

# Introducción

El objetivo de este documento es describir el proceso de optimización del Asignador de Puestos, partiendo de la definición del Modelo de Asignación hasta la resolución final del problema y obtención de su solución.

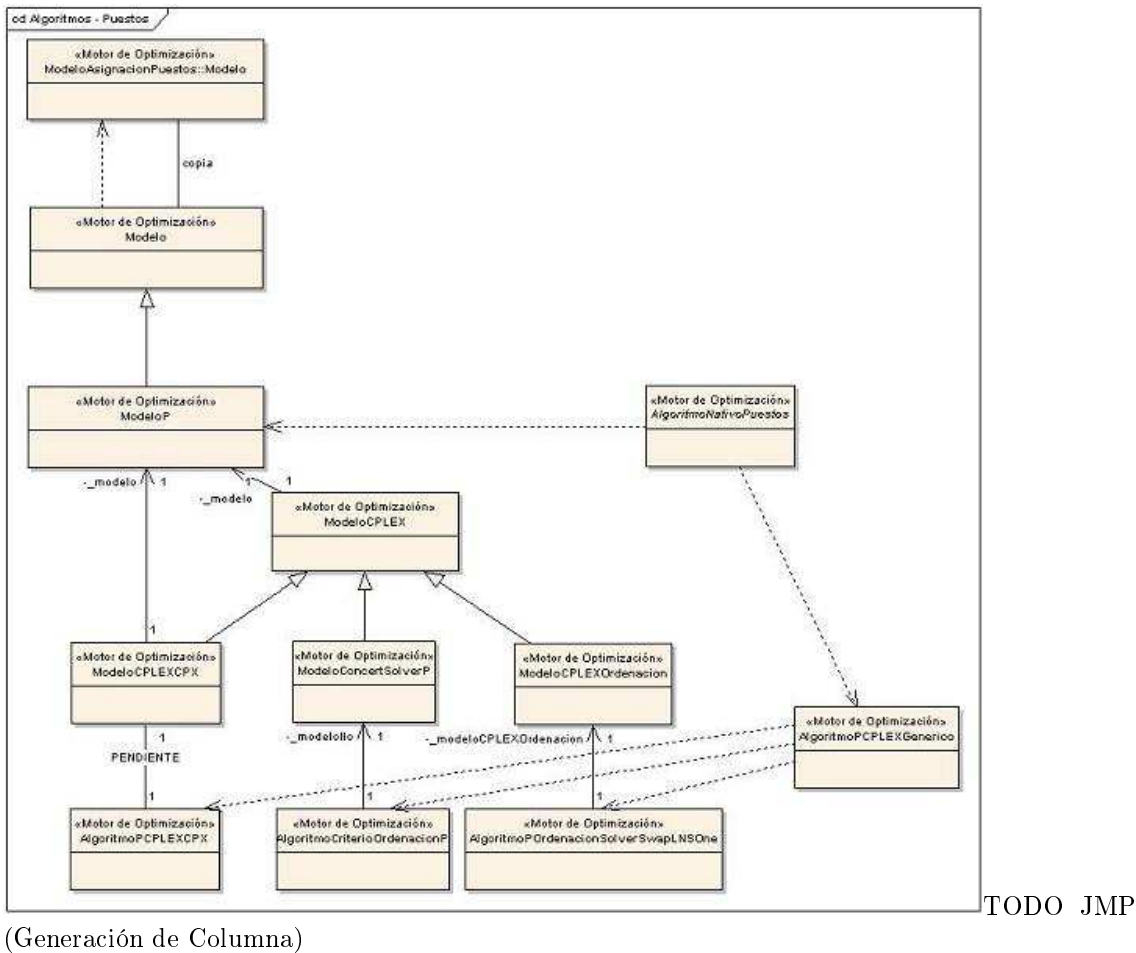
## Elementos presentes en el Proceso de Asignación

Durante el Proceso de Asignación, se van a utilizar los siguientes elementos:

