mejor estructura. Cuando no quede ningún padre cuya unión mejore la puntuación, es decir, ésta se incremente, entonces se para y se continua con el siguiente nodo hasta que se completen todos los nodos de la red.

Al tener un orden topológico, no se necesita verificar si hay ciclo o no, ya que sabemos que éstos no se van a producir, y se pueden elegir los padres para cada nodo de forma independiente. Además se debe especificar el número máximo de padres posibles que queremos que tenga cada nodo.

Como entradas tiene:

- Un número de nodos.
- El orden topológico que se pretende que tengan esos nodos. Este será permanente.
- El número máximo de padres que pueden tener los nodos.

- Una lista de patrones.
- El tipo de función de puntuación que vamos a utilizar: *BDeu o BIC.*

Como salida obtenemos la mejor red donde a cada nodo se le han asignado el conjunto de mejores padres o más probables que pueden tener, dada la lista de patrones.

2.3 Metodología

La metodología utilizada para el desarrollo del algoritmo es *estructurada.* El hecho de que el proyecto trabaje directamente con una librería desarrollada bajo este paradigma de programación condicionó la elección.

El proyecto se comenzó pensando en los requisitos que debería de cumplir el lenguaje de programación elegido a partir del programa a desarrollar. El lenguaje elegido fue el *ANSI C.* La razón principal para elegir este lenguaje es su portabilidad, compatibilidad y nivel de estandarización, lo que le hace manejable y transportable entre plataformas. La idea era haber hecho las pruebas también sobre UNIX, aunque al final no se han llevado a cabo. En teoría, al desarrollar el programa en *ANSI C* permite la migración, pero como ya he dicho antes no se ha podido probar.

Una vez implementado, era necesario especificar las pruebas para todas aquellas partes del programa que pudieran ser causa de problemas (sobre todo en lo referente a que las estructuras que se utilizaban en todo momento siguiesen las reglas marcadas por las clases de equivalencia), así como después las pruebas para medir la eficacia y el éxito de los nuevos operadores.

Por último, el entorno de desarrollo elegido es el *dev-C++*, una herramienta de libre distribución similar a los entornos Visual o Borland. Se ha intentado seguir esta línea de estandarización y utilización de software libre para facilitar el uso del programa y las opciones que tiene (GraphViz para visualizar los resultados, LaTeX [12] para escribir la memoria).

Capítulo 3

Descripción informática

3.1 Introducción

A continuación se van a introducir a grandes rasgos los elementos que componen el algoritmo, las librerías utilizadas y desarrolladas, su ordenación y las funciones que las componen.

En primer lugar, y antes de entrar en los módulos desarrollados para la utilización del algoritmo y de las estructuras en él presentes, fue necesario el aprendizaje de la librería desarrollada por Alicia Puerta Torralbo [6]. Esta fue desarrollada para el tratamiento de Redes Bayesianas y Clases de Equivalencia. Los archivos que la componen son los siguientes:

- `RedBay.c`: Es el módulo principal y el que se encarga de dar soporte a todas las operaciones con Redes Bayesianas (creación, destrucción, transformaciones de redes a grafo y viceversa, ...).
- `Patrones.c`: Da soporte a todas las operaciones con patrones y de listas de patrones, que serán las muestras en las que se basarán para dar forma a los grafos.
- `Inferencia.c`: Permite los aprendizajes de las estructuras, el cálculo de las verosimilitudes y el muestreo de una red dado un número de patrones, lo cual será muy útil para la creación de patrones a partir de redes aleatorias.
- `K2.c`: Alberga una librería para la utilización de este método de búsqueda como contrapunto y comparación de nuestro Algoritmo Evolutivo.
- `Grafo.c`, `PDAG.c`, `Puntuación.c` y `MovimientosPDAG.c`: Estos módulos son los encargados de trabajar con las clases de equivalencia en formato *PDAG*, desde su creación, transformación, comparación y puntuación hasta su destrucción.

- `Nodos.c`: Controla y maneja todas las funciones relacionadas con el uso de nodos a lo largo de la ejecución del algoritmo.
- `MatrizPC.c`: Controla y maneja todas las funciones relacionadas con el uso de las matrices de probabilidad condicionada de cada uno de los nodos a lo largo de la ejecución del algoritmo.
- `FuncMath.c`: En este módulo se abarcan tanto funciones puramente matemáticas como funciones de apoyo para otros módulos de la librería.

Por otra parte está toda la implementación codificada en el proyecto para el desarrollo del Algoritmo Evolutivo y el manejo de las estructuras necesarias para el almacenaje de los datos. Los módulos desarrollados son los siguientes:

- `Principal.c`, donde se encuentra el Algoritmo Evolutivo.
- `Población.c`, donde tenemos las funciones que manejan la población con la que trabaja el Algoritmo Evolutivo y el vector de puntuaciones asociado a dicha población. La población es un conjunto de $TAMANOPOB$ elementos, siendo $TAMANOPOB$ una constante presente en el archivo de cabecera `Población.h`. Este módulo nos permite:
  - Crear una población y un vector inicial.
  - Insertar elementos en la población.
  - Ordenar la población con respecto a la puntuación de sus elementos.
  - Seleccionar el mejor y el peor elemento de la población.
  - Mostrar por pantalla el vector de puntuaciones.
  - Guardar las mejores estructuras resultantes de la población.
- **SelecciónNodos.c**, donde se encuentran todas las funciones que se dedican a la selección aleatoria y no aleatoria de los nodos de los elementos de la población para su mutación. Este módulo nos permite:
  - *Hacer una selección de nodos aleatoria*
    Esta sería independiente del tipo de operación a realizar. Los nodos se tomarían al azar para cada operación.
  - *Hacer una selección de nodos no aleatoria*
    Cada una de las operaciones requiere que los nodos afectados cumplan una serie de condiciones.
    Insertar Arco Dirigido e Insertar Arco no Dirigido. Se buscan dos nodos dentro de la estructura entre los que no exista ningún arco previamente.
    Borrar Arco Dirigido. Se buscan dos nodos en la estructura entre los que exista una arco dirigido.
    Borrar Arco no Dirigido. Se buscan dos nodos entre los que exista un arco no dirigido.

Invertir Arco Dirigido. Se buscan dos nodos entre los que exista un arco dirigido en el sentido buscado.

Construir V Estructura. Se buscan tres nodos que cumplan las condiciones que se pueden ver en el punto 1.2.4 (Pág. 22).

Compartir Arco Dirigido. Se buscan dos nodos en una estructura donante entre los que exista un arco dirigido. Este arco después se intentará pasar a otra estructura receptora.

Compartir Arco no Dirigido. Se buscan dos nodos en una estructura donante entre los que exista un arco no dirigido, para después intentar pasarselo a otra estructura receptora.

- **FuncAux.c**, donde se encuentran toda una serie de funciones auxiliares necesarias para el funcionamiento del algoritmo. Este módulo nos permite:
  - Crear un encabezamiento para la ejecución del algoritmo.
  - Mostrar información acerca de las operaciones que se vayan realizando sobre los elementos de la población.
  - Elegir un progenitor de entre unos cuantos candidatos fijándonos en la puntuación de este.
  - La recogida y tratamiento de todos los parámetros necesarios para el desarrollo del algoritmo.
  - La posibilidad de seleccionar el valor de las densidades de los arcos de una manera aleatoria o elegida.
  - Comprobar que un *PDAG* es consistente.
  - Pasar una Red Bayesiana a un formato en que pueda ser representada en modo gráfico.
  - La escritura de la red actual con la que esté trabajando el algoritmo en un momento determinado a un fichero de texto.
  - La escritura de un grafo a un fichero.
  - La carga de patrones desde un fichero exterior.
  - Funciones de control de la ejecución (detención temporal de la ejecución, salida del programa, ...).
  - Aleatorización de datos entre topes de valores (0 y el valor utilizado).
  - Carga de la red original.
  - Creación de patrones a partir de una red dada.
  - Creación de índices para la selección de elementos de la población.
  - Comparación de números enteros.
  - Creación de redes objetivos aleatorias.
  - Muestra de estadísticas de la ejecución.
  - Comparación de la red actual con la red original.

En los siguientes puntos se hará una revisión más en detalle de lo implementado en el proyecto. Se realizará la especificación, diseño e implementación de las funciones más importantes del mismo, así como una enumeración del resto. También se intenta hacer un enfoque global del proceso de programación, el porqué de las decisiones más importantes que se hayan tomado y por último mostrar la implementación del código.

3.2 Especificación

La especificación de los requisitos del sistema es la culminación de la parte de análisis del proyecto. En esta parte se realiza un análisis detallado de cada función, su comportamiento, así como información acerca de todos los datos de interés (pruebas, requisitos de diseño, ...) que pueda tener. Nos centraremos en las funciones más importantes, tanto en esta parte como en la parte de diseño e implementación.

Las funciones seleccionadas serán las más representativas del proyecto y las que por ser similares a otras especifiquen y den sentido a la parte más importante del código.

Antes de comenzar a describir las funciones, se definiran los tipos de datos básicos a utilizar durante las fases de análisis, diseño e implementación. La especificación de las funciones dependerá directamente de estos tipos de datos. Los más relevantes son los *DAG* (grafo dirigido acíclico), los *PDAG* (grafo parcialmente dirigido acíclico) y el *BN* (Red Bayesiana). Ellos son los que representan a las estructuras que se utilizarán más a menudo.

Se intentará hacer una descripción de lo que en cada momento de la vida del programa se ha requerido y la solución que se le ha ido dando a cada una de las partes. En realidad no se van a especificar todas las funciones presentes en los paquetes programados, sino el programa principal y las funciones principales del mismo o las que presentan algún aspecto más o menos importante para la ejecución.

Se comenzó con la especificación de requisitos del enfoque evolutivo. Era necesaria una población de n individuos, a partir de los cuales comenzar a crear una descendencia capaz de generar un individuo que se asemeje en la medida de lo posible a la red objetivo.

Para ello partimos de la necesidad de crear, mantener, mejorar y destruir esa población siempre que sea necesario. El parámetro clave para el manejo de la misma será la puntuación de los elementos. Esta puntuación es la medida de valor que tomamos para clasificar los elementos de la población en mejores o peores. Sabiendo esto se comienzan a especificar las principales funciones del módulo `población`, como son las de crear la población inicial o asignarle puntuaciones a los individuos que la forman.

creacionPoblacion La función `creacionPoblacion` es la encargada de crear los primeros individuos para que el algoritmo pueda empezar a trabajar. La

condición indispensable de esta función era que devolviera la estructura creada al programa principal para que desde este se empezará a trabajar. La creación de los individuos depende de la selección hecha por el usuario. Los posibles tipos de selección son aleatorios, decididos por el usuario o decididos por la máquina.

En el primer caso, la selección de densidades de los arcos es aleatoria para cada uno de los elementos presentes en la población. Lo que se consigue con esto es que de inicio, si no se tiene la densidad de los arcos adecuada para la red, se tiene la posibilidad de empezar con varias redes que seguramente se aproximarán bastante a la densidad de la buscada. La función que lo implementa es `seleccionDeDensidadesAlea`.

Si por el contrario se cuenta con una información previa de cual puede ser la densidad de los arcos, entonces se puede introducir la densidad que se quiere aplicar a los individuos de la población. Esto condicionará que todos los individuos partirán con un número más o menos similar de arcos desde el inicio, y que si este número de arcos es apropiado, entonces las redes en un primer momento tendrán una mayor similitud con el objetivo.

Por último, si dejamos que sea el programa el que decida cual es la densidad a poner, entonces este aleatoriamente creará unos cuantos individuos con cada una de las densidades posibles (función `seleccionDeDensidadesNoAlea`), los puntuará y hará la media de las puntuaciones recibidas con cada una de las densidades para ver cual es la que más se asemeja al objetivo. La función que crea la población en estos dos últimos casos es `crearPoblacionInicial`.

Una vez decidida la estructura que tendrá la población y los individuos que la conformen se empieza a pensar en el modo de selección de los individuos para ser mutados, la elección de la operación y de los nodos que participan en ella. La selección del individuo que se va a mutar se puede hacer de dos maneras.

La primera es *proporcional al fitness*, donde se seleccionan algunos individuos aleatoriamente de la población para luego, confrontando las puntuaciones, seleccionar al mejor de ellos como padre.

La otra manera, que es la que utilizamos en nuestro algoritmo, es tomar por orden todos los elementos de la población e ir mutándolos secuencialmente. Los pros y contras de esta técnica respecto de la anterior es que es posible que tenga una menor efectividad, puesto que se mutan individuos de peor puntuación en detrimento de los de mejor puntuación, pero lo que se consigue también es una mayor variedad de resultados, puesto que en la técnica anterior es mucho más fácil que el algoritmo se centre en unas pocas estructuras válidas y se pierda diversidad.

seleccionOperador Una vez que entramos en el bucle, se elige aleatoriamente una de las ocho posibles operaciones que podemos realizar. Una vez obtenida la operación se controla que tipo de ejecución seleccionó el usuario. Si la operación se puede realizar con ese tipo de ejecución, entonces el algoritmo sigue. Si por el contrario la operación no se puede realizar con ese tipo de ejecución, entonces se repite el proceso anterior hasta encontrar una operación válida para ese tipo de ejecución(p.ej., no se podría seleccionar la operación SHAREAD si hemos seleccionado la opción de ejecución básica, porque no es una de las operaciones que soporta).

seleccionDeNodosNoAleatoriaIAD En la función `seleccionDeNodosNoAleatoriaIAD`, por ejemplo, recorremos la estructura del individuo que mutaremos a partir de dos nodos seleccionados aleatoriamente buscando un par de nodos que cumplan las condiciones mínimas necesarias para que entre ellos se pueda insertar un arco dirigido. La condición en este caso es que entre los dos nodos seleccionados no exista ningún arco, ni dirigido ni no dirigido.

El resto de funciones de selección de nodos funciona de manera similar, cambiando las condiciones que se deben dar entre los nodos seleccionados para poder realizar las operaciones (en el caso de borrar arco no dirigido, por ejemplo, entre los dos nodos seleccionados debe existir un arco no dirigido para poder realizar la operación). Si no fuera posible realizar la operación seleccionada, la función en lugar de devolver un conjunto de nodos válidos, como sería normal, provoca un nueva selección de la operación a realizar y una nueva búsqueda de nodos, para que pueda mutar al individuo en cuestión y crear al nuevo.

Una vez decidido el individuo que vamos a mutar, la operación y los nodos sobre los que ejecutarla, lo siguiente será mutar el individuo y darle la puntuación correspondiente. Esto se realiza con unas funciones presentes en la librería sobre la que parte este trabajo, expuestas en el proyecto de fin de carrera de Alicia Puerta Torralbo [6]. En el caso de insertar un arco dirigido la función sería `InsertarPuntADPG`, pero al igual que en la selección de los nodos cada operación tiene su función de puntuación propia.

insertarElementoEnPoblacion El elemento creado debe ser introducido en la estructura de alguna manera, y aquí aparece la función `insertarElementoEnPoblacion`. Esta recibe la puntuación y el individuo creado y lo introduce en la población en la posición correspondiente. La poblacion esta ordenada de mejor a peor individuo, basandose en su puntuación, por tanto el individuo creado se introducirá en la posición que le corresponda en la población siguiendo ese criterio.

ordenarPoblacion Llegamos al momento en que tenemos que ver que individuos hemos creado y con que individuos nos vamos a quedar. Para esto nos hace falta una función que ordene por puntuación a los individuos, tanto los creados como los progenitores. Una vez ordenados es muy fácil la elección de los mejores individuos y la desestimación de los demás.

La función encargada de esto es `ordenarPoblacion`. Mediante un *algoritmo de ordenación de burbuja* pasamos los mejores valores, tanto del vector de puntuaciones como de la población, a las primeras posiciones. Una vez colocados en las primeras posiciones el algoritmo puede hacer uso de los mejores individuos de la población para crear la siguiente generación y desestimar a los demás individuos. También el vector de puntuaciones está preparado para ser utilizado, puesto que la selección y desestimación de valores a partir de este momento es sencilla.

Una vez conseguidos los objetivos de ejecución del programa, se realiza el control estadístico del mismo. Partiendo de los valores que almacenaba cada uno de los individuos se comenzaron a idear funciones que recogiesen esos valores en estructuras aparte y que pudiesen exportar esos valores a ficheros externos para que desde estos a su vez se pudiera hacer una interpretación precisa con programas estadísticos especializados. También a lo largo de la ejecución hay una constante actualización de variables, que luego serán mostradas por pantalla y almacenadas en los ficheros para su visualización en forma de gráficos.

mostrarEstadisticas La función que se encarga de mostrar las estadísticas por pantalla es `mostrarEstadisticas`. Esta función toma los valores almacenados en un vector que recoge los valores estadísticos que se dan durante la ejecución y lo muestra en pantalla.

almacenadoValoresVectorEstadisticas Las otras funciones importantes de esta parte son las que crean las estructuras específicas para cada tipo de datos y las que crean los ficheros para la visualización de los datos en forma de gráfico. Una de las funciones que crean y rellenan las estructuras específicas es `almacenadoValoresVectorEstadisticas`, que toma los valores estadísticos y los coloca en una estructura aparte para su posterior visualización. Existen funciones que realizan la misma operación con las puntuaciones que se han ido dando y con las comparaciones hechas contra el grafo objetivo.

infoGrafoEstadistica Por otra parte tenemos las funciones que crean y rellenan los ficheros de gráficos. Una de ellas es `infoGrafoEstadistica`, que toma los valores del vector de estadísticas y los pone en un fichero para luego obtener

las correspondientes gráficas.

El mismo proceso se llevó a cabo para exportar las redes que resultaron vencedoras en el proceso o cualquier red que, por cualquier razón, tuviera que ser exportada y visualizada en algún formato. Para ello era necesario representarlas en formatos visibles desde el exterior (dot, bif o txt).

solucionFinalAFich La función que permite el paso del formato `.bif` al formato `.dot`, que es el que podrá interpretar el visualizador de gráficos GraphViz [10] [11] es `solucionFinalAFich`, dentro de la cual se llama a la función `pasoBNAFormatoGraphViz`, que es la que realmente pasa la información al formato `.dot`.

comparacionMejorRedRedObjetivo Una funcion que se usa en algunas ocasiones y que es también destacable dentro de la ejecución del algoritmo es `comparacionMejorRedRedObjetivo`. En ella se hace una comparación de arcos entre las dos redes que devuelve el número de arcos de ambas coincidentes, tanto dirigidos como no dirigidos, y el número de arcos invertidos (que unen dos nodos pero en sentido contrario al presente en la red original) existentes en la estructura de la mejor red obtenida.

