Restricciones Para El Aprendizaje de Redes Bayesianas

Luis M. de Campos Ibañez
Fco. Javier García Castellano

Uncertainty Treatment in Artificial Intelligence Research Group
Department of Computer Science and Artificial Intelligence

Granada University (Spain)

Objetivo

- Permitir que los algoritmos de aprendizaje de redes bayesianas incluidos en Elvira pueda utilizar restricciones/conocimiento previo.

- Esto Implica:
  - Estamos trabajando con restricciones duras: La información dada por las restricciones prevalecerá.
  - La red obtenida debe satisfacer las restricciones.
  - Los elementos del espacio de búsqueda considerado, en gran medida deben satisfacer las restricciones.

Tipos de Restricciones

- **Restricciones de existencia de arcos y/o aristas:**

$$\exists x \rightarrow y \ o \ \exists x — y$$

- **Restricciones de ausencia de arcos y/o aristas:**

$$\nexists x \rightarrow y \ o \ \nexists x — y$$

- **Restricciones de orden parcial (relaciones de precedencia)**

$$x < y$$

Se asume la transitividad de esta relación, para que no sea necesario especificar todas las relación (si $x<y$ e $y<z$ entonces asumimos que también $x<z$)

Tipos de Restricciones

- En memoria:
  - <u>Existencia</u>: Como un elemento de la clase Graph. Nunca en una Bnet, puesto que en ésta última no puede haber aristas.
  - <u>Ausencia</u>: Como un elemento de la clase Graph. Nunca en una Bnet, puesto que en ésta última no puede haber aristas y ciclos dirigidos.
  - <u>Orden Parcial</u>: Como un elemento de la clase Graph (en este caso podría ser una Bnet). Donde $x < y$ se almacena como que existe el arco $x \rightarrow y$ en ese grafo.

Almacenamiento de Restricciones

- En **Disco** las restricciones se almacenarán como tres ficheros **.elv** independientes, utilizando el subtipo **Graph**

  - Un fichero .elv puede tener los subtipos Bnet, Network, IDiagram y Graph

- El subtipo Graph sólo se compone de:
  - Nombre
  - Tipo de Grafo (Directed, Undirected o Mixed)
  - Nodos (NodeList)
  - Enlaces (LinkList)

Almacenamiento de Restricciones

AutoConsistencia

- **<u>Existencia</u>: El grafo (mixto) resultantes de incluir todas las restricciones de este tipo no tendrá ciclos dirigidos.**

- **<u>Ausencia</u>: No se requiere ninguna.**

- **<u>Orden Parcial</u>: El grafo resultante de incluir todas las restricciones de este tipo Como un elemento de la clase Graph (en este caso podría ser una Bnet).**

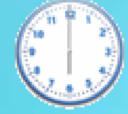
Donde $x < y$ se almacena como que existe el arco $x \rightarrow y$ en ese grafo.

Consistencia entre tipos

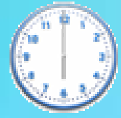
- **Denotemos por G_e, G_a y G_o los grafos correspondientes a las restricciones de existencia, ausencia y de orden respectivamente.**
- **Consistencia entre G_e y G_a:**
  - **Si $x \rightarrow y \in G_a$ entonces $x \rightarrow y \in G_a$ Un arco ausente no puede estar a la vez presente.**
  - **Si $x — y \in G_a$ entonces $x — y \notin G_e$ y $x \rightarrow y \notin G_e$ y $x \leftarrow y \notin G_e$. Una arista ausente no puede estar presente, ni ninguno de sus arcos derivados.**

Téngase en cuenta que los casos del tipo en que, por ejemplo, tenemos $x \rightarrow y \in G_a$ y $x — y \in G_e$ a efectos prácticos tendremos que $x \leftarrow y \in G_e$

Consistencia entre tipos



Consistencia entre G_o y G_a: No es necesario ningún tipo de comprobación.