- **Modelo de Asignación:** Es el punto de partida de la asignación, y representa en su totalidad el problema a resolver, desde un punto de vista de negocio (es decir, aparecen Agentes, Puestos, Compatibilidades entre Agentes y Puestos, etc.,)
- **Modelo de Optimización:** En función del Motor de Optimización que vaya a ser utilizado, será una formulación del problema contenido en el Modelo de Asignación en el formato definido por el Motor de Optimización utilizado. En el Modelo de Optimización se importarán, desde el Modelo de Asignación:
  - Agentes de Asignación
  - Puestos de Asignación
  - Restricciones
- **Algoritmo de Optimización:** Representa una técnica para resolver el problema definido por el Modelo de Asignación. Para ello, solicitará uno o varios de los Modelos de Optimización asociados a dicho Modelo de Asignación. En los Algoritmos de Optimización será donde residirán los Criterios de Evaluación del Modelo de Asignación. Con respecto a esto, tendremos dos tipos de Algoritmos:
  - **Algoritmos Genéricos:** Son aquellos que son capaces (quizás mediante el uso de otros algoritmos) de resolver el problema en su totalidad (es decir, todos los Criterios de Evaluación)
  - **Algoritmos Específicos:** Son aquellos que son capaces de resolver el problema pero teniendo en cuenta solamente uno de los Criterios de Evaluación presentes en el mismo. Estos Algoritmos serán utilizados por algunos de los Algoritmos Genéricos para resolver el problema.
- **Solución:** Es un conjunto de pares Agente – Puesto, representando el hecho de que el Puesto del par ha sido asignado al Agente. Diremos que una solución  $S$  es factible si cumple todas las restricciones presentes en el Modelo de Asignación. Diremos que  $(a, p) \in S$  si el Puesto  $p$  ha sido asignado al Agente  $a$  en la Solución  $S$ .

Las relaciones entre estas entidades, así como los nombres de las Clases de Diseño, vienen descritos en el siguiente Diagrama de Clases.





Se puede observar que, en este diagrama, tenemos:

- Un Modelo de Asignación, representado por las clases *ModeloAsignacionPuestos::Modelo* en Java y *siar::pp::ModeloP* en C++.
- Tres Modelos de Optimización:
  - *ModeloCPLEXCPX*: Representando un Modelo de Optimización basado en la tecnología ILOG CPLEX.
  - *ModeloConcertSolverP*: Representando un Modelo de Optimización basado en la tecnología ILOG Solver.
  - *ModeloCPLEXGC*: Representando un Modelo de Optimización basado en la tecnología ILOG CPLEX pero usando la técnica de Generacion de Columnas.
- Tres Algoritmos Específicos:
  - *AlgoritmoPCPLEXCPX*: Algoritmo basado en la tecnología ILOG CPLEX, y utilizado para la resolución de Criterios de Evaluación por coeficiente. Se puede observar que este Algoritmo hace uso del Modelo de Optimización *ModeloCPLEXCPX*.
  - *AlgoritmoCriterioOrdenacionP*: Algoritmo basado en la tecnología ILOG Solver y utilizado para la resolución de Criterios de Evaluación por ordenación. Se puede observar que este Algoritmo hace uso del Modelo de Optimización *ModeloConcertSolverP*.







El proceso de asignación comienza con un Modelo de Asignación, creado mediante la ejecución del proceso de extracción, y representando por completo todos los aspectos del problema a resolver.

A continuación se procederá a la resolución del modelo siguiendo estos pasos:

1. Selección del (los) algoritmos que serán utilizados para la solución del problema. Estos algoritmos serán genéricos, de modo que serán capaces de resolver el problema en su totalidad.
2. Ejecución de los algoritmos seleccionados. Eventualmente, alguno de estos algoritmos utilizará algoritmos específicos para resolver partes del problema
3. Obtención de la solución y devolución de la misma.