3.3 Diseño

En la parte de diseño se traduce la información que hemos recogido en la parte de especificación y análisis. Mientras que en la parte anterior tratamos de encontrar las bases, de llegar a entender que es lo que tenemos que hacer, en esta parte se pasa a pensar en como hacerlo.

Esta sección, como dijimos anteriormente, se centrará en las funciones más importantes del código, que ya hemos introducido en la sección de especificación y análisis.

creacionPoblacion En ella se decide el modo de creación de la población inicial a partir de una selección realizada anteriormente. La condición que se utilizará será el modo de creación de población que haya seleccionado el usuario. Esta selección se realiza en una función diferente, que recoge la decisión del usuario. Con esta opción se decide la manera de inicializar la población. Las posibles maneras de hacerlo son:

- `Seleccion de densidades aleatoria`
  Al ser una función que modificará la población, introducimos a esta por referencia para poder hacer las modificaciones necesarias. Una vez dentro de la función se dan los pasos típicos para la creación de una estructura de este tipo: creación de estructuras auxiliares, creación de redes iniciales

aleatorias, introducción de valores en los nodos de cada una de estas redes y una vez finalizado este proceso inserción del elemento creado en la población, previo paso del formato de red al formato de grafo (gracias a las estructuras auxiliares).

La razón de que se individualice esta función radica en la utilización de diferentes tipos de densidades para cada uno de los individuos que forman la población, en lugar de aplicar la misma densidad de arcos para todos. Cuando se crea cada individuo se selecciona aleatoriamente la densidad de los arcos con la que se le crea. Excepto por esa diferencia, la población es exactamente igual que en los dos próximos casos.

- `Selección automática del programa`
  Cuando el usuario selecciona esta opción, antes de crear la población es necesario decidir la densidad de los arcos mediante una función llamada `seleccionDeDensidadesNoAlea`, que crea unos cuantos elementos con cada una de las densidades posibles, los puntúa, halla la media que se obtiene de cada una de las diferentes densidades y a continuación decide cual es la densidad más aconsejable para la población. La función devuelve un valor entero que corresponde al índice de la densidad correspondiente. Ese valor lo conseguimos con la función `seleccionDeDensidades`. Será esa la densidad que se le pase a la función `crearPoblacionInicial` para que cree la población.

- *Selección elegida por el usuario*
  Con esta opción lo que se hace es pasar el índice correspondiente de la densidad de los arcos al valor real que le corresponde con la función `seleccionDeDensidades` y pasarle este valor a la función `crearPoblacionInicial` para que cree la población.

Las dos últimas maneras de creación tienen en común la utilización de la función `crearPoblacionInicial`.

Los pasos que se siguen en esta función son: creación de estructuras auxiliares, creación de redes iniciales aleatorias, introducción de valores en los nodos de cada una de estas redes y una vez finalizado este proceso inserción del elemento creado en la población, previo paso del formato de red al formato de grafo (gracias a las estructuras auxiliares).

seleccionOperador En ella se pide como valor de entrada el tipo de ejecución que el usuario ha elegido. A partir de este valor, y de forma totalmente aleatoria se selecciona una de las 8 posibles operaciones, para luego comprobar a partir de este valor si la operación seleccionada puede ser realizada teniendo en cuenta la selección del usuario. Si es así, devuelve un valor entero que será el utilizado en la estructura condicional del programa principal para seleccionar la operación en cuestión. En cambio si la operación no es correcta con respecto a lo seleccionado por el usuario entonces se vuelve a repetir el proceso aleatorio hasta que se de con una correcta. Las posibilidades de ejecución que tiene el usuario son los cuatro

tipos antes mencionados (*Básica, Básica + SHAREAD, Básica + SHAREAND y Básica + SHAREAD + SHAREAND*).

El tipo de ejecución *Básica* comprende la utilización de los 6 operadores de Chickering [3]: *Insertar Arco Dirigido, Insertar Arco no Dirigido, Borrar Arco Dirigido, Borrar Arco no Dirigido, Invertir Arco y Construir V Estructura.*

El tipo *Básica + SHAREAD* comprende la utilización de los 6 operadores anteriores y la de *Compartir Arco Dirigido*, mientras que el tipo *Básica + SHAREAND* es igual que la anterior pero sustituyendo la operación Compartir Arco Dirigido por *Compartir Arco no Dirigido.*

Por último está el tipo *Básica + SHAREAD + SHAREAND*, que utiliza los 8 operadores posibles.

seleccionDeNodosNoAleatoriaIAD Una vez que tenemos seleccionada la operación el siguiente paso es realizarla. Para ello es necesaria la selección de los nodos que se verán involucrados en el proceso. Cada uno de las 8 posibles operaciones tiene unas exigencias al respecto, por ello para cada una de ellas es necesaria una selección individualizada de los nodos. La función `seleccionDeNodosNoAleatoriaIAD` toma como parámetros de entrada el grafo que se va a mutar y el número de nodos de este. Entonces se seleccionan de manera aleatoria dos nodos desde los cuales comienza la búsqueda de la pareja de nodos que cumpla las condiciones necesarias de la operación. El hecho de que esta selección sea aleatoria intenta que todas las posiciones del *PDAG* tengan las mismas posibilidades de ser seleccionadas. Si cada vez que se intentase una operación se comenzase a inspeccionar la estructura desde las primeras posiciones, posiblemente jamás se daría una operación entre dos nodos del final de la estructura. Para esta operación la condición es que no exista previamente ningún arco entre esos dos nodos. En el momento en el que se encuentra una pareja de nodos que cumple la condición se detiene la búsqueda y se devuelve la pareja de nodos y un indicador de que la búsqueda ha tenido éxito. Si por el contrario no se encontrase ninguna pareja de nodos que cumpla las condiciones exigidas por la operación, la función devuelve un indicador que señala que la operación no es posible, con lo que se evita perder el tiempo intentando realizar la operación y se vuelve a repetir el proceso desde la selección de la misma.

insertarElementoEnPoblacion La función `insertarElementoEnPoblacion` se encarga de meter en la población al individuo recién creado. Para ello se necesita contar con el individuo, la población, que será introducida por referencia, al igual que el vector de puntuación, pues ambos serán modificados. También es necesario el índice del individuo de la población que ha sido mutado, ya que la posición del hijo creado será la posición del progenitor más el tamaño de la población(p.ej. si la población es de 50 individuos y el padre es el individuo 13, el hijo creado a partir de este será introducido en la posición $50 + 13 = 63$ de la población. El hecho de mantener una estructura para la población que sea el doble del tamaño real de la misma obedece a una más fácil manipulación de

las estructuras. Es mucho más cómodo para la ordenación tener un solo vector y no es necesaria una mezcla de ambos a la hora de seleccionar los mejores individuos. Lo mismo sucede con el vector de puntuación. Para la actualización del vector puntuación también es necesario que se introduzca la puntuación que ha obtenido el hijo como parámetro para la actualización del vector puntuación. En estos pasos previos todas aquellas estructuras que estuvieran en la población o en el vector puntuación serán suplantadas por la nuevas estructuras y por los valores de puntuación de estas.

Después de realizar la mutación, puntuación e inserción de los hijos de cada uno de los individuos de la población pasamos a la ordenación de la población.

ordenarPoblacion La función encargada de la ordenación es `ordenarPoblacion`. Como parámetros de entrada se recogen la población y el vector puntuación pasados por referencia. Una vez en la función tomamos ambas estructuras y con dos bucles *for* anidados desarrollamos el *algoritmo de ordenación de la burbuja.* En él, comenzando por la primera posición del vector de puntuación, comparamos cada una de las posiciones del vector con todas las que tiene por debajo, para ver si hay alguna que la mejore. Si alguna lo hiciera, entonces intercambiamos las posiciones de ambos, tanto en el vector de puntuaciones como en la población. Lo que devuelve la función es la población y el `vectorPuntuacion` con las posiciones correspondientes al tamaño de la población reordenadas, en este caso 50.

almacenadoValoresVectorEstadisticas Lo siguiente es una acción de almacenado de datos, que más tarde utilizaremos para la creación de las gráficas. El almacenado de este tipo de datos se da cada cierto número de iteraciones. En nuestro caso ese número de iteraciones es diez, que es un valor consecuente con el número de iteraciones y bueno para ver la evolución de los diferentes valores a lo largo de la ejecución.

Una de las funciones que hacen este almacenaje es `almacenadoValoresVectorEstadisticas`. Los parámetros que recibe son el `vectorEstadisticas`, en el que se almacena a lo largo de toda la ejecución los valores estadísticos referentes a las operaciones, el `vectorGrafoEstadisticas`, que es pasado por referencia y que será el que almacene los valores que aparezcan en el `vectorEstadisticas` cada diez generaciones y que serán los que se utilizarán luego para realizar las gráficas estadísticas, el contador, que será la marca para saber a partir de que posición hay que comenzar a escribir en el `vectorGrafoEstadisticas` la siguiente ocasión y la generación en la que se origina la inserción de datos. Todos estos datos aparecerán en el gráfico estadístico que se generará con este.

mostrarEstadisticas Tras ejecutar la operación anterior con cada una de las tres funciones de almacenado de datos para gráficas viene la muestra de es-

tadísticas en pantalla. Para ello utilizaremos la función `mostrarEstadisticas`, la cual recibe como parámetros el `vectorPuntuacion` y el `vectorEstadisticas` y que muestra en pantalla diferentes valores de estos dos vectores, como número de operaciones exitosas o estadísticas de puntuaciones.

Una vez que se ha finalizado la ejecución del programa principal, llega el momento de exportar la información para la obtención de gráficos y visualización de los resultados obtenidos.

infoGrafoEstadisticas Existen tres funciones que producen ficheros desde los que luego podremos generar gráficas acerca de la información recogida a lo largo de la ejecución, `infoGrafoEstadisticas`, `infoGrafoPuntuaciones` e `infoGrafoComparaciones`. En el caso por ejemplo de `infoGrafoEstadisticas` recibe como parámetro de entrada el vector `vectorGrafoEstadisticas` y el contador del número de veces que se han hecho inserciones en el vector. A partir de estos valores construimos la estructura del fichero que exportaremos para poder hacer las gráficas, con tantas columnas como valores hayan sido introducidos en cada ocasión más el número de generación en que ocurriera esta introducción y tantas filas como introducciones se hayan realizado. El formato del fichero resultado es .csv, un formato muy simple que se puede visualizar con el programa EXCEL o que puede ser transformado a cualquier otro tipo sin excesiva dificultad.

solucionFinalAFich Por último y antes de salir de la ejecución se debe hacer una última actualización. En este caso la función utilizada es `solucionFinalAFich`. Esta recibe como parámetros la población, el `vectorPuntuacion` y el `vectorEstados`. Este último será necesario para poder pasar los grafos del formato PDAG con el que hemos estados trabajando durante toda la ejecución al formato final de representación en forma de Red Bayesiana, mientras que los otros dos serán necesarios para la toma de estructuras en el caso de la población y de valores que saldan por pantalla en el caso del `vectorPuntuacion`. La toma de las estructuras es necesario puesto que los tres mejores individuos se guardan en sendos archivos con el formato .bif (mejorRed.txt, segundaMejorRed.txt y terceraMejorRed.txt, respectivamente). Pero lo más importante que hace esta función es la llamada a la función `pasoBNAFormatoGraphViz`.

pasoBNAFormatoGraphViz Esta función recibe como parámetro de entrada la mejor red obtenida tras la ejecución del algoritmo. A continuación toma la red y la pasa al formato .dot para que pueda ser leída por la aplicación *GraphViz*, que construye un grafo a partir del fichero .dot con el que se puede comparar visualmente el parecido entre la red objetivo y la mejor conseguida con el algoritmo. El fichero producido se llama `mejorRed.dot`.

comparacionMejorRedRedObjetivo Esta función recibe como parámetros de entrada el mejor individuo de la población en el momento en el que se invoque a la función, el grafo objetivo y un `vectorArcos` pasado por referencia que almacenará el número de arcos que comparten ambas estructuras para un posterior estudio estadístico. Lo que hace la función es comparar las estructuras de ambos grafos a la búsqueda de arcos que se repitan en ambas. La búsqueda es secuencial, esto es, arrancamos del nodo 1 y comprobamos todos los posibles arcos que este tenga y vemos si están presentes en la otra estructura, en este caso la estructura de la red objetivo.

3.4 Implementación

En la parte de implementación, al igual que en las dos secciones anteriores, haremos especial hincapié en las funciones que ya hemos destacado como las más importantes, pero también haremos una pequeña explicación de cada una de las funciones presentes en cada uno de los módulos que componen el proyecto. En el caso del módulo `Principal.c`, en el que está el programa principal, haremos un análisis en pseudocódigo para explicar en detalle el funcionamiento del Algoritmo Evolutivo. Los módulos que repasaremos son:

- `Principal.c`
- `Poblacion.c`
- `FuncAux.c`
- `SeleccionNodos.c`

Lo que se hará en esta parte de implementación será mostrar someramente las funciones más importantes en pseudocódigo.

Principal.c A continuación se muestra la representación en pseudocódigo del algoritmo principal. Hay partes de introducción de datos que no llevan el orden que luego tienen en el algoritmo, pero aquí se ponen juntas para una mayor claridad:

```
Declaracion Inicial de Variables;

/* Inicializacion de los valores de las variables estadísticas del
   programa */
inicializacionVectorEstadisticas(vectorEstadisticas);

/* Seccion de introducción y recogida de datos */
intro();
redSeleccionada=seleccionRed();
tipoPuntuacion=recogidaTipoFuncionPuntuacion();
pesoPuntuacion= recogidaPesoPuntuacion();
```

```
numRepeticiones = recogidaRepeticiones();
semilla= recogidaSemilla(numRepeticiones);
visualizarDetalles= recogidaDetalles();
seleccionParadas=recogidaParadasEjecucion();
seleccionDensidad= recogidaTipoDensidad();

/* Carga de la red objetivo desde fichero */
redAuxiliar=cargaFicheroRedOriginal(redSeleccionada);
numNodos=NumNodosBN(redAuxiliar);

/* Obtencion del vector de estados de los nodos */
obtencionnúmeroEstadosNodos(redAuxiliar,vectorEstados,
                            redSeleccionada);

/* Carga del archivo de patrones */
lp=cargaListaPatrones(redSeleccionada);

/* En este momento de la ejecución comprobamos que tipo de
   algoritmo se ha selecionado para realizar la búsqueda. Si el
   algoritmo seleccionado es el K2 entonces se ejecutan solo las
   siguientes lineas. Sino es el resto del programa.*/
Si (tipoPuntuacion==1) entonces
      ejecucionK2(redAuxiliar,lp,numNodos,vectorEstados);


/* NUEVA DECLARACION DE VARIABLES
Estos dos vectores son declarados aparte del resto de variables
porque su tamaño depende del número de veces que se ejecute el
algoritmo.*/
 int vectorGrafoEstadisticas[(numRepeticiones/10)*9];
int vectorGrafoComparacionRedes[(numRepeticiones/10)*4];
double vectorGrafoPuntuaciones[(numRepeticiones/10)*4];

/* Introduccion de la aleatoriedad de la semilla
aleatoriedadSemilla(semilla);

/* Creacion de las poblaciones
densidadArcos=creacionPoblacion(seleccionDensidad,poblacion,
                                numNodos,vectorEstados,
                                pesoPuntuacion,tipoPuntuacion,lp);

/*Continuacion de la seccion de introduccion y recogida de datos*/
tipoEjecucion=recogidaTipoEjecucion();

/* Muestra en pantalla de los valores recogidos en la parte de
   introduccion de datos */
```

```
muestraValoresIntroducidos(numRepeticiones,semilla,densidadArcos,
                           visualizarDetalles,pesoPuntuacion,
                           seleccionParadas,tipoEjecucion);

/* Creacion, puntuacion y destruccion de la red que se toma como
   objetivo */
grafo=CrearGD();
grafoRedOriginal=CrearGD();
grafo=BN_A_DAG(redAuxiliar);
DAG_A_CPDAG(grafo, grafoRedOriginal);
puntuacionIdeal=PuntuacionPG(grafoRedOriginal, vectorEstados, lp,
                             ok);
DestruirGD(grafo);

/* Obtencion de las puntuaciones de cada uno de los PDAGs
   pertenecientes a la poblacion inicial */
puntuacionPoblacion(vectorPuntuacion,poblacion,vectorEstados,
                    lp,ok);
generacion=0;
condicionSalida=FALSE;

/* Comienzo de los bucles de ejecucion del algoritmo */
para generacion=0 hasta numRepeticiones hacer{
  para i=0 hasta TAMANOPOB hacer{
    /* Inicializamos parámetros y estructuras auxiliares */
    indice=i;
    grafo=CrearGD();
    grafoActual=CrearGD();
    grafoMejorado=CrearGD();
    grafoActual=poblacion[indice];
    hacer{
       flagCambio=0;
       puntos=vectorPuntuacion[indice];
       puntuacion=&puntos;
       puntosAntes=puntos;
       random= seleccionOperador(tipoEjecucion);
       selecciona(random){
          caso 0: vectorEstadisticas[0]=vectorEstadisticas[0]+1;
             nodos=seleccionDeNodosNoAleatoriaIAD(grafoActual,
                                                  numNodos);
             operacionPosible=nodos[0];
             nodoUno=nodos[1];
             nodoDos=nodos[2];
             nodoTres=0;
             if(!operacionPosible){
                if (TRUE==(InsertarPuntADPG(grafoActual,
```

```
                                        grafoMejorado,
                                        nodoUno, nodoDos,
                                        lp, vectorEstados,
                                        puntuacion))){
                puntosDespues=*puntuacion;
                flagCambio=1;
                vectorEstadisticas[1]=vectorEstadisticas[1]+1;
                }
             }
          ; break;

       caso 1 :vectorEstadisticas[3]=vectorEstadisticas[3] + 1;
          nodos=seleccionDeNodosNoAleatoriaIAND(grafoActual,
                                                numNodos);
          operacionPosible=nodos[0];
          nodoUno=nodos[1];
          nodoDos=nodos[2];
          nodoTres=0;
          if(!operacionPosible){
             if (TRUE==(InsertarPuntANDPG(grafoActual,
                                        grafoMejorado,
                                        nodoUno, nodoDos, lp,
                                        vectorEstados,
                                        puntuacion))){
                puntosDespues=*puntuacion;
                flagCambio=1;
                vectorEstadisticas[4]=vectorEstadisticas[4] + 1;
                }
             }
          ; break;

          ... Así con todas las operaciones ...

       por defecto:printf("\n Error en el switch.\n");
                salidaLenta();
                break;
     } /* fin del switch */
   } mientras que flagCambio==0;

   Si visualizarDetalles==seleccionado entonces
      visualizacionDetalles(random, indice, nodoUno,nodoDos,
                            nodoTres,puntosAntes,puntosDespues);

   grafoActual=CrearGD();
   grafoActual=grafoMejorado;
   grafoMejorado=CrearGD();
```

```
        insertarElementoEnPoblacion(grafoActual,TAMANOPOB+i,
                                    vectorPuntuacion,puntuacion,
                                    poblacion);
        grafoActual=CrearGD();
        DestruirGD(grafoMejorado);
        DestruirGD(grafoActual);
        }  /* fin del bucle secundario*/

   ordenarPoblacion(poblacion, vectorPuntuacion);
   comparacionMejorRedRedOriginal(poblacion[0],grafoRedOriginal,
                                  vectorArcos);

   /* Recogida de datos para estadisticas */

   Si (((generacion+11)%10)==0) entonces{
     media=mediaAritmetica(vectorPuntuacion);
     contador=almacenadoValoresVectorEstadisticas(vectorEstadisticas,
                                          vectorGrafoEstadisticas,
                                          contador,j);
     contador2=almacenadoValoresVectorPuntuaciones(vectorPuntuacion,
                                          vectorGrafoPuntuaciones,
                                          contador2, media,
                                          j);
     contador3=almacenadoValoresVectorComparaciones(vectorArcos,
                                          vectorGrafoComparacionRedes,
                                          contador3,j);
     }

   indicador = (generacion + GENESTADISTICA+1)%GENESTADISTICA;
   Si indicador==0 entonces{
      /* Muestra de estadisticas */
     comparacionMejorRedRedOriginal(poblacion[0],grafoRedOriginal,
                                    vectorArcos);

     /* MUESTRA DE ESTADISTICAS

     leerVectorPuntuacion(vectorPuntuacion);
     mostrarEstadisticas(vectorEstadisticas, vectorPuntuacion);
     densidadMediaDeArcos(poblacion,vectorEstados,redSeleccionada);
     }
  generacion=generacion+1;
  if (generacion>=numRepeticiones){
     condicionSalida=TRUE;
     }
}/* fin del bucle principal */
```

```
/* Carga de los ficheros de las estadisticas */
infoGrafoEstadisticas(vectorGrafoEstadisticas,contador);
infoGrafoPuntuaciones(vectorGrafoPuntuaciones,contador2);
infoGrafoComparaciones(vectorGrafoComparacionRedes,contador3);

/* MUESTRA DE ESTADISTICAS */
leerVectorPuntuacion(vectorPuntuacion);
mostrarEstadisticas(vectorEstadisticas,vectorPuntuacion);
densidadMediaDeArcos(poblacion,vectorEstados,redSeleccionada);

/* almacenamos las tres mejores redes y el fichero alarm.dot */
solucionFinalAFich(vectorPuntuacion, poblacion, vectorEstados);

comparacionMejorRedRedOriginal(poblacion[0], grafoRedOriginal,
                               vectorArcos);

/* Destruimos todas las estructuras que hemos utilizado hasta ahora
   en el algoritmo.*/
DestruirGD(grafoActual);
DestruirGD(grafo);
DestruirGD(grafoMejorado);
salidaLenta();
}    /* fin del programa principal */
```