Consistencia entre G_o y G_e: Estos dos tipos de restricciones interactúan debido a que la existencia de un arco $x \rightarrow y$ supone también una restricción de orden $x < y$.

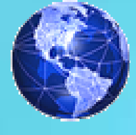
La comprobación de consistencia no se puede hacer componente a componente, sino de forma global.

Ejemplo: si $y<z$, $z<t$ y $t<x$, mientras que tenemos $x \rightarrow v$ y $v \rightarrow y$, tenemos una inconsistencia no detectable de forma local.

Hay que verificar que el grafo unión de G_o con G_e no tenga ciclos dirigidos. La unión de estos dos grafos debe hacerse con la convención de que $\{x — y\} \cup \{x \rightarrow y\} = \{x \rightarrow y\}$

Téngase en cuenta que los casos del tipo en que, por ejemplo, tenemos $x \rightarrow y \in G_o$ y $x — y \in G_e$ a efectos prácticos tendremos que $x \rightarrow y \in G_e$

Verificación de las restricciones

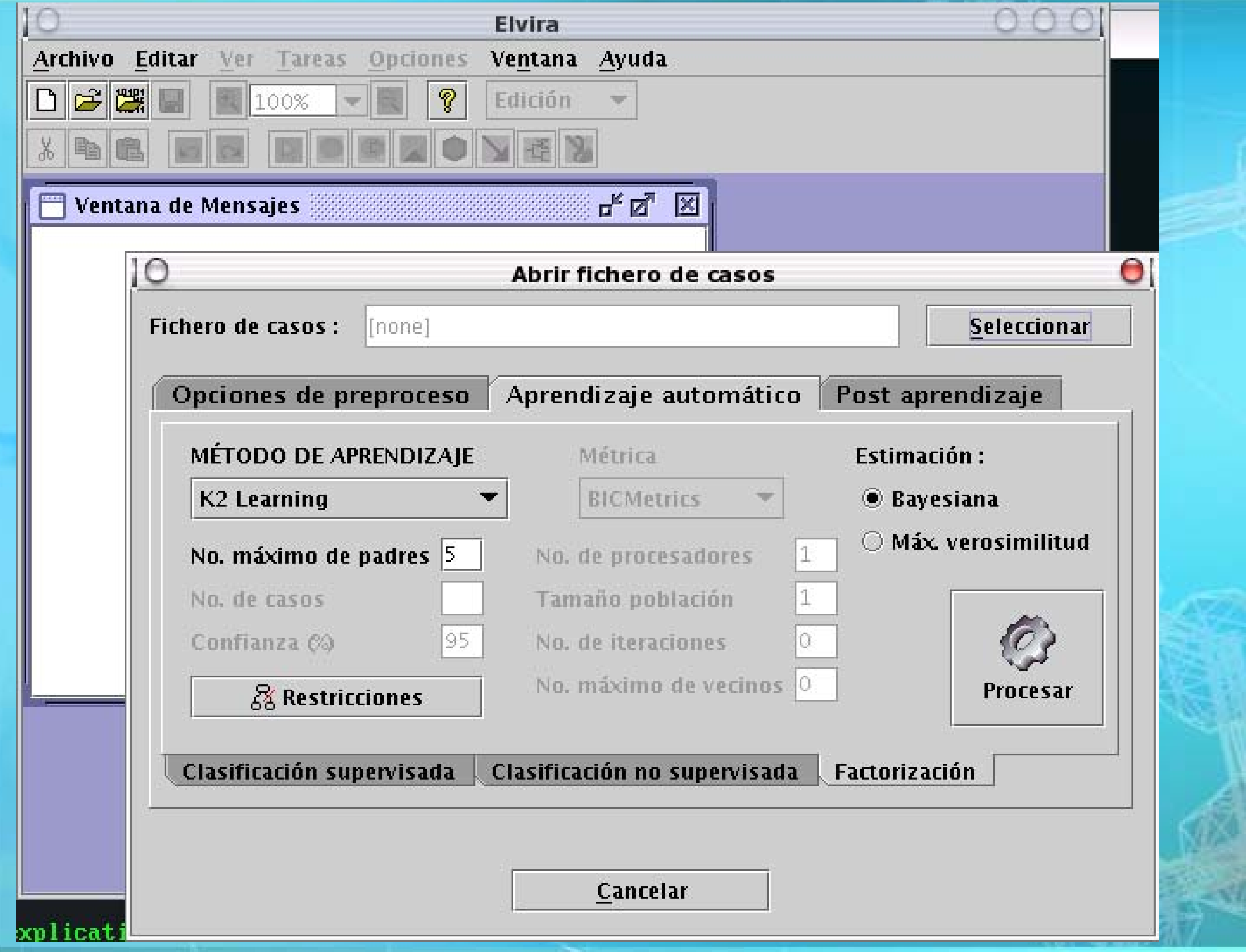


Se trata de estudiar cómo comprobar si una red que estemos considerando es realmente un elemento válido, es decir, cumple las restricciones impuestas

Suponemos que queremos comprobar si cumple o no las restricciones la red G:

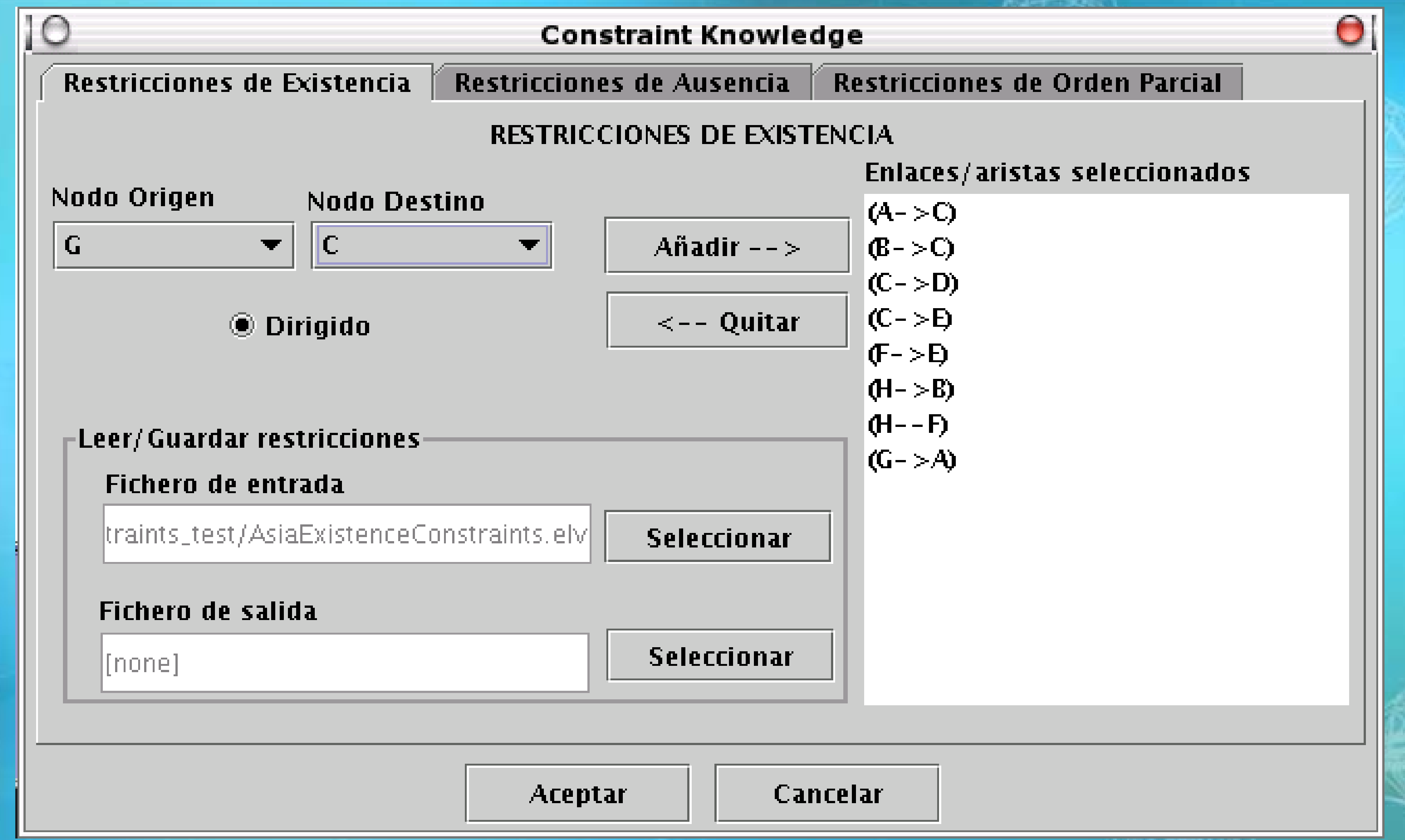
- Restricciones de existencia
  - Si $x \rightarrow y \in G_e$ entonces $x \rightarrow y \in G$
  - Si $x — y \in G_e$ entonces $x \rightarrow y \in G$ o $x \leftarrow y \in G$
- Restricciones de ausencia
  - Si $x \rightarrow y \in G_a$ entonces $x \rightarrow y \notin G$
  - Si $x — y \in G_a$ entonces $x \rightarrow y \notin G$ y $x \leftarrow y \notin G$
- Restricciones de orden: $G_o \cup G$ es un dag (no tiene ciclos dirigidos)