Como se puede observar, hay dos decisiones importantes a la hora de determinar el Algoritmo a utilizar:

- Elección del Algoritmo o secuencia de Algoritmos *Genéricos* que van a ser utilizados para resolver el Modelo de Asignación. Aquí tenemos dos posibles elecciones:
  - En el caso de que todos los Criterios de Evaluación sean por Coeficiente, encaja muy bien una definición Lineal del Modelo de Optimización. En este caso, utilizaremos un Algoritmo Genérico basado en Algoritmos específicos Lineales. En función del tamaño del Modelo de Asignación (en término de Puestos y Agentes), se utilizarán los siguientes Algoritmos:
    - *AlgoritmoPCPLEXGenerico*: Si el tamaño del Modelo de Asignación es lo suficientemente pequeño
    - *AlgoritmoPGCGenerico*: Si el tamaño del Modelo de Asignación es demasiado grande

La solución final obtenida será óptima, por lo que no hace falta utilizar ningún algoritmo de mejora de soluciones.

- En el caso de que exista algún Criterio de Evaluación por Ordenación, la definición Lineal del Modelo de Optimización se hace inestable, por lo que es necesario un Algoritmo Genérico que sea capaz de comunicar distintos Algoritmos Específicos. Este es el Algoritmo Genérico por Fixings (*AlgoritmoFixingsP*). Al ser posible que la solución provista por este Algoritmo no sea la óptima, a continuación se ejecutará un Algoritmo de Búsqueda Local con la intención de mejorar la solución obtenida (*AlgoritmoPOrdenacionSolverSwapLNSOne*).
- Elección del Algoritmo Específico dentro de cada uno de los Algoritmos Genéricos. Esta elección estará descrita en los apartados correspondientes a cada uno de estos Algoritmos Genéricos.

A continuación se procederá a describir cada uno de los componentes que forman parte del proceso de Asignación.

## Modelo de Asignación (siar::pp:ModeloAsignacionP)



El Modelo de Asignación es el punto de partida del Proceso de Optimización. Contendrá toda la información necesaria para definir cualquier Modelo de Optimización a resolver, esta información, descrita en los apartados posteriores, constará de:

- Agentes de Asignación
- Puestos de Asignación
- Restricciones
- Criterios de Evaluación

El objetivo del Asignador será el de resolver este Modelo, encontrando una solución que:

- Asigne, a cada Agente de Asignación, o bien uno de los Puestos de Asignación presentes en el Modelo de Asignación, o bien determine que el Agente queda `NO_ASIGNADO`.
- Cumpla todas las restricciones presentes en el Modelo de Asignación (descritas a continuación)
- Sea la mejor encontrada con respecto a los Criterios de Evaluación considerados de forma lexicográfica. En este sentido, si tenemos los Criterios de Evaluación  $C_1, \dots, C_n$ , y dos soluciones  $S_1, S_2$ , se considerará mejor la Solución que sea *mejor* con respecto al Criterio de Evaluación que esté situado *antes* en la lista. Si, por ejemplo, la Solución  $S_1$  fuese mejor con respecto a los Criterios  $C_2, \dots, C_n$ , pero peor con respecto al Criterio  $C_1$ , se consideraría que la Solución  $S_2$  es mejor que la Solución  $S_1$ .

### **Agentes de Asignación (siar::Agente)**

Son los agentes que, en función de las reglas aplicadas, son candidatos a ser asignados. Estos Agentes de Asignación podrían estar compuestos de dos Agentes físicos, de modo que la obtención de sus datos podría implicar a uno o a ambos agentes.

### **Puestos de Asignación (siar::pp:Puesto)**

Representan los Puestos que van a ser asignados a los Agentes de Asignación. Cada uno de estos Puestos de Asignación no tiene por qué corresponder, necesariamente, a un Puesto de la Base de Datos. Podría perfectamente corresponder a varios Puestos de la Base de Datos, cubiertos durante Franjas Horarias disjuntas, o bien a Puestos nuevos creados durante el proceso de extracción de los datos (normalmente reserva)

Algunos Puestos de Asignación vendrán marcados como Reserva Enumerada. Esto significará que estos Puestos de Asignación serán cubiertos cuantas veces se desee, pero, en el caso de que el Criterio de Evaluación sea con respecto a coeficiente, cada vez que se cubran el coste de la asignación aumentará.