Poblacion.c A continuación se muestran las funciones que existen dentro del paquete `Poblacion.c`:

void ordenarPoblacion(PDAG poblacion[], double * vecPunt);
Ordena la población y el vector de puntuación de mayor a menor.

void obtencionResultadosEstadisticos(double vecPunt[], double vecEst[]);
Obtiene a partir del vector de puntuación resultados estadísticos como los cuartiles, la varianza y la desviación típica.

void mostrarDensidadMediaDeArcos(PDAG pgrafo, int estados[],int redSelecionada);
Muestra la densidad media de arcos que presentan los nodos de los *PDAG* de la población en determinados momentos de la ejecución.

void crearPoblacionInicial(PDAG poblacion[],int numNodos, int estados[], double densidadArcos, int pesoPuntuacion, int tipoPuntuacion);
Crea un vector del tamaño de la población y lo rellena con los *PDAGs* que luego

formarán la población inicial. A cada uno de estos *PDAGs* se les modifica en cada uno de los nodos que los componen el valor del peso y del tipo `dirichlet` de la matriz de adyacencia, solo si la métrica seleccionada es *BDeu*.

double * crearVectorPuntuacion(PDAG población[], ListaPatrones lp, int estados[], int *ok);
Recorre la estructura de la población actual, halla las puntuaciones de cada uno de los grafos que la forman y con ellas forma el vector de las puntuaciones.

void leerVectorPuntuacion(double vector[]);
Lee los valores que tiene el vector de puntuación de la población y los muestra en pantalla.

void InsertarElementoEnPoblacion(PDAG pgrafo, int indice,double vector [], double *puntuación, PDAG población[]);
Inserta un elemento en la población, en un lugar determinado de la misma que se pasa en forma de índice. También se actualiza la posición correspondiente del vector de puntuaciones.

void solucionFinalAFich(double puntuación [],PDAG poblacion[], int estados[]);
Función que pasa a fichero las tres mejores redes presentes en la poblacion tras la ejecución del algoritmo. La primera de ellas, además de pasarla a formato TXT-BNIF, también se pasa a formato .DOT, que es el que utiliza GRAPHVIZ para la representación de redes dirigidas.

void densidadMediaDeArcos(PDAG población[], int vectorEstados[], int redSeleccionada);
Función que muestra en pantalla la cantidad de arcos dirigidos y no dirigidos de cada uno de los individuos que forman la población y compara el número total de arcos de cada uno de ellos y el total de arcos de la red objetivo.

double creacionPoblacion(int seleccionDensidad,PDAG población[], int numNodos, int vectorEstados[],int pesoPuntuacion, int tipoPuntuacion, ListaPatrones lp);
Función que crea la población que será utilizada en el algoritmo dependiendo del modo de obtención de las densidades de los arcos que se le asignarán a los individuos.

void puntuacionPoblacion(double vectorPuntuacion[],PDAG población[], int vectorEstados[], ListaPatrones lp, int *ok);
Función que halla y devuelve el vector de puntuaciones con el resultado de cada uno de los individuos de la población.

FuncAux.c A continuación se resumen las funciones presentes en el paquete `FuncAux.c`:

void intro();
Función que imprime en pantalla una introducción al programa.

void mostrarInformacionOperacion(int índice, int n1, int n2, int n3, double puntAnt, double puntDesp, int operacionRealizada);
Función que imprime en pantalla información de la operación que se ha realizado. En pantalla se muestran los nodos afectados, la operación que se intenta realizar, las puntuaciones antes y después de realizada la operación y si dicha operación hace que se mejore el *PDAG* del que partía.

int elegirProgenitor(double *vector);
Selecciona un progenitor de entre la población por su puntuación. Se selecciona aleatoriamente un pequeño grupo de individuos de la población (en este caso 3 individuos) y se selecciona el individuo que presenta una mejor puntuación como progenitor.

int recogidaTipoFuncionPuntuacion();
Recoge el tipo de función de puntuación que queremos seleccionar, bien *BDeu* o K2.

int recogidaRepeticiones();
Función que recoge el número de veces que el usuario quiere que la aplicación repita el bucle principal de la misma.

int recogidaSemilla(int numRep);
Recoge el valor que servirá como semilla para mantener la aleatoriedad en la creación de los elementos de la población. Sí el valor introducido no es correcto se toma el valor seleccionado en el número de repeticiones.

int recogidaValorAproximacionArcos();
Recoge el valor de aproximación que se desea para la parada del algoritmo. Una vez que la diferencia entre la puntuación ideal y la puntuación del mejor de los elementos de la población es menor que la aproximación insertada por el usuario, entonces se sale de la ejecución del programa.

int recogidaDetalles();
Recoge la decisión de si el usuario quiere ver en detalle cada una de las operaciones que se realizan a lo largo de la ejecución del programa, tanto las exitosas como las que no lo son.

double valorDensidad(int índice);
Devuelve el valor de la densidad que se le aplicará a los grafos a partir del índice introducido como parámetro.

int recogidaTipoEjecucion(void);
Función que recoge el tipo de ejecución que se quiere efectuar, es decir, los operadores que se van a utilizar en la ejecución.

void mostrarTipoEjecucion(int tipoEjecucion);
Función que muestra en pantalla el tipo de ejecución que se ha seleccionado anteriormente.

double seleccionDeDensidades();
Devuelve el valor de la densidad que se le aplicará a los grafos. Primero se hace una petición acerca de cual es la densidad que deseas se utilice en el/los grafos, se pasa un índice a la función valorDensidad y esta devuelve el valor correspondiente.

int recogidaParadasEjecucion();
Función que recoge la decisión del usuario de parar o no la ejecución del algoritmo cada vez que por pantalla se muestren los resultados estadísticos.

void seleccionDeDensidadesAlea(PDAG población [], int numNodos, int vectorEstados[], int pesoPuntuacion, int tipoPuntuacion); Función que asigna a cada uno de los elementos de la población un valor aleatorio de los posibles que pueden tomar las densidades de los arcos. Se utiliza para dar una mayor variedad inicial a la población.

double seleccionDeDensidadesNoAlea(int numNodos, int vectorEstados[], ListaPatrones lp, int pesoPuntuacion, int tipoPuntuacion);
Función que hace un estudio de como de bien se comportan las diferentes posibles densidades de arcos seleccionables para la población. Crea un grupo de individuos y le aplica las diferentes densidades posibles, haciendo una media de las puntuaciones que obtiene cada uno para determinar cúal de las densidades es la que mejor funciona en cada caso.

int recogidaTipoDensidad();
Función que recoge la opción elegida por el usuario para las densidades a utilizar en la creación de las redes. La primera opción es la de seleccionar las densidades de un modo aleatorio, dándole a cada uno de los miembros de la población uno de los posibles valores de las densidades. El segundo modo es la asignación de un valor determinado de densidad para todos los elementos de la población partiendo del estudio de unos cuantos elementos de la población a los que se les da una densidad determinada, se les puntúa y se sabe cual de las densidades utilizadas ofrece unos mejores resultados en cuanto a las puntuaciones obtenidas. La tercera opción es por libre elección del usuario. Él decide cúal es la densidad que le va a aplicar a los grafos en su creación.

int recogidaPesoPuntuacion();

Función que recoge el peso que el usuario quiere asignar a la función de puntuación *BDeu*, siempre que anteriormente haya sido seleccionada esta opción.

PDAG PDAGConsistente(PDAG grafoNoDirigido);
Función que toma el *PDAG* resultante de una mutación y lo pasa a *DAG* para comprobar que el *PDAG* resultante de la mutación admite extensiones y por tanto es correcto. Posteriormente vuelve a pasar este *DAG* a *PDAG* y lo devuelve.

void pasoBNAFormatoGraphViz(BN bn);
Función que toma una Red Bayesiana y la transforma al formato de visualización de gráficos de libre distribución `GraphViz` para poder con este ver la configuración final de la red que ha sido elegida como mejor representante de la población.

void BNActualAFich(BN red);
Función que escribe en el fichero de texto "RedActual.txt" una red pasada como parámetro.

void infoGrafoEstadisticas(int * vecAlmacen, int contador);
Función que pasa los valores del vector que almacena ciertas estadísticas a un formato que pueda ser leído por un editor de gráficos, en este caso EXCEL.

void infoGrafoPuntuaciones(double * vecAlmacen, int contador);
Función que pasa los valores del vector que almacena las puntuaciones más importantes a un formato que pueda ser leído por un editor de gráficos, en este caso EXCEL.

void infoGrafoComparaciones(int * vecAlmacen, int contador);
Función que pasa los valores del vector que almacena los resultados de las comparaciones a un formato que pueda ser leído por un editor de gráficos, en este caso EXCEL.

void grafoABNAFich(PDAG pgrafo, int estados []);
Función que pasa un grafo a Red Bayesiana y a su vez esta es escrita en un fichero de texto llamado "RedMejorada.txt".

ListaPatrones cargaDePatronesDesdeFichero(void);
Función que descarga desde el fichero "Patron.txt" los patrones que luego el programa utilizará para las puntuaciones.

void salidaLenta(void);
Función de salida del sistema. Utilizada para cualquier situación anómala que provoque la salida del sistema o simplemente al llegar al final de la ejecución del mismo. Para poder finalizar la función es necesario pulsar "enter".

void pulsaTecla(void);
Función de pausa. Cada vez que a lo largo de la ejecución se necesita detener esta momentáneamente, se llama a esta función. Para seguir la ejecución del programa es necesaria la pulsación de la tecla "enter".

void pausa(void);
Función de pausa. Diseñada para evitar posibles problemas en la introducción de los parámetros iniciales.

int randomiza(int tope);
Devuelve un valor aleatorio entero entre 0 y el parámetro entero que se le introduce como tope.

BN cargaFicheroRedOriginal(int redSeleccionada);
Carga en una red una información que le es pasada desde un fichero en formato BIF (Bayesian Interchange Format). Devuelve la red en formato BN (Bayesian Network). La red es seleccionada por el usuario al inicio de la ejecución.

ListaPatrones cargaListaPatrones(int redSeleccionada);
Carga desde un fichero una lista de patrones correspondiente a la red seleccionada al inicio de la ejecución por el usuario.

ListaPatrones generacionListaPatrones(BN red, int cantidad, int numNodos);
Genera una lista de patrones a partir de una red inicial, el número de patrones que queramos crear y el número de nodos de la red. Devuelve la lista creada y crea también un fichero donde guarda la lista creada.

int crearIndice(int entero);
Crea a partir del valor presente en ese momento como índice en el bucle *for* del algoritmo un índice para la población. Se toma el valor introducido por parámetro y se haya el módulo de este respecto del tamaño de la población, que en este caso son 50 individuos. Por tanto el valor del índice variará entre 0 y 49.

int sonIguales(int a, int b, int c);
Compara tres números enteros y dice si son o no iguales entre si. Es utilizada para comprobar la validez de los nodos antes de ser estos utilizados en alguna de las partes del programa.

BN creacionRedOriginal(int numNodos, int estados[], double densidadArcos, int semilla);
A partir del número de nodos de la red, el vector de estados, la densidad de los arcos y una semilla para aleatorizar el proceso, se crea una red aleatoria que será la red a buscar. A partir de esta red se puede desarrollar después la lista de patrones que luego utilizaremos para las puntuaciones de las redes.

void mostrarEstadisticas(int * vectorEstad, double vecPunt[]);
Función que muestra los valores de las estadísticas que se van recogiendo de cada una de las ejecuciones. Muestra los valores para cada una de las posibles operaciones, además de medias de resultados, mejores y peores puntuaciones e información sobre cuartiles.

int aproximacionValida(double puntIdeal, double puntMejor, double valorAproximacion);
Halla la distancia entre la puntuación de la red objetivo y el mejor elemento de la población. Si esa diferencia es menor que el valor de aproximación que el usuario ha introducido por consola entonces se devuelve un TRUE y el programa principal detiene su ejecución. Solo se utiliza cuando la salida de la ejecución no atiende al fin de las iteraciones, sino a una aproximación en la puntuación introducida con anterioridad.

void comparacionMejorRedRedOriginal(PDAG grafoActual, PDAG grafoRedOriginal, int vectorArcos[]);
Compara las estructuras del *PDAG* introducido como mejor individuo de la población actual con la de la red objetivo elegida, mostrando por pantalla el número de arcos coincidentes, tanto dirigidos como no dirigidos, que ambas estructuras tienen, así como los arcos invertidos. Devuelve el `vectorArcos`, que contiene los anteriores valores, para el control estadístico.

int almacenadoValoresVectorEstadisticas(int vecEstad[],int *vecAlmacen, int contador, int generacion);
Función que almacena en un vector los valores estadísticos que se dan a lo largo de la ejecución. Estos valores son los que luego se utilizarán para hacer las estadísticas del éxito de las operaciones.

int almacenadoValoresVectorPuntuaciones(double * vecPunt,double vecAlmacen[], int contador, double media, int generacion);
Función que almacena en un vector las puntuaciones que van apareciendo a lo largo de la ejecución. Estos valores se utilizarán para realizar el estudio de como varían las puntuaciones a lo largo de la ejecución.

int almacenadoValoresVectorComparaciones(int vectorArcos[], int vecAlmacen[], int contador, int generacion);
Función que almacena en un vector los valores que van apareciendo a lo largo de la ejecución en cada una de las comparaciones entre la mejor red existente y la red objetivo. Estos valores son los que luego se utilizarán para hacer las estadísticas de la ejecución.

int seleccionOperador(int tipoEjecucion);
Función que partiendo del tipo de ejecución en el que nos encontremos selecciona un operador correcto para dicho tipo de ejecución.

void incrementoExitos(int random, int vectorEstadisticas[]);
Función que incrementa el vector de estadísticas en el apartado de operaciones exitosas para la operación que se haya realizado con éxito.

void visualizacionDetalles(int random, int índice, int nodoUno, int nodoDos, int nodoTres, double puntosAntes, double puntosDespues);
Función que se encarga de mostrar por pantalla los detalles de las operaciones realizadas en la ejecución (puntuaciones, tipo de operación, si mejora o no el resultado anterior).

void obtencionnúmeroEstadosNodos(BN red,int vectorEstados[],int redSeleccionada);
Función que obtiene el número de estados de los nodos, dependiendo de la selección que se haya hecho de la red objetivo.

void inicializacionVectorEstadisticas(int vectorEstadisticas[]);
Inicializa la estructura que almacenará las estadísticas que se den a lo largo de la ejecución. El valor que se dará será cero para todos.

void muestraValoresIntroducidos(int numRepeticiones, int semilla, double densidadArcos, int visualizarDetalles, int pesoPuntuacion,int seleccionParadas, int tipoEjecucion);
Función que muestra por pantalla los valores seleccionados por el usuario con anterioridad.

void ejecucionK2(BN redAuxiliar, ListaPatrones lp, int numNodos, int vectorEstados[]);
Función que da todos los pasos necesarios para que se pueda dar la ejecución del algoritmo K2.

double mediaAritmetica(double vecPunt[]);
Función que a partir del vector de puntuación introducido por parámetro calcula la media aritmética de las puntuaciones.

void aleatoriedadSemilla(int semilla);
Función que aleatoriza el resultado de la función URand mediante unas iteraciones iniciales.