Introducción de restricciones GUI

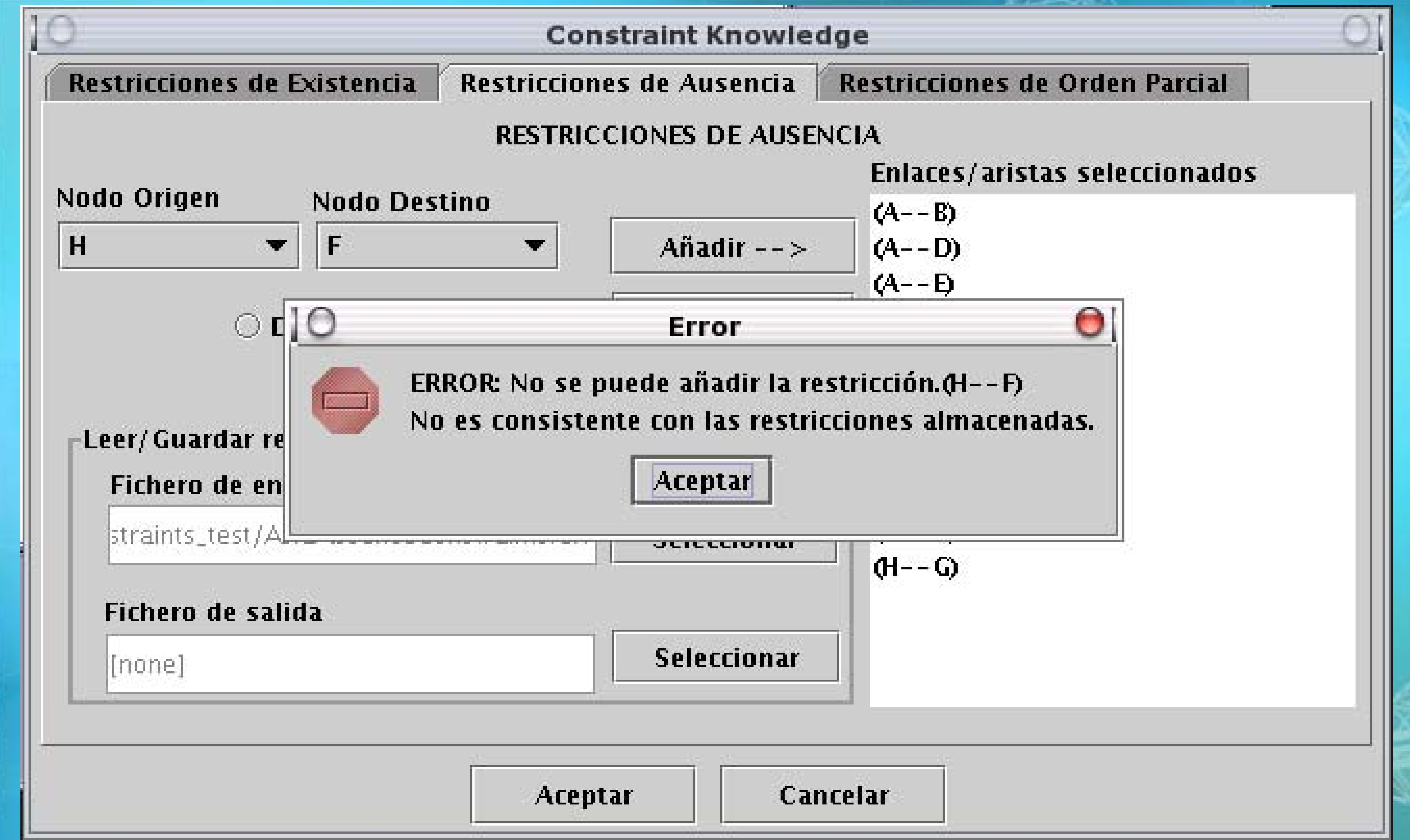


Introducción de Restricciones GUI

Introducción de restricciones GUI



Introducción de restricciones GUI



Introducción de restricciones GUI

Introducción de Restricciones GUI

Operadores para restriciones

Veremos las operaciones de inserción, borrado e inversión (útiles en búsqueda local)

Partimos de un dag G que ya cumple las restricciones, y el dag resultantes de aplicar un operador es G'.

Inserción: $G' = G \cup \{x \rightarrow y\}$ *con* $x \rightarrow y \notin G$

- Si $x \rightarrow y \notin G_a$ y $x — y \notin G_a$
- No existe un camino dirigido desde y hasta x en $G_o \cup G$ (esto es quivalente a decir que $G_o \cup G \cup \{x \rightarrow y\}$ *es un dag.*

Borrado: $G' = G \setminus \{x \rightarrow y\}$, *con* $x \rightarrow y \in G$

- Si $x \rightarrow y \notin G_e$ y $x — y \notin G_e$

Inversión: $G' = (G \setminus \{x \rightarrow y\}) \cup \{x \leftarrow y\}$, *con* $x \rightarrow y \in G$

- Si $x \rightarrow y \notin G_a$ y $x — y \notin G_a$
- Si $x \rightarrow y \notin G_e$ y $x — y \notin G_e$
- No existe camino dirigido de x hasta y en $G_o \cup (G \setminus \{x \rightarrow y\})$

Operadores para Restricciones

Inicialización

Inicialización: En lugar de partir de un grafo vacío, la inicialización consiste en empezar con el grafo *G*e . Pero si tiene aristas no es un dag.

Dirigir las aristas de forma aleatoria, pero asegurándose de que no se crean ciclos aleatorios ni que se vulneran restricciones de orden. Se repite el proceso hasta orientar todas las aristas.

Lo ideal es dirigir la dirección de las aristas según una métrica (por hacer).