### **Restricciones (siar::Restriccion)**



Las restricciones que se van a contemplar en el Modelo de Asignación son resultado de un filtrado realizado durante la ejecución de las reglas del proceso de extracción, así que no tienen por qué coincidir exactamente con elementos existentes en la Base de Datos. *Todas las restricciones existentes deberán cumplirse para que una solución sea válida.*

Por defecto, queda establecida la restricción de Unicidad de Agentes, que establece que a un Agente de Asignación sólo se le puede asignar un Puesto de Asignación. Esta restricción no es configurable, y es de obligado cumplimiento sea cual sea el Modelo de Asignación a resolver.

Los Tipos de Restricción contemplados son:

- Compatibilidades
- Cardinalidades (puras y mixtas)
- Restricciones por partición de Puestos
- Restricciones de Relación entre Agentes

### **Compatibilidades (siar::pp::CompatibilidadRestriccionP)**

Expresan el hecho de que un Agente de Asignación sea compatible con un Plan de Trabajo de Asignación. En el caso de que el Agente de Asignación  $A$  no sea compatible con el Puesto de Asignación  $P$ , entonces ninguna solución en la que el Agente de Asignación  $A$  sea asignado al Puesto de Asignación  $P$  será válida.

En lo sucesivo diremos que  $Compatible(A, P)$  será verdad si el Agente de Asignación  $A$  es compatible con el Puesto de Asignación  $P$  con respecto a las Restricciones de Compatibilidad.

### **Cardinalidades (siar::pp::CardinalidadP)**

Existen cardinalidades de dos tipos:

- *Cardinalidades puras*: Dado un Conjunto de Puestos de Asignación  $CONJ\_P$ , la restricción de Cardinalidad determina que los Puestos de Asignación que pertenezcan a dicho conjunto pueden aparecer asignados un máximo de  $MAX$  o  $MIN$  veces en una solución para que esta última sea válida.
- *Cardinalidades Mixtas*: Dado un Conjunto de Puestos de Asignación  $CONJ\_P$  y un Conjunto de Agentes  $CONJ\_A$ , esta restricción establecerá que los Puestos de Asignación pertenecientes a  $CONJ\_P$  no podrán ser cubiertos más de  $MAX$  o  $MIN$  veces a Agentes de Asignación pertenecientes a  $CONJ\_A$ .

### **Restricciones por Partición de Puestos**

Estas restricciones tienen dos Conjuntos, uno de Agentes (AG) y otro de Puestos (PU). Determinan que, si existe un Agente  $a$ , tal que  $a \notin AG$ , y  $a$  está asignado a un Puesto  $p \in PU$ , entonces tiene que haber otro Puesto  $p' \in PU$ , y otro Agente  $a' \in AG$  tal que  $a'$  esté asignado a  $p'$ .



En la actualidad estas restricciones se están usando para representar las Restricciones por Disponibilidad Parcial (Jornada Reducida). En este caso, tenemos un Puesto (de Base de Datos), que genera una lista de Puestos de Asignación  $\{p_1, \dots, p_n\}$ . Normalmente, uno de estos Puestos de Asignación cubrirá el Puesto por completo (vamos a decir que es  $p_1$ ), mientras que los demás lo cubrirán parcialmente, eventualmente mezclándolo con otros Puestos que se han partido.

La restricción que aparece en el análisis es que los Puestos deben de cubrirse en su totalidad por el mismo Agente, a no ser que una de sus partes haya sido cubierta por un Agente de Disponibilidad Parcial. Sea  $\{a_1, \dots, a_k\}$  la lista de Agentes de Disponibilidad Parcial. Generaríamos, para el Puesto anterior, una restricción de partición de Puestos RPP( $\{a_1, \dots, a_k\}, \{p_2, \dots, p_n\}$ ).