SeleccionNodos.c A continuación se resumen las funciones presentes en el paquete `SeleccionNodos.c`:

int * seleccionDeNodosAleatoria(int numNodos);

Selecciona aleatoriamente tres nodos al azar.

int * seleccionDeNodosNoAleatoriaIAD(PDAG pgrafo, int numNodos);
Selecciona 2 nodos que cumplan las condiciones necesarias para insertar un arco dirigido entre ellos. Esto es, que no exista entre ambos nodos ningún arco.

int * seleccionDeNodosNoAleatoriaIAND(PDAG pgrafo, int numNodos);
Selecciona 2 nodos que cumplan las condiciones necesarias para insertar un arco no dirigido entre ellos. Esto es, que no exista entre ambos nodos ningún arco.

int * seleccionDeNodosNoAleatoriaBAND(PDAG pgrafo, int numNodos);
Selecciona 2 nodos que cumplan las condiciones necesarias para Borrar un arco no dirigido entre ellos. Esto es, que exista entre ambos un arco no dirigido.

int * seleccionDeNodosNoAleatoriaBAD(PDAG pgrafo, int numNodos);
Selecciona 2 nodos que cumplan las condiciones necesarias para Borrar un arco dirigido entre ellos. Esto es, que exista entre ambos un arco dirigido en el sentido que interesa y no exista en el sentido contrario.

int * seleccionDeNodosNoAleatoriaINV(PDAG pgrafo, int numNodos);
Selecciona 2 nodos que cumplan las condiciones necesarias para invertir un arco dirigido ya existente entre ellos. Esto es, que exista entre ambos un arco dirigido en el sentido que interesa y no exista en el sentido contrario.

int * seleccionDeNodosNoAleatoriaVEST(PDAG pgrafo, int numNodos);
Selecciona 3 nodos que cumplan las condiciones necesarias para poder realizar la operacion `Construir V Estructura` entre ellos. Para ello es necesario que entre el primer y el segundo nodo y el segundo y el tercero exista un arco no dirigido, respectivamente.

int * seleccionDeNodosNoAleatoriaSHARE(PDAG pgrafo,PDAG pgrafoCompartidor, int numNodos);
Selecciona un arco perteneciente a un grafo donante y lo inserta en el *PDAG* que se va a mutar si este no presenta ya ese arco en su estructura. No tiene ningún tipo de control en cuanto a si la operación será o no posible en el arco receptor, solo se preocupa de que dicho arco no esté ya insertado.

int * seleccionDeNodosNoAleatoriaSHAREAD(PDAG pgrafo,PDAG pgrafoCompartidor, int numNodos);
Selecciona un arco dirigido perteneciente a un grafo donante y lo inserta en

el *PDAG* que se va a mutar si este no presenta ya ese arco en su estructura. Controla que el arco encontrado en el donante sea dirigido y que en el receptor no haya ningún arco entre los dos nodos que se verán envueltos en la operación.

int * seleccionDeNodosNoAleatoriaSHAREAND(PDAG pgrafo,PDAG pgrafoCompartidor, int numNodos);
Selecciona un arco no dirigido perteneciente a un grafo donante y lo inserta en el *PDAG* que se va a mutar si este no presenta ya ese arco en su estructura. Controla que el arco encontrado en el donante sea no dirigido y que en el receptor no haya ningún arco entre los dos nodos que se verán envueltos en la operación.

3.5 Experimentos realizados

En la parte de experimentación se presenta un análisis de los resultados obtenidos, la búsqueda de las mejores configuraciones para el algoritmo y una comparación con otros métodos de búsqueda, tanto a nivel de resultados como a nivel de requisitos.

La decisión de realizar unas u otras pruebas fue tomada en función de distintos parámetros cualitativos: la novedad del elemento a probar, la importancia del mismo,.... Una de las razones principales del proyecto era la utilización y prueba de nuevas operaciones, así como controlar la posible variación en las distintas tasas de éxito que tenían las operaciones antiguas y si la inclusión de nuevas operaciones variaba esta tasa de algún modo.

Otra de las razones principales del proyecto es la comparación de los resultados obtenidos con el algoritmo con resultados obtenidos con algoritmos de otro tipo, en este caso el K2.

A continuación se da una pequeña entrada de lo que será la batería básica de pruebas a desarrollar:

1. Comparación de resultados y tiempo entre diferentes enfoques algorítmicos (K2 y evolutivo).
2. Evolución de las operaciones a lo largo de la ejecución.
3. Estadística de resultados para diferentes densidades (por arcos conseguidos).
4. Estadísticas de resultados para las diferentes configuraciones (*básica, básica + SHAREAD, básica + SHAREAND y básica + SHAREAD + SHAREAND*).
5. Control y valoración de los nuevos operadores introducidos(*SHAREAD y SHAREAND*).

Todas estas pruebas se realizarán tanto con la red *ALARM* como con la red *INSURANCE*. Cada uno de los posibles procesos de prueba será repetido cinco veces con diferentes arranques para que los resultados obtenidos sean válidos, conservando la aleatoriedad en el proceso.

Dividiremos el análisis en dos partes, los resultados de la red *ALARM* y los de la red *INSURANCE*.

Los diferentes procesos de prueba que realizaremos serán:

- Manteniendo la densidad de los arcos en 0.005 y aplicando las cuatro diferentes configuraciones posibles.
- Manteniendo la densidad de los arcos en 0.1 y aplicando las cuatro diferentes configuraciones posibles.

El hecho de que sean 0.005 y 0.1 los valores elegidos para probar como afecta la densidad de los arcos a la ejecución es debido a que con densidad 0.005 los individuos creados apenas tienen arcos en su estructura, por lo que se empieza prácticamente de cero la búsqueda de la red objetivo. En el caso de la densidad de arcos 0.1 lo que se intenta es empezar formando individuos que tengan más o menos el mismo número de arcos que la red objetivo, para comprobar si eso beneficia un acercamiento más rápido a la estructura objetivo.

Como las posibles configuraciones son cuatro, el número de posibles experimentos diferentes es ocho, teniendo en cuenta que luego cada una de esas ocho posibles combinaciones se repetirá cinco veces para darle diversidad a los resultados.

3.5.1 Red Alarm

La red *ALARM* [14] es una red de 37 variables que representa la monitorización de pacientes en una unidad de cuidados intensivos. La red tiene 46 arcos(figura 3.1, pág. 60), mientras que la clase de equivalencia a la que pertenece tiene 4 de ellos no dirigidos y 42 dirigidos. Es una de las redes más utilizadas en el ámbito bayesiano por el conocimiento que se tiene de la misma, lo que la hace un modelo de análisis muy conocido y de fácil manejo.

Comparación de resultados y tiempo entre diferentes enfoques algorítmicos (K2 y evolutivo).

Antes de hacer una valoración acerca de los resultados obtenidos por uno y otro hay que conocer las diferencias de los dos algoritmos. A parte del diferente enfoque, las diferencias más destacables son que K2 utiliza *DAGs* para su búsqueda, cambiando el resultado a *PDAGs* solo una vez que se ha encontrado, y que necesita un conocimiento a priori del orden de precedencia correcto de las variables, mientras que el Algoritmo Evolutivo no precisa de esta información. En la Red *ALARM*, El algoritmo K2 obtiene muy buenos resultados. En una ejecución cualquiera el algoritmo viene a tardar unos 17 segundos en conseguir la red, con unos resultados en lo relativo a los arcos de 41 arcos dirigidos y 4 no

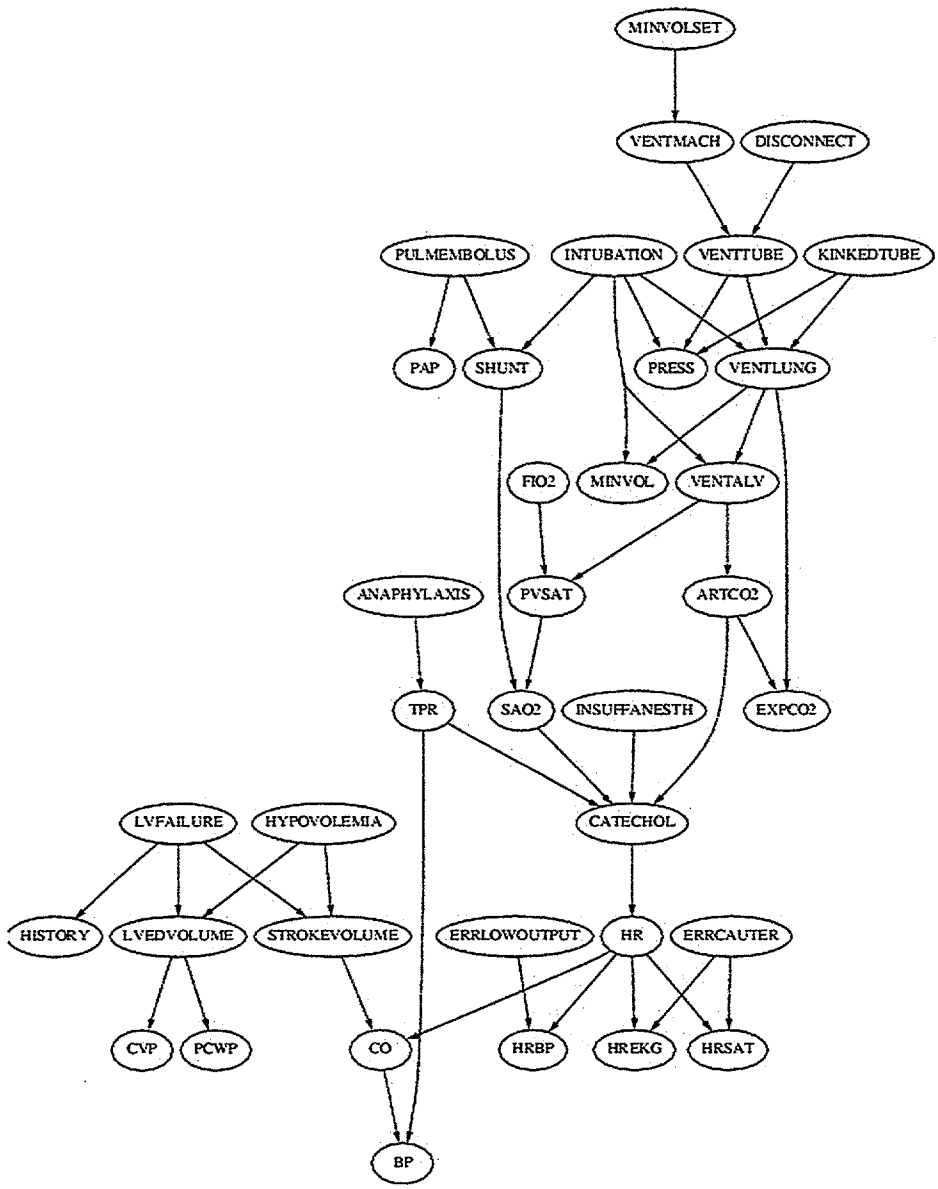

Figura 3.1: Red *ALARM*.

dirigidos, de un total de 45. Esto quiere decir que no hay ningún arco que sobre en la estructura ni ninguno que esté invertido. Esto es, prácticamente igual a la red objetivo.

Por su parte, el Algoritmo Evolutivo tiene unos resultados mucho más modestos que los obtenidos por el K2. La densidad que mejores resultados da es 0.005. En ella, el tiempo medio de ejecución ronda los 380 segundos (6 minutos y 20 segundos, más o menos), y el resultado medio de arcos es de casi 33 arco dirigidos, 3 no dirigidos y 2 invertidos, para un total de 50 arcos, lo que quiere decir que una media de más de 10 arcos de la estructura eran completamente erróneos. Esto, en algunos casos no es completamente cierto. En muchas ocasiones el resultado obtenido es visualmente parecido al objetivo. Lo que sucede es que en determinadas ocasiones se da la circunstancia de que hay arcos no dirigidos en lugares donde deberían estar dirigidos, desfigurando las estadísticas ligeramente. Más adelante, en las próximas comparaciones y en el apartado de comparaciones de estructuras, se podrá apreciar mejor esta tendencia.

Evolución de las operaciones a lo largo de la ejecución.

En esta sección analizaremos cómo varía el grado de operaciones que se realizan con éxito dependiendo del momento de la ejecución en el que nos encontremos o de la densidad de los arcos que se tome, para cada una de las configuraciones posibles. Cuando hablamos de operaciones ejecutadas estamos hablando de aquellas operaciones que han dado como fruto un individuo, independientemente de si este individuo a posteriori ha permanecido o no en la población.

Por ejemplo, para densidades de arcos pequeñas en las que las estructuras iniciales están muy poco densamente pobladas, lo normal es encontrarse curvas de resultados del estilo de la que vemos a continuación (figura 3.2).

En ellas las operaciones de inserción de arcos tienen una especial relevancia al inicio de la ejecución, quedando relegadas las de borrado. Sin embargo, en cuanto el número de arcos en los individuos crece, la tendencia se invierte. En ese momento las operaciones de borrado se hacen muy presentes y relegan a un segundo plano a las de inserción, sobre todo a la de inserción de arcos no dirigidos, que apenas progresa desde los primeros compases de la ejecución.

Tendencias parecidas se pueden ver en los siguientes ejemplos para la densidad 0.005 en la que se introducen las nuevas operaciones de *SHAREAD* y *SHAREAND*, primero por separado y a continuación las dos juntas (Figuras 3.3, 3.4 y 3.5).

Tal vez lo más destacable de estas estadísticas es que pese a la incorporación de las nuevas operaciones los resultados de las otras no varían y que estas aparentemente no interaccionan entre ellas cuando se usan en una misma ejecución, manteniendo sus estadísticas (alrededor de 200 operaciones ejecutadas cada una de ellas).

A continuación podemos ver las mismas estadísticas para ejecuciones que han tenido densidades de arcos con valor 0.1 (Figuras 3.6, 3.7, 3.8 y 3.9).

En las gráficas se pueden apreciar varios cambios. En este caso, desde el inicio las operaciones de borrado son las que más veces se consiguen ejecu-

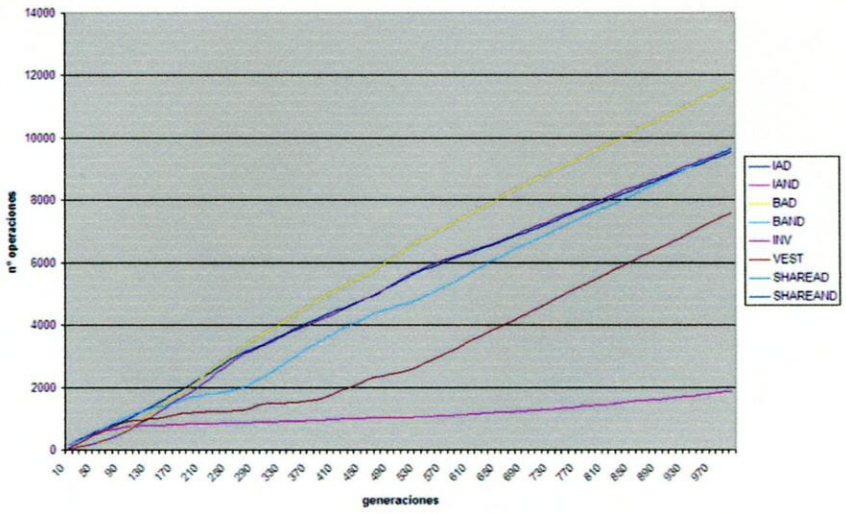

Figura 3.2: Evolución de las operaciones para una configuración *básica* y densidad 0.005.

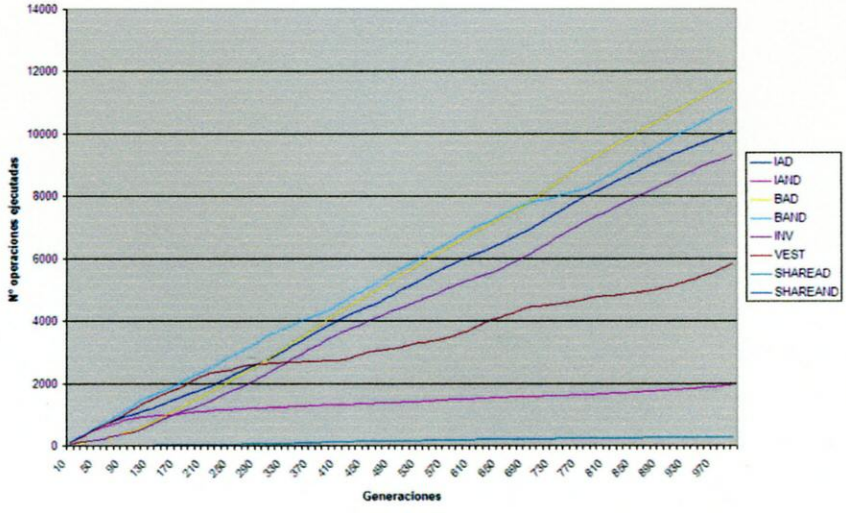

Figura 3.3: Evolución de las operaciones para una configuración *básica + SHAREAD* y densidad 0.005.

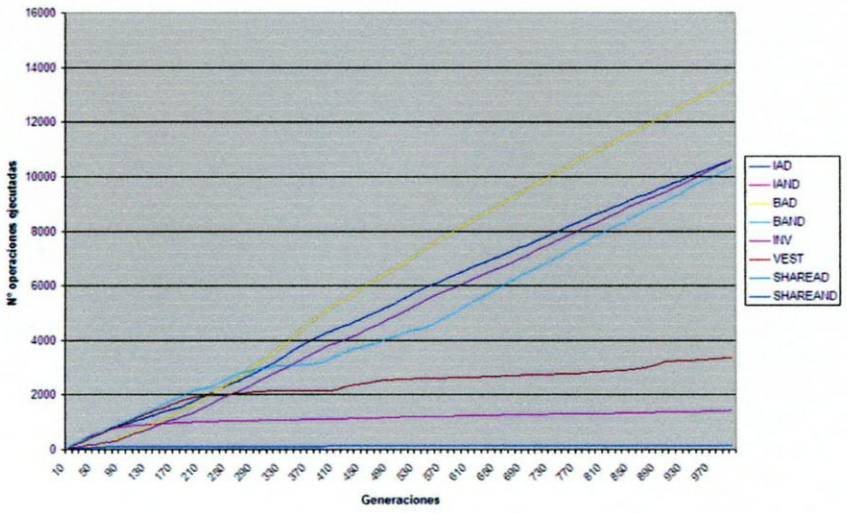

Figura 3.4: Evolución de las operaciones para una configuración *básica + SHAREAND* y densidad 0.005.