Reparación

Sea G un dag que no cumple las restricciones. El proceso de convertirlo en un dag G' que si las cumpla es el siguiente: Sea G'=G

Borrar los arcos prohibidos:

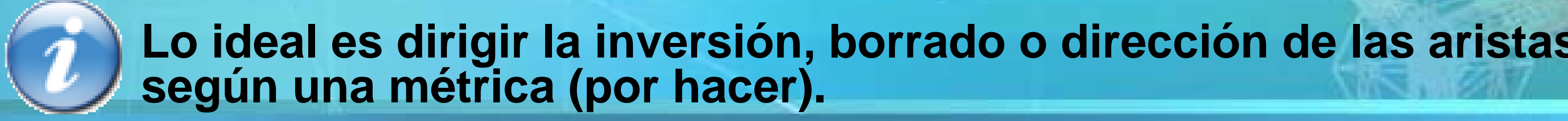
- **Para cada** $x \rightarrow y \in G_a$ **Si** $x \rightarrow y \in G'$ *entonces* $G' = G' \setminus \{x \rightarrow y\}$
- **Para cada** $x — y \in G_a$ **Si** $x \rightarrow y \in G'$ *o* $x \leftarrow y \in G'$ *entonces* $G' = G' \setminus \{x \rightarrow y\}$ *o* $G' = G' \setminus \{x \leftarrow y\}$

Respetar el orden:

- **Identificar arcos problemáticos**
- **Probar a invertir o borrar cada arco problemático de forma aleatoria**
- **Repetir hasta que** $G_o \cup G'$ **sea un dag**

Añadir arcos forzados:

- **Para cada** $x \rightarrow y \in G_e$ **Si** $x \rightarrow y \notin G'$ *entonces* $G' = G' \cup \{x \rightarrow y\}$
- ***Añadir las aristas de G, dándole un sentido de forma elatoria, pero minimizando los ciclos***
- ***Detectar y eliminar ciclos***



Lo ideal es dirigir la inversión, borrado o dirección de las aristas según una métrica (por hacer).

Reparación

Clases de Elvira para restriciones

Paquete *elvira.learning.constraints*

Clases:

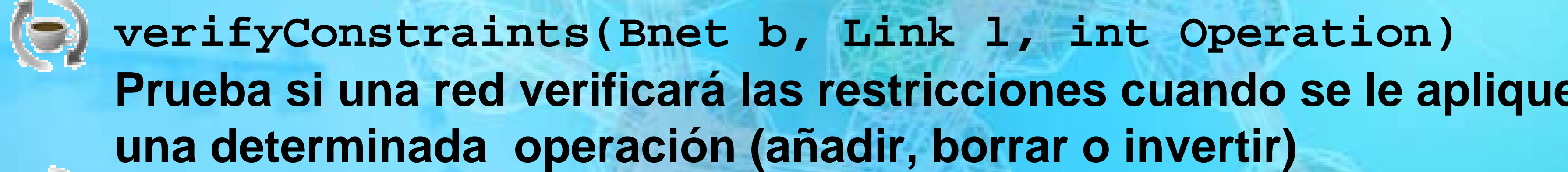
- **ConstraintsKnowledge.java: Restricciones**
- **Algoritmo PC con restricciones:** PCLearningCK.java
- **Algoritmo VNS con restricciones:** DVNSSTLearningCK.java , ThVNSSTCK.java , VNSSTProcesorCK.java
- **GUI:** ConstraintKnowledgeDialog.java, ConstraintKnowledgeDialog.java, ConstraintKnowledgeDialog.java

Métodos útiles para restriciones

Algunos métodos de la clase ConstraintKnowledge:

- `InitialBnet()` **Devuelve una bnet válida**

- **Operadores añadir, borrar, invertir verificando las restricciones:**
  - `createDirectedLink(Graph g, Link l),`
  - `removeLink(Graph g, Link l),`
  - `invertLink(Graph g, Link l)`



- `verifyConstraints(Bnet b, Link l, int Operation)` **Prueba si una red verificará las restricciones cuando se le aplique una determinada operación (añadir, borrar o invertir)**
- `test(Bnet b)` **Verifica que una Bnet cumpla las restricciones**
- `test(Bnet b, LinkList e, LinkList e, LinkList o)` **Verifica que una Bnet cumpla las restricciones y devuelve las restricciones que no cumple**
- `repair(Bnet b)` **Reparara una bnet para que cumpla las restricciones**