### Restricciones de Relación entre Agentes (siar::pp:RelacionAgentesP)

Las restricciones de Relación entre Agentes vienen definidas por una lista de Agentes de Asignación  $AG$ , una lista de Atributos Calculados  $ATR$ , y un conjunto t-uplas  $TP$  que asignan un valor a cada Agente de Asignación y Atributo Calculado contenido en los mismos. Vamos a determinar que  $Valor(t, ac, ag)$  devuelve el valor para la t-upla  $t$  del Atributo Calculado  $ac$  y el Agente  $ag$ . Representando esto en forma de tabla:

	$A_1$		...	$A_n$			
	$AC_1$	...	$AC_k$	$AC_1$	...	$AC_k$	
1	$V_{111}$	...	$V_{1k1}$	...	$V_{11n}$	...	$V_{1kn}$
...	...		...	...			
m	$V_{m11}$	...	$V_{mk1}$	...	$V_{m1n}$	...	$V_{1kn}$

Para describir el comportamiento de esta restricción necesitaremos nomenclatura adicional. Si tenemos una Restricción de Relación entre Agentes  $ra$ , entonces:

- Llamaremos  $ra.T(t, a)$  al conjunto de valores para la t-upla  $t$  y el Agente de Asignación  $a$  en la restricción de Relación entre Agentes  $ra$ , es decir, en nuestro caso,  $\{V_{t1a}, \dots, V_{tka}\}$
- Si  $T$  es uno de los conjuntos de valores descritos con anterioridad, llamamos  $P(T)$  al Conjunto de Puestos de Asignación tales que los valores de los Atributos Calculados de la Restricción de Relación entre agentes computados sobre los mismos correspondan a los de  $T$ , es decir,  $\{p \in P : AC_1.valor(p) = T_1, \dots, AC_k.valor(p) = T_k\}$
- Para cada Agente  $a \in ra.AG$ ,  $CV_a^{ra}$  como el conjunto de conjuntos de valores definidos por la Restricción de Relación entre Agentes para el Agente  $a$ , Es decir,  $CV_a^{ra} = \{ra.T(t, a) : t \in ra.TP\}$ .
- Decimos ahora que,  $\forall a_1, a_2 \in A, a_1 \neq a_2, \forall cv \in CV_{a_1}^{ra}, rel(cv, a_1, a_2) = \{ra.T(t, a_2) : t \in ra.TP \wedge ra.T(t, a_1) = cv\}$ , es decir,  $rel(cv, a_1, a_2)$  es el conjunto de conjuntos de valores que aparecen en la tabla para el Agente  $a_2$  tales que la los correspondientes valores en la misma t-upla para el Agente  $a_1$  son  $cv$ .

Dada esta nomenclatura, la Restricción de Relación entre Agentes establecería que, si un Agente  $A_s$  recibe un Puesto de Asignación  $p$  tal que  $p \in P(ra.T(t, A_s))$ , entonces para todo  $a \in \{A_1, \dots, A_n\} - A_s$ , el Puesto de Asignación asignado a  $a$  ( $p_a$ ) deberá cumplir que  $p_a \in \{P(rel(A_s, ra.T(t, A_s), a))\}$ .



Vamos a ejemplificar estas definiciones, con objeto de clarificar los conceptos. Si tenemos una Restricción de Relación entre Agentes que determina que los Agentes  $a_1$  y  $a_2$  tienen que hacer turnos *distintos*, esto se representaría del siguiente modo (suponiendo que tenemos tres Turnos  $M, T, N$ , y tres Puestos de Asignación  $P_M, P_T, P_S$ , con los Turnos definidos por sus subíndices):

	$a_1$	$a_2$
	TurnoPuesto	TurnoPuesto
1	M	T
2	M	N
3	T	M
4	T	N
5	N	M
6	N	T

Así, quedaría:

- Los Conjuntos de Valores ( $ra.T(t, a)$ ) serían, para ambos Agentes,  $\{M\}, \{T\}, \{N\}$
- Los Conjuntos de Puestos por cada conjunto de Valores ( $P(T)$ ):
  - $P(\{M\}) = \{P_M\}$
  - $P(\{T\}) = \{P_T\}$
  - $P(\{N\}) = \{P_N\}$
- Los Conjuntos de Conjuntos de Valores ( $CV_{a_1}^{ra}$ ) serían, para ambos Agentes,  $\{\{M\}, \{T\}, \{N\}\}$
- Ahora, los conjuntos relacionados ( $rel(cv, a_1, a_2)$ ) serían:
  - $rel(\{M\}, a_1, a_2) = \{\{T\}, \{N\}\}$
  - $rel(\{T\}, a_1, a_2) = \{\{M\}, \{N\}\}$
  - $rel(\{N\}, a_1, a_2) = \{\{M\}, \{T\}\}$
  - $rel(\{M\}, a_2, a_1) = \{\{T\}, \{N\}\}$
  - $rel(\{T\}, a_2, a_1) = \{\{M\}, \{N\}\}$
  - $rel(\{N\}, a_2, a_1) = \{\{M\}, \{T\}\}$

Por lo tanto, la Restricción de Relación entre Agentes determinaría que Si al Agente  $a_1$  le asignamos un Puesto de Asignación con el turno  $\{M\}$ , entonces al Agente  $a_2$  le tendremos que asignar un Puesto de Asignación con uno de estos Turnos  $\{\{T\}, \{N\}\}$ , y así sucesivamente examinando todas las relaciones generadas.

## Criterios de Evaluación (siar::pp::CriterioEvaluacionP)



El Modelo de Asignación soporta dos tipos de Criterio de Evaluación en el caso de Asignaciones de Puestos a Agentes:

- Criterios de Evaluación por Coeficientes: Asignan un coeficiente a cada Asociación de un Puesto de Asignación con un Agente de Asignación.
- Criterios de Evaluación por Ordenaciones: Se definen en función de listas de Preferencias y Prioridades

Ambos tipos quedan definidos a continuación.

### **Criterios de Evaluación por Coeficiente (siar::pp::CEAtributoCalculadoP)**

Estos Criterios de Evaluación proporcionan dos informaciones fundamentales:

- Por cada Agente de Asignación y cada Puesto de Asignación, un coeficiente representando el *coste* de asignar dicho Agente de Asignación al Puesto de Asignación. Diremos que si tenemos un Criterio de Evaluación por Coeficiente  $C^c$ , entonces  $C^c.coef(a, p)$  será el coeficiente asociado a asignar al Agente  $a$  el Puesto  $p$ .
- El *signo* de la evaluación, es decir, si queremos minimizar o maximizar los coeficientes obtenidos. El valor de  $C^c.signo$  será -1 si queremos minimizar, y 1 si queremos maximizar.

Así, si tuviéramos una variable  $x_{ap}$ , valiendo 1 si el Agente  $a$  está asignado al Puesto  $p$  y 0 en otro caso, un Criterio de Evaluación por Coeficiente establecerá la siguiente función objetivo:

$$\{\min | \max\} \sum_{a \in A, p \in P} C^c.coef(a, p) x_{ap}.$$

Por razones de eficiencia y ahorro de memoria, se han definido tres subtipos de Criterios de Evaluación por Coeficiente, en función de la información que necesiten para calcular el mismo:

- *Basados en Agentes (siar::pp::CEAtributoCalculadoAgente)*: Sólo necesitan el Agente de Asignación para obtener el coeficiente. Dado un Agente de Asignación  $a$ , todas las llamadas a  $C^c.coef(a, p)$  devolverán el mismo valor.
- *Basados en Puestos (siar::pp::CEAtributoCalculadoPuesto)*: Sólo necesitan el Puesto de Asignación para obtener el coeficiente. Dado un Puesto de Asignación  $p$ , todas las llamadas a  $C^c.coef(a, p)$  devolverán el mismo valor.
- *Basados en pares Agente-Puesto (siar::pp::CEAtributoCalculadoAgentePuesto)*: Necesitan conocer tanto el Agente de Asignación como el Puesto de Asignación para calcular el coeficiente.