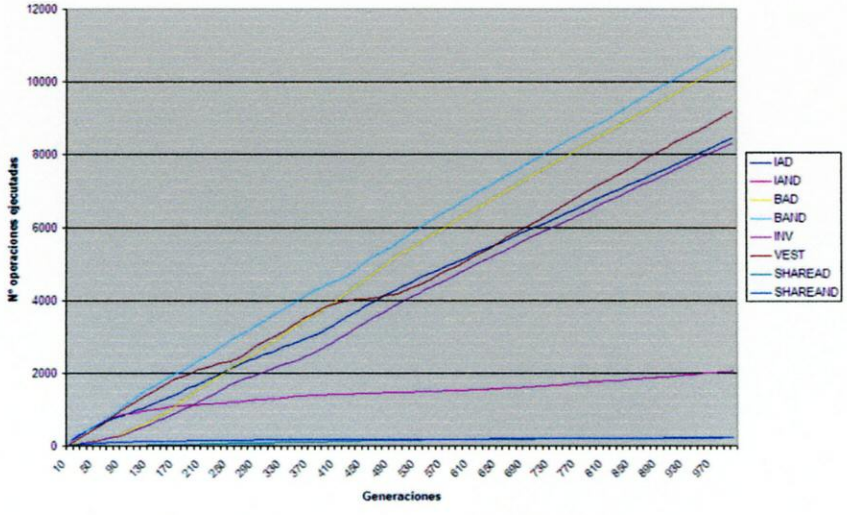

Figura 3.5: Evolución de las operaciones para una configuración *básica + SHAREAD + SHAREAND* y densidad 0.005.

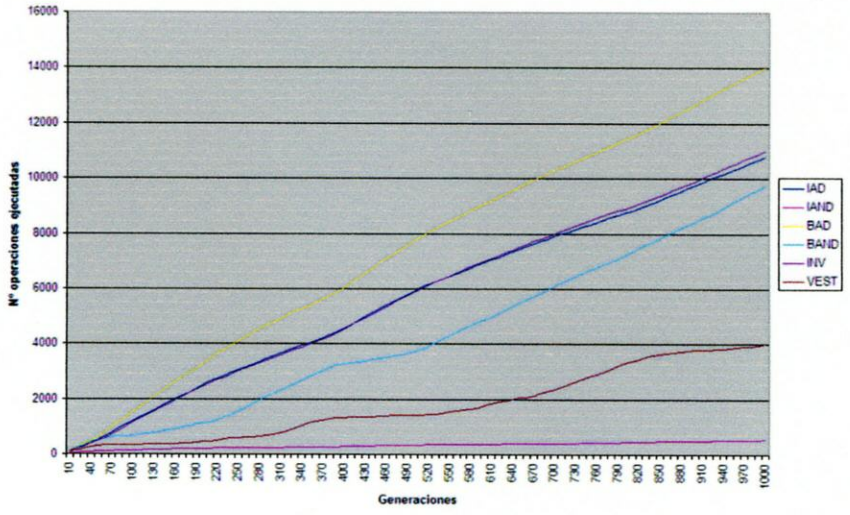

Figura 3.6: Evolución de las operaciones para una configuración *básica* y densidad 0.1.

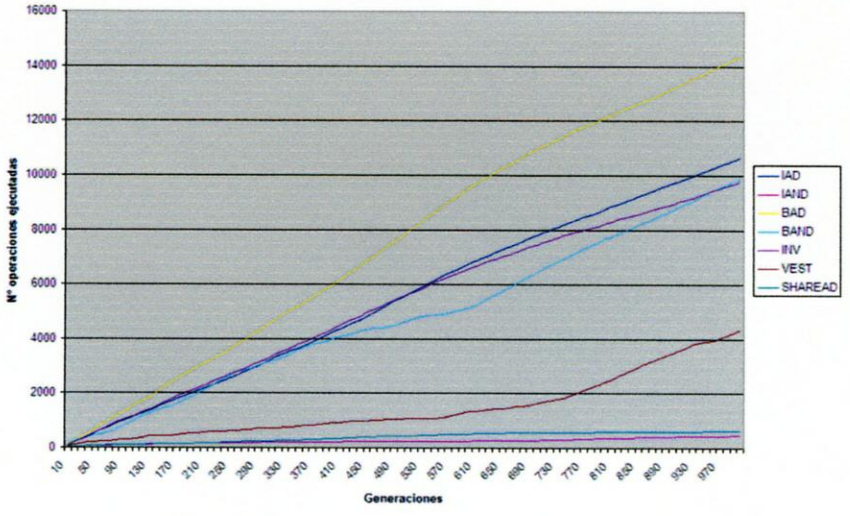

Figura 3.7: Evolución de las operaciones para una configuración *básica + SHAREAD* y densidad 0.1.

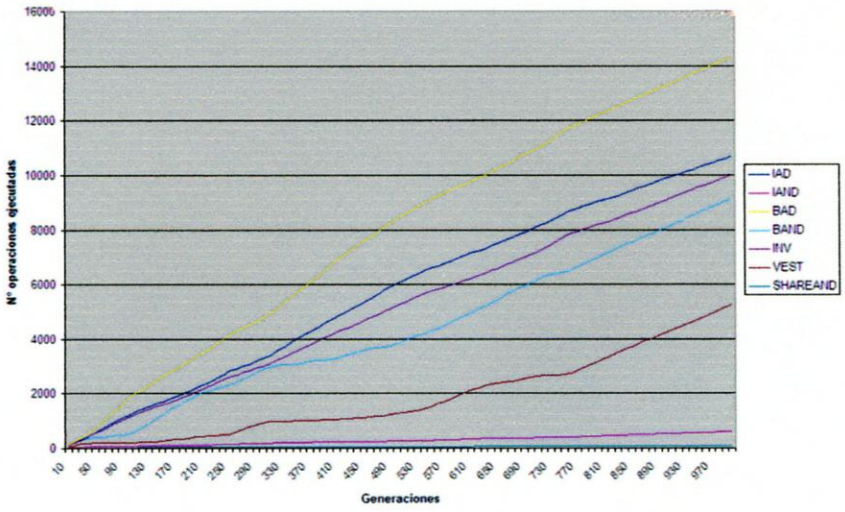

Figura 3.8: Evolución de las operaciones para una configuración *básica + SHAREAND* y densidad 0.1.

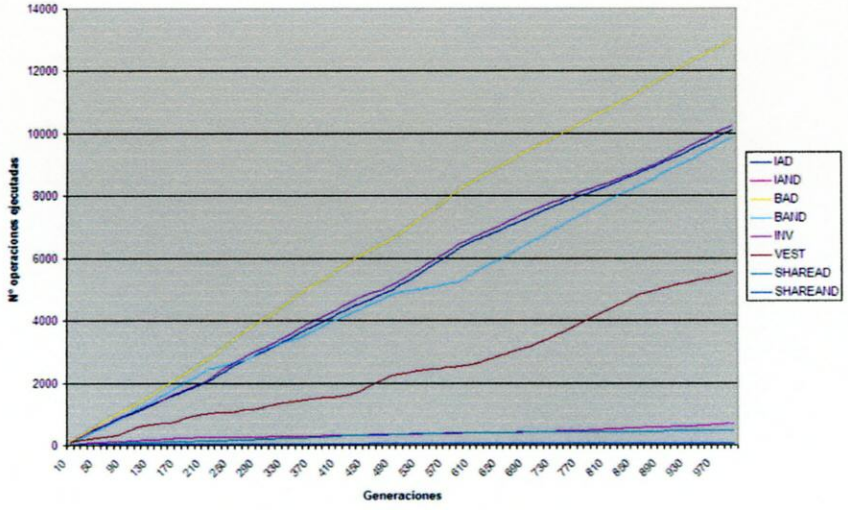

Figura 3.9: Evolución de las operaciones para una configuración *básica + SHAREAD + SHAREAND* y densidad 0.1.

tar relegando a un segundo plano a las de Invertir Arco Dirigido, Construir V Estructura o Insertar Arco Dirigido. Se puede apreciar también el menor rendimiento de la inserción de arcos no dirigidos. En el caso anterior se partía de estructuras que inicialmente estaban vacías. Esto favorece la inserción de los arcos. El hecho es que para poder insertar un arco no dirigido es necesario que no exista ningún arco entre los dos nodos elegidos para ser unidos, pero si entre estos dos nodos ya existe un arco, entonces, pese a que la inserción del arco no dirigido pudiera mejorar la estructura, no se puede realizar hasta que no desaparezca el arco anterior, de ahí que la inserción de arcos no dirigidos sea mucho más complicada.

Esto se extiende a la operación *SHAREAND*, pues por lo anterior el número de arcos no dirigidos en las estructuras es menor y menor la posibilidad de compartir arcos.

Por su parte la operación *SHAREAD* mejora sus resultados levemente por tener desde el principio la posibilidad de compartir algún arco dirigido, cosa que no puede hacer con las densidades de arcos pequeñas.

Estadística de resultados para diferentes densidades (por arcos conseguidos)

Para realizar las pruebas se han elegido dos diferentes densidades, 0.005 y 0.1. La razón de la elección de cada una de ellas es el diferente arranque que ambas proporcionan. En el caso de la densidad 0.005 se tienen redes con muy pocos arcos de inicio. Esto supone un arranque desde un grafo prácticamente vacío. En el caso de la otra densidad, 0.1, se buscaba partir de una densidad de arcos que diera aproximadamente el mismo número de arcos con los que cuenta la red objetivo, entre 45 y 50.

Los resultados obtenidos aparecen en las siguientes tablas (tablas 3.1 y 3.2):

Las conclusiones a las que se pueden llegar son bastante claras. Parece más aconsejable comenzar a partir de una densidad pequeña de arcos que de una grande. A simple vista se puede ver que los resultados para arcos coincidentes son mejores, sobre todo en los arcos no dirigidos. También son menos los arcos que quedan invertidos y bastante menor el número de arcos totales que tienen las estructuras, con lo que se reducen los arcos errados. Aunque también hay casos en los que el hecho de que la densidad de los arcos intente buscar una semejanza desde un inicio puede ser positiva, como en situaciones en las que el número de repeticiones del algoritmo sea limitada. En estos casos nos interesa llegar lo antes posible a la mejor solución posible, y eso se consigue tomando mayores densidades. Para ver esto se puede observar la figura 3.10, que corresponden a la evolución de las puntuaciones para densidades baja y alta de una misma configuración.

En la figura 3.10 se puede ver que la curva que hacen las puntuaciones de los mejores individuos a lo largo de la ejecución son practicamente iguales. Inicialmente las redes con densidades altas tienen una pequeña ventaja, pero esa circunstancia cambia rápidamente y son las de densidades menores las que tienen una curva de crecimiento algo más pronunciadas, para luego tender las

Config.	**Tiempo de ejec.**	**Arcos dirig.**	**Arcos no dirig.**	**Arcos inv.**	**Arcos dirig. total**	**Arcos no dirig. total**
Basica	380.59	32	3.4	2.8	43.8	5.4
Basica + SHAREAD	377.16	34	3.6	1.8	44.6	4.6
Basica + SHARE-AND	384.06	33.4	2.4	2	45.3	6.3
Basica + SHAREAD + SHARE-AND	386.10	33.2	3.2	3.6	47.8	4.0

Tabla 3.1: Media de resultados para la densidad 0.005 (5 ejecuciones por configuración.

Config.	**Tiempo de ejec.**	**Arcos dirig.**	**Arcos no dirig.**	**Arcos inv.**	**Arcos dirig. total**	**Arcos no dirig. total**
Basica	437.23	31.8	1.4	6.6	51.3 57.4	6.1
Basica + SHAREAD	428.73	31.6	1.8	4.8	50.2	5.2
Basica + SHARE-AND	472.60	32.8	2.2	4.2	49.6	5.4
Basica + SHAREAD + SHARE-AND	414.98	32	3	3	42.8	8.2

Tabla 3.2: Media de resultados para la densidad 0.005 (5 ejecuciones por configuración.

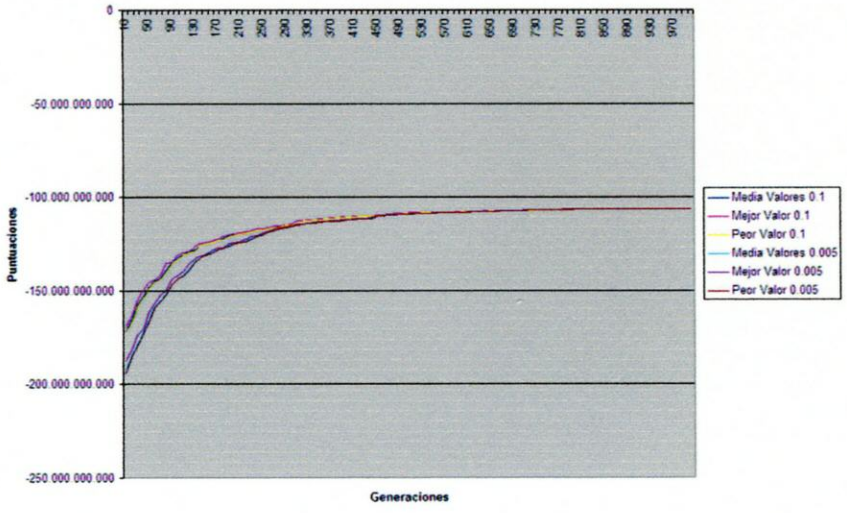

Figura 3.10: Puntuaciones para una configuración *básica* y densidades 0.005 y 0.1.

dos a un mismo limite.

Otra cosa que puede llamar la atención observando las tablas 3.1 y 3.2 (página 67) es la tendencia inversa que tienen las diferentes configuraciones dependiendo de la densidad asignada. Mientras que para la densidad 0.005 las configuraciónes básicas son las que parecen dar un mejor resultado, para la densidad 0.1 es la configuración básica con los operadores Compartir Arco la que lo da. Una vez que se hayan finalizado los experimentos con la Red *INSURANCE* se pasará a profundizar un poco más en los resultados obtenidos y a sacar conclusiones.

Estadísticas de resultados para las diferentes configuraciones (*básica, básica + SHAREAD, básica + SHAREAND y básica + SHAREAD + SHAREAND*)

En los resultados de las tablas 3.1 y 3.2 (Pág.67) se ven las diferencias entre las distintas configuraciones, dependiendo de la densidad de arcos bajo las que trabajen.

Configuración básica La configuración básica tiene buena eficacia con densidades bajas, mientras que para las densidades mayores pierde eficacia. En nuestro caso, y para la densidad 0.1, hubo dos pruebas que arrojaron dos resultados bastante mediocres, mientras que los otros tres mantenían la tónica que se veía en la otra densidad.

Para comprobar el desarrollo de la mejor red en ambas densidades a lo largo del tiempo se puede ver la gráfica que corresponden a las dos mejores pruebas para ambas densidades (figura 3.11).

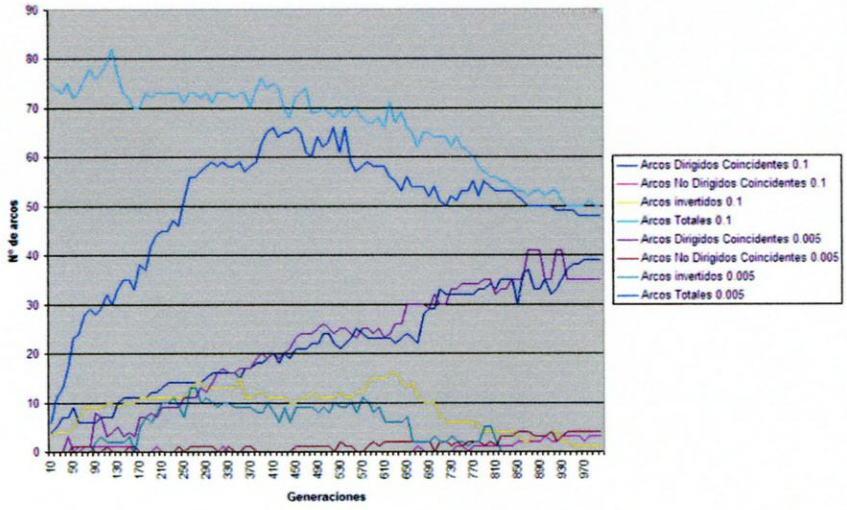

Figura 3.11: Evolución del número de arcos del mejor individuo para una configuración *básica* y densidades 0.005 y 0.1.

En la figura 3.11 se puede apreciar un momento de inflexión a partir de la generación 700 más o menos, en el que una gran cantidad de arcos invertidos parecen colocarse en el sentido correcto y eso permite una mejora considerable del individuo en cuestión.

Configuración básica + SHAREAD Para las densidades bajas esta es la configuración que da mejores resultados. Incluso para los arcos no dirigidos, que en teoría no deberían verse afectados, o al menos no de manera positiva, resulta ser una configuración buena. Sin embargo para la densidad 0.1 esta configuración no es tan buena, o al menos, para esa densidad, cumple lo que se podría prever antes de la realización de las pruebas. Esto es, cuantos más tipos de operadores tienes, mejores resultados obtienes. Y además los arcos no dirigidos se ven afectados negativamente en la ejecución.

Si nos centramos en los arcos dirigidos, que serían los afectados más directamente por el operador *SHAREAD*, indican que para las densidades bajas se cumple lo que se podría esperar, que es la mejora de resultados con respecto a la configuración básica. Para la densidad 0.1 en un primer momento podría parecer que es al contrario. Sin embargo el número de arcos encontrados es menor debido a una muy mala prueba, mientras que el resto de pruebas mantienen un

nivel homogéneo y por encima de la configuración básica.

Configuración básica + SHAREAND En este caso los resultados obtenidos marcan pequeñas diferencias. Para la densidad 0.005 resulta curioso ver que los arcos dirigidos encontrados son bastante pocos, comparados sobre todo con otras configuraciones que deberían tener resultados peores para los no dirigidos. En cambio, para la densidad 0.1 los no dirigidos de esta configuración mejoran los valores de las configuraciones en las que no está presente la operación *SHAREAND*.

Configuración básica + SHAREAD + SHAREAND Para la densidad 0.005 ofrece unos resultados bastantes buenos. Ante esta configuración es necesario tener en cuenta un factor importante, y es el número de intentos que se pueden realizar por tipo de operación. Mientras que en la configuración básica el número total de intentos posibles se dividía entre 6 posibles operaciones, aquí el número de operaciones crece hasta 8. Esto puede suponer una cierta 'incompletitud' en los resultados. Lo que quiere decir es que es posible que para llegar a un resultado equivalente al de la configuración básica, por ejemplo, sería necesario que se hicieran más generaciones de las que se han hecho. Aun así los resultados indican que trabajando con esta configuración se pueden obtener muy buenos resultados. Para la densidad 0.005 obtiene buenos resultados tanto en grafos dirigidos como en no dirigidos, pero tal vez el dato resaltable es el número de arcos que se le quedan invertidos, que puede ser tomado como un síntoma de que los individuos no estaban aun totalmente depurados, al igual que el hecho de que sea la configuración que más arcos totales tiene.

Para la densidad 0.1 los resultados son aun más prometedores. Obtiene el segundo mejor valor para los arcos dirigidos y el mejor para los no dirigidos. Además tiene 3 arcos invertidos que podrían significar mejoras si la ejecución hubiese continuado.

Control y valoración de los nuevos operadores introducidos

Tanto el operador *SHAREAD* como el *SHAREAND* han mostrado buenas líneas de progreso a pesar de su baja efectividad. La razón de esta baja efectividad puede ser debida a la falta de variedad dentro de la población. Si una estructura se extiende con velocidad esto puede afectar muy negativamente a estos operadores, pues las cualidades buenas de los individuos estarán muy extendidas y será complicado pasar información interesante de unos a otros.

La densidad que más favorece a las operaciones ejecutadas por ambos operadores sería la de valor 0.005, no tanto por el número de ejecuciones total entre ambas operaciones, sino porque permite a la operación *SHAREAND* ejecutarse en más ocasiones, con lo que se propicia el paso de información entre individuos. En un primer momento se realizan muchas operaciones *SHAREAND* válidas, para a continuación, y con el llenado de las estructuras de arcos dirigidos, comenzar a aumentar el número de operaciones *SHAREAD* realizadas.

Para la otra densidad la operación *SHAREAND* se realiza en muy pocas ocasiones (menos de 100 en casi todos lo casos), por lo que su influencia decrece, aunque paradójicamente, y posiblemente por el arranque de las estructuras con bastantes arcos, sea aquí donde se pueda notar más la influencia de este operador.

La operación *SHAREAD* consigue una efectividad mucho mayor en configuraciones con densidad de arcos mayores, lo cual es normal si pensamos en que es mucho más posible encontrar un buen arco desde el inicio para poder compartir. En cambio si la densidad es menor, su efectividad baja bastante y es una progresion mucho más moderada, mientras que para las densidades altas a mitad de ejecución ya se ha realizado un elevado porcentaje de las operaciones totales que realizará.

3.5.2 Red Insurance

La Red *INSURANCE* [15] es una red de 27 variables que representa los riesgos en la aseguración de vehiculos. La red (figura 3.12) tiene 52 arcos, de los cuales 18 de ellos son no dirigidos y 34 dirigidos en su clase de equivalencia. Al igual que la Red *ALARM* es una de las redes estándar en el ámbito de las Redes Bayesianas para realizar pruebas y contrastar resultados.

En la figura 3.12 (Pág. 72) se puede ver la forma de la Red *INSURANCE*.

Comparación de resultados y tiempo entre diferentes enfoques algorítmicos (K2 y evolutivo).

En la Red *INSURANCE* el algoritmo K2 no obtiene unos resultados tan sobresalientes como en el caso de la Red *ALARM*. Las ejecuciones con el K2 dan unos resultados de 27 arcos dirigidos y 11 no dirigidos encontrados, siendo el total de arcos encontrados en la red de 43. Los resultados por tanto son bastante peores que los obtenidos por el mismo algoritmo para la Red *ALARM*, puesto que el total de arcos con los que cuenta la red original es de 52.

Al contrario de como sucedía en el caso de la Red *ALARM*, los resultados obtenidos en varios de los experimentos para la Red *INSURANCE* son mejores que los obtenidos por el K2, aunque las medias obtenidas con algo peores que lo obtenido por el K2.

Evolución de las operaciones a lo largo de la ejecución

A continuación se muestran las gráficas de las operaciones ejecutadas en cada una de las configuraciones y densidades probadas (figuras 3.13, 3.14, 3.15, 3.16, 3.17, 3.18, 3.19 y 3.20):

A través de las diferentes gráficas podemos ver que las tendencias que ya se marcaban en las pruebas con la Red *ALARM* se mantienen en la Red *INSURANCE*. No hay apenas diferencias entre las estadísticas en líneas generales. Puede que lo más destacable entre esta tanda de experimentos y la anterior sea la diferencia de protagonismo que tiene la construcción de V estructuras.

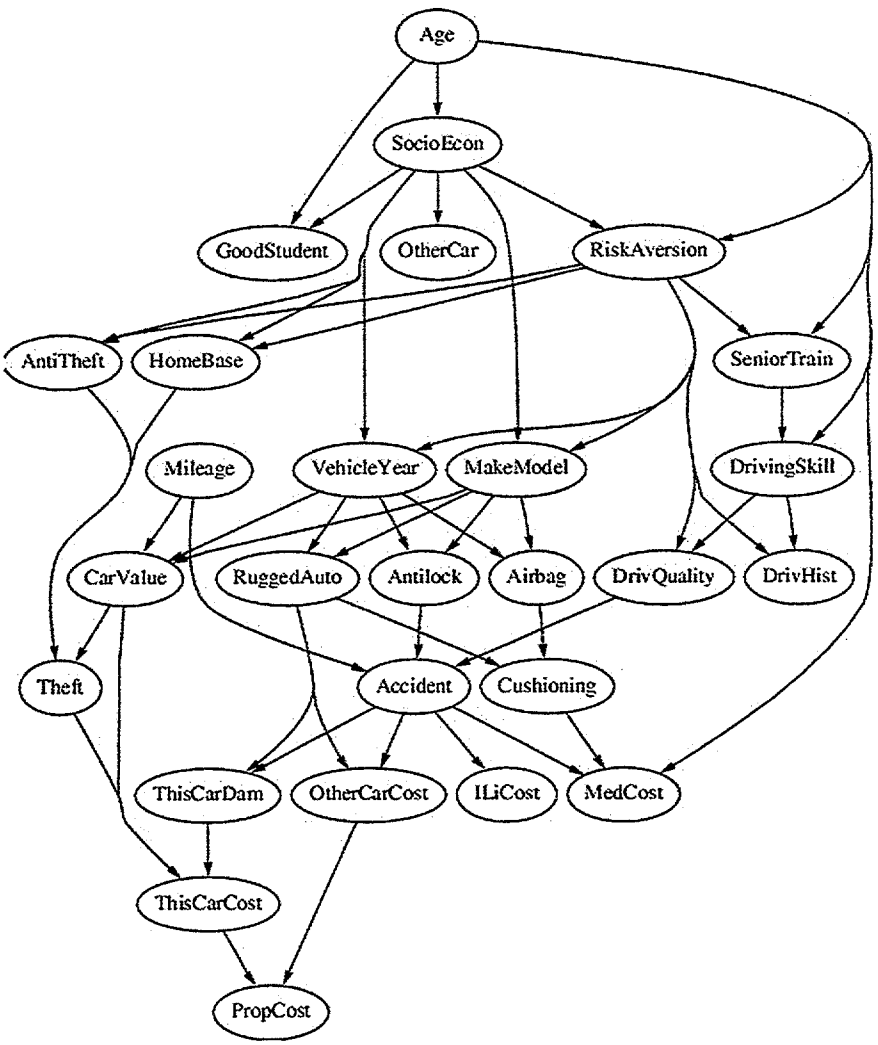

Figura 3.12: Red *INSURANCE*.

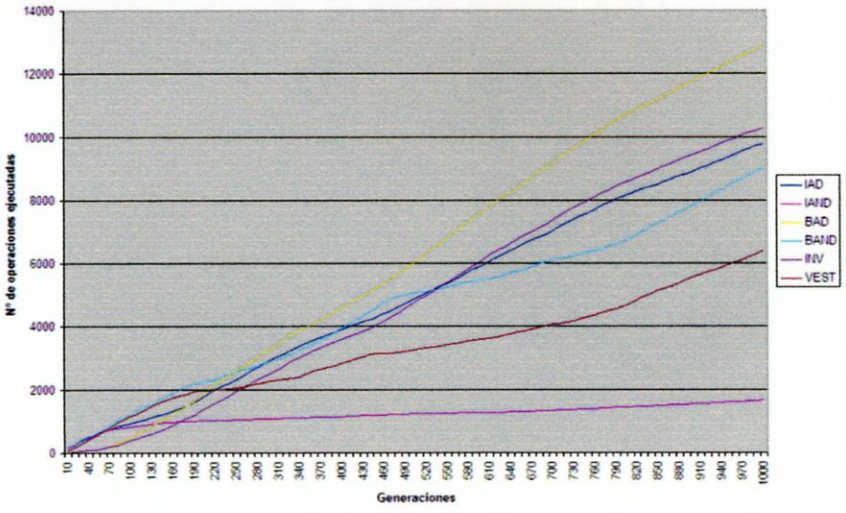

Figura 3.13: Evolución de las operaciones para una configuración *básica* y densidad 0.005.

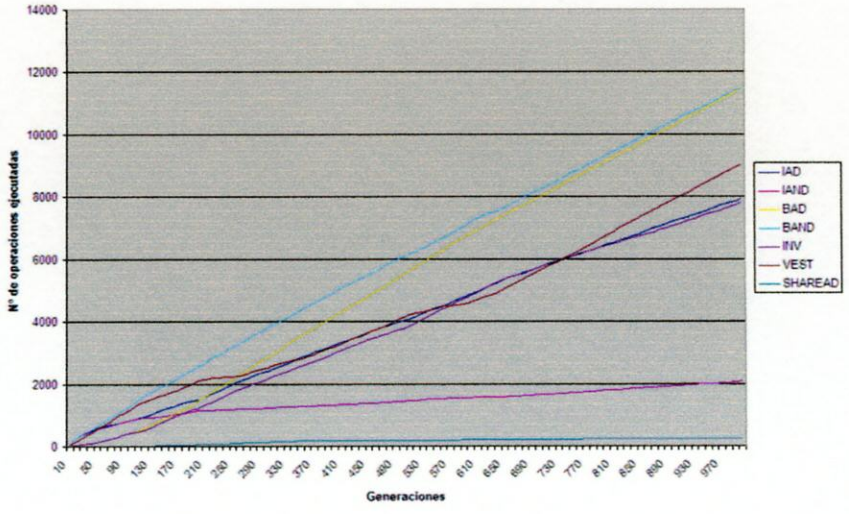

Figura 3.14: Evolución de las operaciones para una configuración *básica* + *SHAREAD* y densidad 0.005.

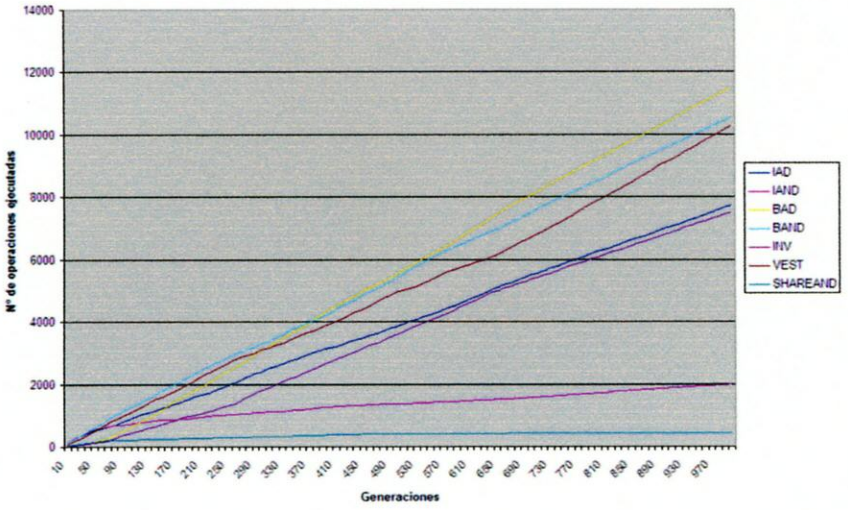

Figura 3.15: Evolución de las operaciones para una configuración *básica + SHAREAND* y densidad 0.005.

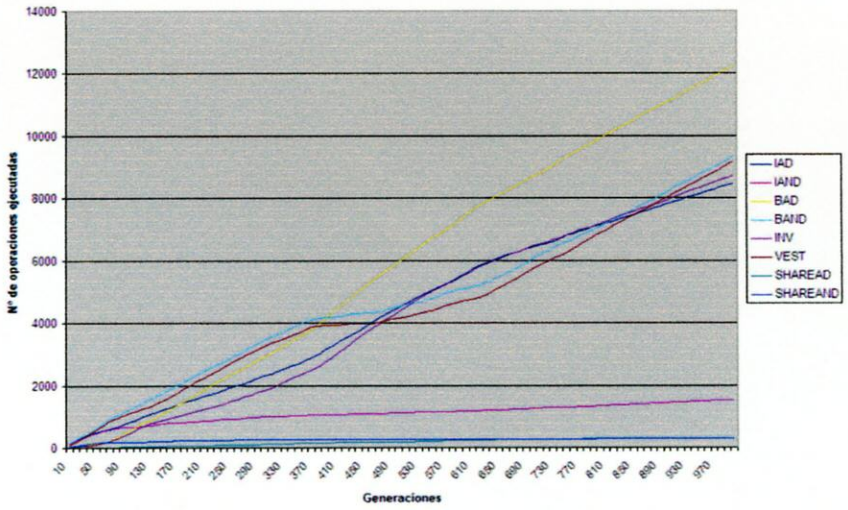

Figura 3.16: Evolución de las operaciones para una configuración *básica + SHAREAD + SHAREAND* y densidad 0.005.

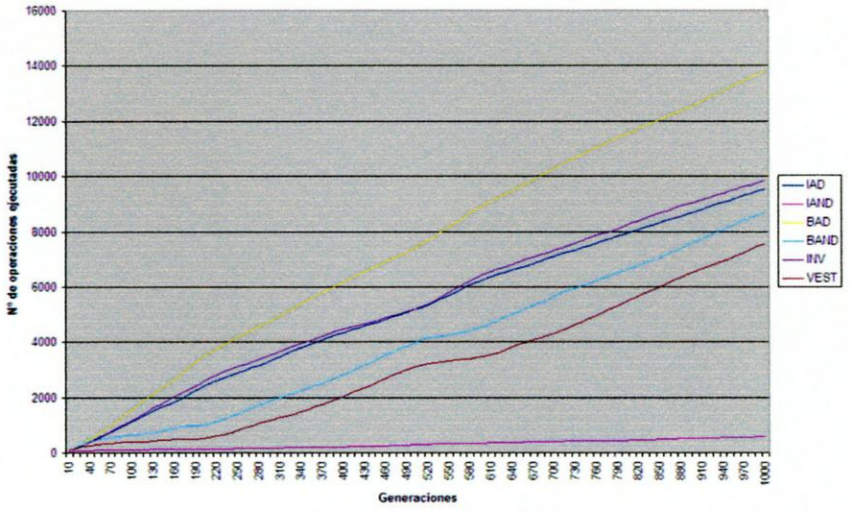

Figura 3.17: Evolución de las operaciones para una configuración *básica* y densidad 0.1.

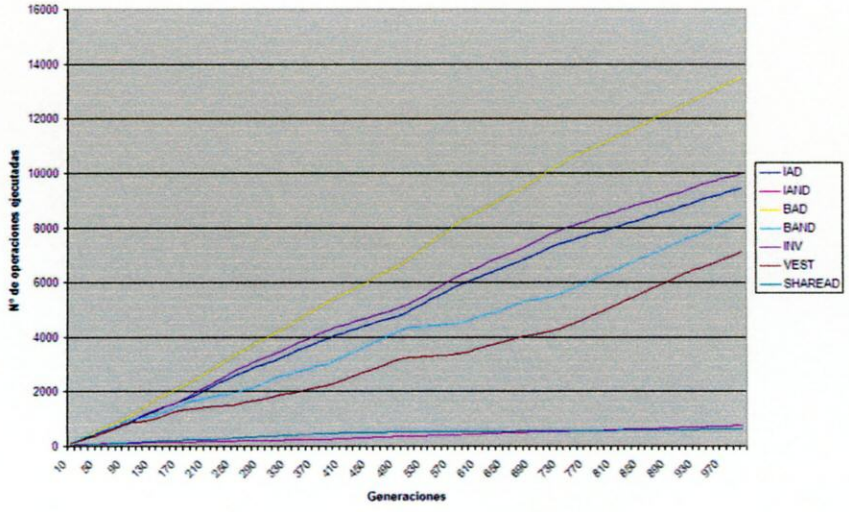

Figura 3.18: Evolución de las operaciones para una configuración *básica* + *SHAREAD* y densidad 0.1.

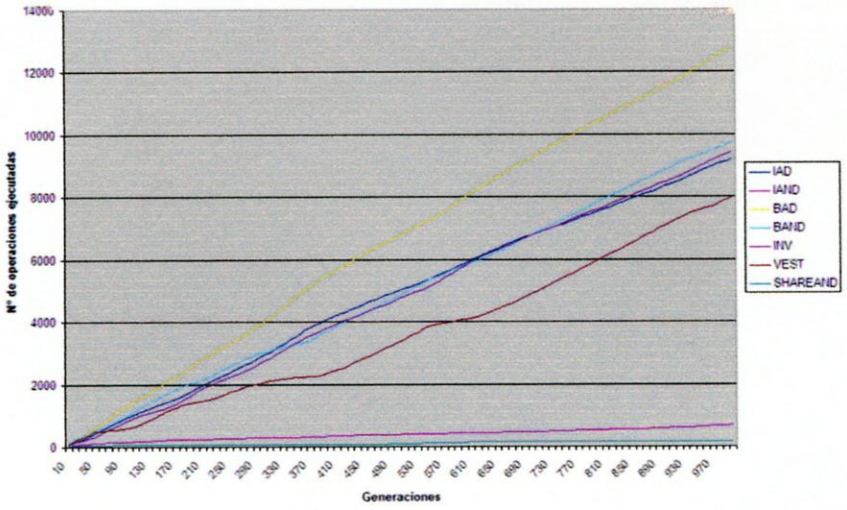

Figura 3.19: Evolución de las operaciones para una configuración *básica + SHAREAND* y densidad 0.1.

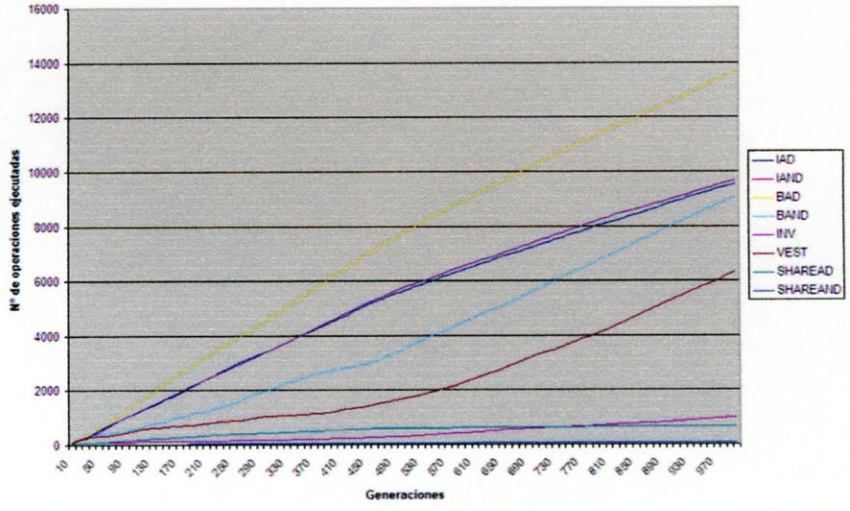

Figura 3.20: Evolución de las operaciones para una configuración *básica + SHAREAD + SHAREAND* y densidad 0.1.

Mientras que en la red anterior era una de las operaciones que más veces proporcionaba individuos a la población, en esta segunda vez queda muy por debajo de los rendimientos obtenidos anteriormente.

Cabe destacar la escasa efectividad de *SHAREAND* cuando la densidad de los arcos es 0.1, pero eso también es una imagen que ya hemos visto en el análisis de la anterior red.

Estadística de resultados para diferentes densidades (por arcos conseguidos)

Para realizar las pruebas se han elegido las mismas densidades que en la red anterior, 0.005 y 0.1.

Los resultados obtenidos aparecen en las siguientes tablas (tablas 3.3 y 3.4):

Config.	**Tiempo de ejec.**	**Arcos dirig.**	**Arcos no dirig.**	**Arcos inv.**	**Arcos dirig. total**	**Arcos no dirig. total**
Basica	269.23	25.6	6.2	1.6	39.8	6.8
Basica + SHAREAD	266.68	24.4	7.6	1.8	35.4	11.4
Basica + SHARE-AND	264.16	27.6	8.4	0	35.2	11.2
Basica + SHAREAD + SHARE-AND	256.63	23.6	6	2.2	39	7.6

Tabla 3.3: Media de resultados para la densidad 0.005 (5 ejecuciones por configuración.

Las conclusiones a las que se pueden llegar son bastante claras. Parece más aconsejable comenzar a partir de una densidad pequeña de arcos que de una grande. A simple vista se puede ver que los resultados para arcos coincidentes son mejores, sobre todo en los arcos no dirigidos. También son menos los arcos que quedan invertidos y bastante menor el número de arcos totales que tienen las estructuras, con lo que se reducen los arcos errados. Aunque también hay casos en los que el hecho de que la densidad de los arcos intente buscar una semejanza desde un inicio puede ser positiva, como en situaciones en las que el número de repeticiones del algoritmo sea limitada. En estos casos interesaría llegar lo antes posible a la mejor solución posible, y eso se consigue tomando mayores densidades. Para ver esto podemos observar la figura 3.10 , que corresponde a la evolución de las puntuaciones para una densidad baja y otra alta con la misma configuración.

Config.	Tiempo de ejec.	Arcos dirig.	Arcos no dirig.	Arcos inv.	Arcos dirig. total	Arcos no dirig. total
Basica	266.18	21.8	3	3.6	42	5.4
Basica + SHAREAD	258.07	22	6.2	3	35.6	10.6
Basica + SHARE-AND	255.69	24.6	2.4	1.6	41.6	4.2
Basica + SHAREAD + SHARE-AND	260.82	23	9	1	31	16

Tabla 3.4: Media de resultados para la densidad 0.1(5 ejecuciones por configuración).

Si nos fijamos en los resultados obtenidos en este caso se puede empezar a apreciar una tendencia en las dos redes. Si trabajamos con configuraciones básicas de 6 operadores es en ocasiones más beneficiosa la utilización de densidades de arcos menores, mientras que para densidades mayores parece que las configuraciones que mejor funcionan son las que utilizan los operadores SHAREAD y SHAREAND. Esto puede ser debido a que estos operadores necesitarían de redes ya desarrolladas para que su efectividad sea real.

Imaginemos el caso de tener una red con una densidad pequeña recién iniciada. En ese contexto es mucho más difícil conseguir un éxito de cualquiera de estas operaciones. Sin embargo, si esto lo intentamos con una red que ya esté medianamente formada, aunque aún no se parezca demasiado al objetivo, los resultados serán mucho mejores. Pese a que en la tabla no se aprecia demasiado bien por culpa de un resultado aislado, la configuración que mejor trabaja de las dos tablas es la que con densidad 0.1 utiliza la configuración en la que aparecen los 8 operadores juntos. Además si nos fijamos en los arcos totales, tanto los dirigidos como los no dirigidos, se puede ver que el valor se asemeja mucho al que tiene la red objetivo (34 arcos dirigidos y 18 no dirigidos), cosa que no hace ninguna de las otras configuraciones.

Queda por tanto pendiente el estudio de la conveniencia o no de la utilización de estos operadores a partir de cierto momento de la ejecución, o a partir de que los individuos presenten un número mínimo de arcos.

En cuanto a las puntuaciones, podemos ver en la comparación de las figura 3.21 que la tendencia de similitud que existía en la pruebas de la primera red se mantiene en esta segunda.

Tampoco existen diferencias remarcables en el comportamiento de las puntuaciones en cuanto a si estamos trabajando con una densidad u otra o estamos

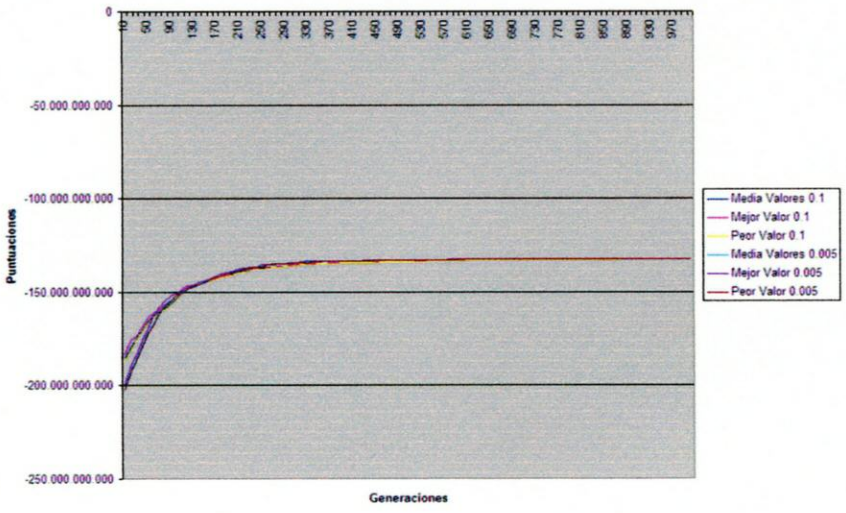

Figura 3.21: Evolución del número de arcos del mejor individuo para una configuración *básica* y densidades 0.005 y 0.1.

utilizando diferentes configuraciones, tanto para la Red *INSURANCE* como para la Red *ALARM*.

Estadísticas de resultados para las diferentes configuraciones (*básica, básica + SHAREAD, básica + SHAREAND y básica + SHAREAD + SHAREAND*)

La comparación de las configuraciones se basa en los resultados que muestran las tablas 3.3 y 3.4 (Pág. 77).

Configuración básica Existe un claro retroceso entre la utilización de una u otra densidad. Se repite lo que sucedió en la tanda de pruebas realizadas con la Red *ALARM*, que parece indicar un mejor funcionamiento de las configuraciones más sencillas con densidades de arcos bajas, mientras que para densidades mayores la efectividad desciende.

Configuración básica + SHAREAD Parece que la tendencia se repite, al igual que en el caso anterior. Los resultados son mejores para densidades más pequeñas, por el ya comentado tema de las configuraciones iniciales. Sin embargo, también parece advertirse un pequeño cambio, una aproximación de las densidades altas a los rendimientos de las densidades pequeñas, que podría ser debido a las operaciones *SHAREAD y SHAREAND*.

Configuración básica + SHAREAND Al igual que en los casos anteriores, la tendencia se repite, y pese a que la diferencia entre una y otra densidad la diferencia es considerable, la distancia cada vez es menor.

Configuración básica + SHAREAD + SHAREAND Lo que en los apartados anteriores era una tendencia en este se hace definitivamente visible. Pese a lo que pueda indicar la media de los resultados mostrados en las tablas, lo cierto es que 4 de los 5 experimentos realizados con esta configuración y densidad son muy satisfactorios, acercándose a los resultados obtenidos por el algoritmo K2, e incluso superándolo en el caso de la mejor de las redes. Además dentro de los experimentos existe una tendencia generalizada a tener el mismo número de arcos que la red objetivo (34 dirigidos y 18 no dirigidos), lo que hace pensar en una posible conveniencia de la combinación de configuraciones con operadores *SHAREAD* o *SHAREAND* con densidades de arcos elevadas o a partir de momentos de la ejecución en los que las redes cuenten con un porcentaje de arcos determinado.

Control y valoración de los nuevos operadores introducidos

Al igual que pasaba en las pruebas anteriores, se mantienen las tendencias de los nuevos operadores.

Mientras que el operador *SHAREAD* funciona mejor con densidades altas, en las que se puede dar con mayor frecuencia el intercambio de buenos arcos, el operador *SHAREAND* tiene una mayor tasa de operaciones exitosas cuando las redes iniciales son poco pobladas.

Más allá de las tasas de éxito que cada uno de los operadores por separado puedan obtener, lo que sí que se aprecia son las mejoras que la inserción de estos operadores supone para las redes obtenidas.

No importa la red sobre la que se hable o la densidad que se esté utilizando, se perfila una cierta tendencia de mejora de los resultados a medida que se van introduciendo dichos operadores en las configuraciones. El problema de que no se vea más claramente esta tendencia en los resultados puede ser la poca efectividad de los operadores.

En esta dirección, y como propuesta de futuros trabajos, despunta la idea de entrar a valorar cualidades fenotípicas del arco que va a dar el grafo donante. En la actualidad el algoritmo solo se basa en la posibilidad de que se pueda hacer la operación y o no, sin fijarse en si el arco que se va a pasar contiene buena información, por lo que se puede dar el paso información poco relevante o importante entre los grafos.

La otra posibilidad es la de aumentar el peso probabilístico de las operaciones para determinados momentos de la ejecución. Por ejemplo, si una de ellas trabaja más a menudo al principio de las ejecuciones cuando la densidad de los arcos es baja, entonces aprovechar esa circunstancia y tratar de hacer que esa operación se de el mayor número de veces posible al inicio de las ejecuciones de ese tipo de configuraciones, sin que el usuario tenga que conocer ninguno de estos detalles.

En nuestro caso las tasas de éxito de las operaciones eran tan sumamente bajas que cada vez que se intenta cualquiera de las dos operaciones se entra en un bucle de 20 repeticiones, del cual solo se puede salir si se da una operación exitosa o si se cumplen las 20 iteraciones del bucle.

Lo que no queda claro cual es el índice que asignarle al bucle. Un valor demasiado alto puede retrasar en exceso la ejecución y no mejorar el resultado tanto y un valor demasiado bajo puede tener el efecto contrario, no afecta apenas al tiempo de ejecución, pero tal vez tampoco al resultado. Después de probar valores entre 10 y 50, se llegó a la decisión de tomar ese valor 20 como índice. Ese valor podría ser otra razón de prueba y estudio del algoritmo.

Capítulo 4

Conclusiones

4.1 Logros alcanzados

Se puede llegar a la conclusión de que los mayores logros alcanzados en este proyecto son cinco:

1. Estudio del contexto teórico.
2. La prueba y validación de nuevos operadores de mutación en el contexto de la inducción de modelos gráficos para Redes Bayesianas utilizando Algoritmos Evolutivos.
3. Analisís del mérito de los nuevos algoritmos.
4. Selección, aprendizaje y uso de diferentes herramientas y aplicaciones(*Dev-C++, GraphViz, Latex, C,...*).

4.1.1 Contexto Teórico

La primera necesidad que hubo al inicio del proyecto fue la *inmersión* teórica en el mundo de las clases de equivalencia. Para ello fue necesario revisitar clásicos conceptos estadísticos, sobre todo los concernientes a la Estadística Bayesiana. Partiendo de esa premisa inicial, se prosiguió con el estudio de los grafos y de la relación de estos con las características bayesianas antes mentadas.

Lo siguiente fue la introducción a las definiciones de grafo y sus sistemas de puntuaciones, los conceptos de clases de equivalencia, según la cual se puede agrupar a grafos por su estructura pese a que esta no sea exactamente igual, y la importancia de mantener un control exhaustivo sobre estas como medio de optimización del algoritmo.

4.1.2 Nuevos Operadores de Mutación

Una de las principales innovaciones del proyecto es la inclusión dentro de este esquema de la programación evolutiva la figura de los operadores *SHARE*.

El hecho de que la información pueda pasar de un miembro a otro de la población a través de estos operadores, mediante la utilización de arcos procedentes de un donante utilizables en un receptor, amplia las miras del enfoque.

Ahora se maneja la opción de que la información se puede transferir entre los mismos miembros de la población y no solo que la información llegue de manera aislada a cada uno de los individuos. Esto, tomando como grafo donador genérico a uno de los individuos de la población que tenga una puntuación alta, hace que la posibilidad de traspaso de información buena entre individuos sea alta.

4.1.3 Análisis del Mérito de los Nuevos Algoritmos

Tal vez el mayor mérito que se les pueda aplicar a estos algoritmos sea ser una alternativa válida a los enfoques voraces, como el K2, sobre todo ante problemas en los que no se tiene el orden de precedencia de los nodos dentro de las estructuras.

Incluso partiendo con esa desventaja inicial, el algoritmo ha demostrado ser capaz de dar buenos resultados a la búsqueda, incluso mejorando repetidas veces los resultados obtenidos por K2 para la Red *INSURANCE*.

Por tanto, se puede hablar de resultados positivos para las pruebas realizadas.

4.1.4 Selección, Aprendizaje y uso de Diferentes Herramientas y Aplicaciones

Más allá del punto de vista teórico y empírico respecto a lo logrado por el proyecto, es valorable la asimilación de conceptos y nuevas técnicas que se han tenido que hacer.

El lenguaje seleccionado fue el *ANSI C*, a propuesta del tutor del proyecto y siguiendo la línea de lo desarrollado anteriomente en la libreria sobre los *PDAGs* utilizada [6].

El ambiente de desarrollo seleccionado fue *Dev-C++*, una herramienta de libre distribución válida para el desarrollo de proyectos tanto en lenguaje *C* como en *C++*.

El visualizador, tambien de libre distribucion, *GraphViz*. En este caso, además de aprender el manejo de la aplicación fue necesario el estudio de los formatos utilizados por la aplicación, para poder ver los resultados obtenidos en forma de grafo.

La hoja de cálculo *EXCEL* para realizar las gráficas de las estadísticas.

Por último, la utilización y aprendizaje del editor de texto LaTeX [12], que ha sido de gran utilidad y ha facilitado enormemente toda la parte de fórmulas matemáticas presentes en el primer capítulo.

4.2 Trabajos Futuros

En este sentido existen diferentes vertientes de progreso. Los principales son:

1. Variantes fenotípicas.
2. Otros operadores: cruce vía Scatter Search.
3. Idea de frontera de inclusión(Chickering(2002b)).
4. Diferentes probabilidades de selección de operadores según el instante de ejecución en el que se encuentre el algoritmo.

4.2.1 Variantes Fenotípicas

El concepto de fenotipo da un paso más allá en el concepto de la orientación evolutiva de los algoritmos.

Cuando se trabaja con variantes genotípicas solo nos interesa el cambio, más o menos aleatorio, mediante los operadores de mutación, de los elementos de la población. Sin embargo, con la introducción de las variantes fenotípicas se da un paso en cuanto a la complejidad de la decisión de la mutación, puesto que nos comenzamos a fijar en particularidades de la mutación en lo referente a la puntuación.

La idea es que ahora también se contaría con una medida de bondad de los arcos. Entonces, cada vez que se tuviera que hacer una operación de borrar o de compartir podriamos ver cuales son los arcos que mejor o peor puntuación tienen y realizar la operación con ellos.

4.2.2 Otros Operadores: Cruce Vía Scatter Search

El Scatter Search [8] [9] es una técnica según la cual, partiendo de una población inicial y mediante combinaciones de los individuos que la componen, se crean nuevos individuos que, si pasan la criba de puntuación, pasarán a formar parte de la población, y que a su vez serán creadores de nuevos individuos. Cada uno de estos nuevos elementos creados será un nuevo elemento *representativo*, un centro de referencia del cual partir para las siguientes generaciones de elementos, una especie de máximo local.

La ventaja que tiene Scatter Search frente a los Algoritmos Evolutivos es que no es necesario tener una población demasiado elevada para mantener la diversificación de individuos.

4.2.3 Idea de Frontera de Inclusión

Dentro del mismo año en el que salió el trabajo acerca de los seis operadores básicos para la mutación de clases de equivalencia, Chickering planteó una segunda posible aproximación a la solución del problema. La *frontera de inclusión* [?] es el conjunto de clases adyacentes a una dada.

La nueva idea aporta un grado de completitud mayor que la que da la de los operadores a la hora de alcanzar todas las posibles clases que se encuentran alrededor de otra.

Si tomáramos una clase de equivalencia cualquiera, alrededor de esta se encontrarían una serie de clases adyacentes, alrededor de cada una de las cuales se encontrarían sus respectivas clases adyacentes. Pues bien, con los operadores actuales se cubre gran parte pero no el total de las clases adyacentes que tiene cada clase. Esto supone una pérdida de potencia, puesto que nos dejaríamos sin comprobar determinadas clases que a posteriori podrían llegar a la solución.

En [?] aparecen esos nuevos operadores.

4.2.4 Diferentes Probabilidades de Selección de Operadores

A lo largo del algoritmo la selección de los diferentes operadores para la mutación es un proceso totalmente aleatorio. La idea que aquí surge es, viendo los comportamientos a lo largo de la ejecución, si en determinados momentos de esta no sería conveniente darle más peso a alguna de las opciones.

Por ejemplo, en algunos casos se toma como opción inicial el tomar redes que estén vacías o prácticamente vacías de arcos. En esta situación no es comprensible que, al menos hasta pasadas algunas iteraciones del algoritmo, se proceda a hacer algún tipo de operación que no sea una inserción.

También se ha pensado en hacer algo parecido en la parte final de las ejecuciones con el operador Invertir Arco Dirigido o los operadores *SHAREAD y SHAREAND.*

Apéndice A

Requisitos del sistema

El programa ha sido ejecutado en varias máquinas, con diferentes resultados.

Pese a que el tamaño del programa no es demasiado grande (el ejecutable viene a ocupar algo así como 120 KB), los requerimientos de memoria si lo pueden ser en determinados momentos. Un ejemplo de esta situación se puede tener si se realiza una ejecución con una densidad de arcos elevada, 0.1 por ejemplo. Se debe tener presente que en memoria hay un mínimo de 50 individuos de la población, cada uno de los cuales contará con un número de arcos bastante elevado por nodo, por lo que el espacio necesario en memoria para alojar todos esos datos aumentará considerablemente.

También son considerables los recursos de computación que utiliza, por lo que sería más que aconsejable correr la ejecución en una máquina que contará con un procesador medianamente potente, digamos 1 Ghz, para no eternizarla. La máquina más pequeña en la que ha sido probado con éxito el programa ha sido un Pentium MMX a 166 Mhz con una memoria de 32 MB. Lo cierto es que en esa máquina la ejecución era tan sumamente lenta que solo se probó una vez a modo de testeo.

Las pruebas se han realizado en una máquina con un procesador a 2.8 Ghz y con 512 MB de memoria. Como ya se ha dicho antes, una máquina que tenga un procesador de 1 Ghz y 128 MB de memoria no debería tener grandes problemas para hacer correr el programa con agilidad.

Apéndice B

Guía del usuario

En el disco compacto que se entrega con la memoria hay cuatro apartados:

- *Memoria*, donde se encuentra todo lo relacionado con la documentación entregada en el proyecto.
- *Pruebas*, que tiene todos los ficheros generados en cada uno de los experimentos realizados para probar el algoritmo.
- *Código*, en el que se encuentra el código final desarrollado, junto con la librería proporcionada por [6].
- *Aplicaciones*, en el que se encuentran las aplicaciones utilizadas y necesarias para el desarrollo del proyecto y la visualización de los resultados.

Es esta última parte de código la que nos va a interesar y sobre la que tratará la guía del usuario.

La guía del usuario se puede dividir en los siguientes puntos:

1. *Instalación*
2. *Ejecución*
3. *Utilización*
4. *Valoración de resultados*

B.0.5 Instalación

En la carpeta código aparece un archivo llamado *evolutivo.zip*. Para instalar el programa que va en su interior será necesario darle un directorio de destino dentro de la máquina donde se hará la ejecución. También es posible correrlo directamente desde el disco compacto, pero al final de la ejecución dará unos errores de apertura de ficheros, puesto que tiene que escribir los resultados obtenidos para su posterior análisis y no puede hacerse directamente en el disco

compacto. Lo aconsejable sería crear o buscar una carpeta donde poder descomprimir el fichero *evolutivo.zip*. Para ello se puede utilizar el *winzip* o el *winrar*, por poner dos ejemplos de compresores que pueden descomprimir el archivo en cuestión.

B.0.6 Ejecución

Una vez que se tenga descomprimido el fichero dentro de la carpeta lo siguiente será ejecutar el programa. Dentro de la descompresión realizada tiene que aparecer un archivo llamado *Proyecto.exe*. Ese es el archivo que contiene el ejecutable y que se debe abrir para que se lanze la aplicación.

B.0.7 Utilización

Una vez que arranca la ejecución, en la pantalla aparecerá algo como lo que sigue a modo de introducción:

```
printf("\n*-------------------------------------------------*");
printf("\n*-                  Algoritmo Evolutivo          -*");
printf("\n*-         para la Obtencion de Modelos gráficos -*");
printf("\n*-                  de Redes Bayesianas          -*");
printf("\n*-                 a partir de Patrones          -*");
printf("\n*-                                               -*");
printf("\n*-                   Enrique López               -*");
printf("\n*-                   Septiembre 2004             -*");
printf("\n*-------------------------------------------*\n\n\n");
```

A continuación comenzarán a aparecer preguntas acerca de algunos parámetros y decisiones que se deben tomar con respecto a la ejecución.

- *Red que desea utilizar.* Para la ejecución del programa se necesita una red de referencia. Las posibilidades que se ofrecen en la ejecución son dos: La Red *ALARM*, de 37 nodos, 42 arcos dirigidos y 4 no dirigidos (46 arcos en total) y no muy densamente poblada de arcos, y la Red *INSURANCE*, de 27 nodos, 34 arcos dirigidos y 18 no dirigidos (52 arcos en total), con mayor densidad de arcos que la *ALARM*. Cualquiera de ellas dos serán los objetivos hacia los que apuntarán las ejecuciones.

  Si la selección se hace incorrectamente, se seleccionará de manera automática la Red *ALARM*.

- *Tipo de Puntuación.* Las dos posibles opciones de puntuación son K2, que es el enfoque no evolutivo con el que se compara nuestro programa y que está basado en la búsqueda iterativa sobre una misma red de la estructura, y el enfoque evolutivo. Si se selecciona esta opción, ya no se pedirá la introducción de ningún parámetro más. Devolverá un mensaje en el que muestre el número de arcos dirigidos, no dirigidos e invertidos

que se han encontrado con respecto a la red original y un mensaje que informa acerca de la finalización de la ejecución.

Por otra parte está el tipo de puntuación *BDeu*. Si se selecciona este tipo de puntuación se recogerán más parámetros, que se verán a continuación.

Si la selección se hace incorrectamente, se seleccionará de manera automática la puntuación *BDeu*.

- *Peso de la puntuación.* El peso de la puntuación que recibe la función *BDeu* condiciona los valores de la matriz de probabilidad condicionada. El valor que se usa por defecto, y que se utilizará en todas las pruebas del proyecto, es 1, aunque también son usuales valores entre 0.01 y 10 para el peso.

  Si la selección se hace incorrectamente, se seleccionará de manera automática el valor 1.

- *Número de repeticiones.* En este apartado se debe introducir el número de repeticiones o *generaciones* que queremos que tenga nuestro algoritmo. En las pruebas realizadas se utilizó el valor de 1000 repeticiones, pero la elección del usuario es libre.

  Si la selección no se hace correctamente, se seleccionará de manera automática el valor 1000.

- *Semilla de la aleatoriedad.* Este valor proporciona una variedad inicial en las redes creadas para la población. Dependiendo del valor que se le introduzca saldrá una u otra combinación de individuos en la población.

  Si la selección no se hace correctamente, se seleccionará de manera automática el valor que se le haya dado al número de repeticiones.

- *Visualización de las operaciones realizadas.* En este apartado se decide si el usuario desea ver los resultados de cada una de las operaciones que tienen lugar en el seno del programa, y en las que se informa de una serie de valores:

  - Operación que se va a realizar.
  - El índice del *PDAG* que se va a mutar.
  - Los nodos afectados en la operación.
  - La puntuación que tenía el individuo que hace las veces de padre.
  - La puntuación que tiene el nuevo individuo hijo después de la operación.
  - Si el hijo mejora la puntuación que tenía el padre.

Si el usuario presiona la tecla 0, entonces se pasa a la visualización en este modo. Si presiona cualquier otra tecla no se visualizarán las operaciones.

- *Detención en la muestra de estadísticas.* Esta opción nos permite detenernos en la visualización de las estadísticas que en el caso del programa aparecen en pantalla cada 50 generaciones/iteraciones del algoritmo.

  Si se pulsa el 0 el programa se detendrá en cada paso por las estadísticas, mientras que si se pulsa cualquier otra tecla la ejecución no se detendrá hasta llegar al final de la misma.

- *Selección de la densidad de los arcos.* La densidad de los arcos es una de las principales características iniciales de las redes. Existen tres posibles maneras de seleccionar la densidad de las redes:

  - *Selección aleatoria.* El programa decide aleatoriamente una de las cuatro densidades posibles (0.005, 0.01, 0.05 y 0.1) individualmente para cada una de los individuos que se crean para la población inicial.
  - *Selección no aleatoria del programa.* El programa crea, con cada valor de la densidad, unos cuantos individuos aleatoriamente a los que puntúa. La media de los resultados obtenidos por los individuos para cada una de las densidades se valora y se toma de partida aquella que tenga mejor puntuación media.
  - *Selección no aleatoria realizada por el usuario.* Es el usuario el que decide el valor de inicio de las densidades de los arcos. Como referencia, se puede decir que el valor 0.005 inserta muy pocos arcos en la red, y que este porcentaje aumenta a medida que aumenta la densidad. Los posibles valores que puede seleccionar el usuario son 0.005, 0.01, 0.05, 0.1 y 0.15. El hecho de que la densidad seleccionada sea mayor incrementa el tiempo que tarda el algoritmo en puntuar a los individuos.

- *Tipo de ejecución.* Esta opción decide el tipo de ejecución que se quiere siga el algoritmo. Las diferentes opciones que se pueden tener son:

  - *Básica.* En ella se utilizan los seis operadores implementados en el trabajo de Alicia Puerta [6], que son *Insertar Arco Dirigido(IAD), Insertar Arco No Dirigido(IAND), Borrar Arco Dirigido(BAD), Borrar Arco No Dirigido(BAND), Invertir Arco Dirigido(INV) y Construir V Estructura(VEST).*
  - *Básica + SHAREAD.* En ella, además de los seis operadores precedentes, se utiliza un nuevo operador, Compartir Arco Dirigido, *SHAREAD.* Éste toma un arco dirigido del mejor de los elementos de la población y lo inserta en otro de los individuos que la forman.
  - *Básica + SHAREAND.* Al igual que en la anterior, se parte de la configuración básica, esta vez utilizando otro operador nuevo, Compartir Arco No Dirigido *SHAREAND.* Éste toma un arco no dirigido del mejor de los elementos de la población y lo inserta en otro de los individuos que la forman.

- *Basica + SHAREAD + SHAREAND.* Igual que en el caso de las anteriores, se toman los seis operadores básicos y los dos nuevos como operadores para realizar las mutaciones en la población.

Si la introducción de los datos es incorrecta, entonces se elige por defecto la primera opción.

Una vez finalizadas las preguntas, aparecerán en pantalla los valores que se han seleccionado para cada uno de los valores requeridos en la ejecución. Una vez que se pulse la tecla *enter* comenzará la ejecución del algoritmo.

B.0.8 Valoración de resultados

Los resultados que la ejecución del algoritmo va produciendo se pueden ver de varias maneras.

La primera de ellas es mediante la *visualización de las operaciones*, aunque es un método poco efectivo por el tiempo que se pierde, no deja de ser ilustrativo si quieres ver como funciona el algoritmo a bajo nivel.

Otra manera es el *paso de las generaciones.* A cada paso de generación se hace una comparativa entre el mejor individuo de la población y la red objetivo. En esa comparación se puede ver como mejora el algoritmo los individuos de la población a lo largo del tiempo, aproximandose a la red objetivo.

También *la muestra de las estadísticas* permite conocer detalles de la evolución de la ejecución.

- Por la lectura de las puntuaciones de los individuos (Las puntuaciones a lo largo de la ejecución van aproximándose a la puntuación de la red objetivo dada en el resumen de datos introducidos).
- Por las estadísticas de las operaciones, los intentos exitosos y no exitosos de estas. Con estos datos se puede observar cuando una operación tiene más o menos éxito, cuales son las operaciones más exitosas, ...
- Por los valores de medias, mínimos, máximos y medianas de las puntuaciones.
- Por el número de arcos dirigidos, no dirigidos y densidades con respecto a la red objetivo que tiene cada uno de los elementos de la población. Esta última nos dice si la población se acerca a un número de arcos parecido al que tiene la red objetivo, si el número de arcos dirigidos y no dirigidos se adaptan al número de arcos de esos tipos que tiene la red objetivo, y sobre todo permite observar la posible diversidad dentro de los individuos de la población (Si existen varias combinaciones claramente diferenciables de arcos la diversidad es grande. Si por el contrario solo existe un patrón en la población, entonces la diversidad es mínima).

Todas esas que hemos dicho hasta ahora son valoraciones que podemos hacer a lo largo de la ejecución, pero hay otro grupo de valoraciones que solo se pueden hacer a posteriori.

La primera de ellas es el tiempo de ejecución que se ha necesitado para la misma. Cuando finaliza la ejecución aparece un mensaje en pantalla indicando el tiempo que se ha necesitado para la ejecución del programa, ya sea para la elección K2 como para el tipo evolutivo.

Otras de las valoraciones, aparte de un último mensaje estadístico, es la puntuación de las tres mejores redes y la comparativa entre el mejor individuo de la población y la red objetivo.

Una vez finalizada la ejecución del programa se almacenan una serie de datos en unos ficheros.

Los primeros tienen información estadística de la ejecución. Esta información se ha ido tomando cada diez iteraciones durante toda la ejecución del algoritmo y separada de la propia de cada fichero en todos ellos se ha ido almacenando la generación en la que se hacía la recogida de los valores, para hacerse una idea mejor de como iban variando los resultados. Estos son los ficheros y la información que contienen cada uno de ellos:

- *archivoGrafoEstadisticas.csv*, que guarda la información referente a las operaciones exitosas. En él se puede ver como varía en el tiempo el éxito de cada una de las operaciones.
- *archivoGrafoPuntuaciones.csv*, que guarda la información acerca de la mejor, la peor y la media de las puntuaciones.
- *archivoGrafoComparaciones.csv*, que guarda la información de los arcos dirigidos y no dirigidos coincidentes, los invertidos y los totales para los mejores individuos en cada uno de los momentos de recogida de la información a lo largo de la ejecución.

Todos estos ficheros son fácilmente visibles con la herramienta EXCEL o directamente con un editor de textos cualquiera, aunque la apertura de la misma con una hoja de cálculo daría la posibilidad de realizar gráficas y ver más claramente los resultados obtenidos y poder hacer un mejor análisis.

El otro fichero que se genera es el *mejorRed.dot*. En él se almacena la estructura de la red que se ha obtenido como mejor individuo de la población. Para visualizar esta red es necesario utilizar el visualizador *GraphViz* (ver carpeta de aplicaciones) y dentro de él la aplicación *dot*. Se introduce como fichero de entrada el archivo *mejorRed.dot* generado por nuestro programa y como fichero de salida el que almacenará la gráfica de la estructura. Una vez determinado el fichero de salida, hay que decidir la extensión de este (debe ser .dot) y realizar el Layout del fichero introducido. Una vez hecho esto se selecciona la visualización del layout realizado y en pantalla aparecerá el gráfico generado.

Bibliografía

[1] Murphy, K. (2003): *A Brief Introduction to Graphical Models and Bayesian Networks.* Disponible en la dirección www.ai.mit.edu/ murphyk/Bayes/bayes.html .

[2] Raftery, A.E. (1994): *Bayesian Model Selection in Social Research.*

[3] Maxwell Chickering, D. (2002): *Learning Equivalence Classes of Bayesian-Network Structures.* Journal of Machine Learning Research 2, 445-498.

[4] Muruzábal, J. Cotta, C. (2003): *A primer on the Evolution of Equivalence Classes of Bayesian-Network Structures.*

[5] Cooper, G.F. y Herskovits, E.A. (1992): *A Bayesian Method for the Induction of Probabilistic Networks from Data.* Machine Learning 9, 309-347.

[6] Puerta Torralbo, A. (2003): *Implementación y Evaluación de una Biblioteca de Redes Bayesianas en C.* Proyecto Fin de Carrera, Universidad de Málaga.

[7] Cotta, C. y Muruzábal, J. (2002): *Towards a More Efficient Evolutionary Induction of Bayesian Networks.*In Merelo, J., Adamidis, P., Beyer, H.G., Fernández-Villacañas, J.L., Schwefel, H.P., eds. (2002): Parallel Problem Solving From Nature VII. Volume 2439 of Lecture Notes in Computer Science. Springer-Verlag, Berlín, 730-739.

[8] Glover, F., Laguna, M. y Martí, R. (2002): *Scatter Search and Path Relinking: Advances and Applications.*

[9] Glover, F. y Laguna, M. (2000): *Fundamentals of Scatter Search and Path Relinking.* Control and Cybernetics 29, 653-684.

[10] North, S.C. (2002): *Drawing Graphs with NEATO.* Disponible en www.research.att.com/sw/tools/graphviz .

[11] Gansner, E., Koutsofios, E. y North, S.C. (2002): *Drawing Graphs with DOT.* Disponible en www.research.att.com/sw/tools/graphviz .

[12] Oetiker, T., Partl, H., Hyna, I. and Schlegl, E. (2004): *The Not So Short Introduction to LaTeX.*

[13] Larrañaga, P., Poza, M., Yurramendi, Y., Murga, R.H. y Kuijpers, C.M.H. (1996): *Structure Learning of Bayesian Networks by Genetic Algorithms: A performance Analisys of Control Parameters.* IEEE Transactions on Pattern Analysis and Machine Intelligence 10, 912-926.

[14] Beinlich, I., Suermondt, H., Chavez, R., Cooper, G. (1989): *The ALARM Monitoring System: A Case Study with Two Probabilistyc Inference Techniques for Belief Networks.* In Hunter, J., Cookson, J., Wyatt, J., eds. Proceeds of the Second European Conference on Artificial Intelligence and Medicine, Berlin, Springer-Verlag, 247-256

[15] Binder, J., Koller, D., Rusell, S. and Kanazawa, K. (1997):*Adaptative Probabilistic Networks With Hidden Variables.* Machine Learning 29, 213-244.

[16] Chickering, D.M., Geiger, D. and Heckermann, D. (1995): *Learning Bayesian Networks is NP-complete.* In Fisher, D. and Lenz, H.J., editors, Learning form Data: AI and Statistics V, Springer-Verlag 121-130.

[17] Wong, M.L., Lam, W. and Leung, K.S. (1999):*Using Evolutionary Programing and Minimum Description Length Principle for Data Mining of Bayesian Networks.* IEEE Transactions on Pattern Analisys and Machine Intelligence 21, 174-178.