Por otra parte, y afectando a este tipo de Criterios de Evaluación, aparece la *Reserva Enumerada* (RE). Esta Reserva Enumerada afecta en dos sentidos al Criterio de Evaluación:

- El Criterio de Evaluación debe determinar si tiene en cuenta o no la Reserva Enumerada para computar sus coeficientes
- En función de lo decidido en el punto anterior:
  - Si no se tiene en cuenta, para todo  $p \in RE$ , el *coste* de asignar  $p$  a un Agente  $a$  será siempre el mismo ( $= C^c.coef(a, p)$ ).



- Si se tiene en cuenta:
  - Se computará un *offset* del siguiente modo:  $\text{offset} = (\max \{C^c.\text{coef}(a, p) : a \in A, p \in \text{RE}\} - \min \{C^c.\text{coef}(a, p) : a \in A, p \in \text{RE}\} + 1) \cdot (-C^c.\text{signo})$ .
  - El *coste* asociado a las asignaciones de cada Puesto  $p \in \text{RE}$  vendrá representado por la siguiente función Lineal:

$$\left( \sum_{a \in A} C^c.\text{coef}(a, p) * \text{asignado}(a, p) \right) + \text{offset} * \text{numAsig}(p)$$

Siendo  $\text{asignado}(a, p)$  igual a 1 si  $p$  está asignado a  $a$  en la solución, y 0 en otro caso, y  $\text{numAsig}(p)$  igual al número de veces que  $p$  ha sido asignado a algún Agente.

### Criterios de Evaluación por Criterio de Ordenación (siar::pp::CECriterioOrdenacion)

Los Criterios de Evaluación por Criterio de Ordenación están formados por dos listas:

- Una lista de **Prioridades**, expresando la prioridad de los Agentes de Asignación para acceder a cada Puesto de Asignación. Define, por cada Puesto de Asignación, una Lista ordenada de Agentes de Asignación con respecto a su prioridad al acceder al Puesto.
- Una lista de **Preferencias**, expresando la preferencia de los Agentes de Asignación por cada Puesto de Asignación. Define, por cada Agente de Asignación, una lista ordenada de Puestos de Asignación ordenada por preferencia del Agente.

Con esto, se definirá que una asignación  $(A_1, P_1)$  es conflictiva si existe otra asignación  $(A_2, P_2)$  tal que  $A_1$  prefiere  $P_2$  a  $P_1$  (en función de la evaluación lexicográfica de CONJ\_PREF), y además  $A_1$  es más prioritario que  $A_2$  para cubrir el Puesto  $P_2$ . Diremos que un Conflicto( $(A_1, P_1), (A_2, P_2)$ ) será VERDAD si se produce esta situación. El objetivo de estos Criterios de Evaluación será el de minimizar el número de estos conflictos.

Como hemos dicho anteriormente, un Criterio de Evaluación por Ordenación viene definido, fundamentalmente, por dos Entidades: **Preferencias** y **Prioridades**. A continuación se describe cada una de ellas:

- **Preferencias**: Una preferencia es una clase que representa la *preferencia* de un Agente de Asignación sobre Puestos de Asignación con respecto a un aspecto concreto (Turno, Distancia a Localización de Entrada, etc.,).

En general una Preferencia proporciona un valor para cada Puesto, y una definición de la Preferencia del Agente en función de dichos valores. Para ello, debe implementar una característica de cada una de las siguientes perspectivas:

- *Información Necesaria para obtener el Valor*: Con respecto a esta perspectiva, tenemos las siguientes posibilidades:
  - *Simple*: Sólo necesita conocer el Puesto de Asignación para conocer el valor (e.g. Preferencia por Turno).
  - *Doble*: Necesita conocer tanto el Puesto de Asignación como el Agente de Asignación para conocer el Valor (e.g. Preferencia por cercanía a Localización de Entrada)



- *Condicionalidad*: Si la expresión de la preferencia depende de una condición evaluada sobre el Puesto de Asignación. Aquí tenemos tres posibilidades:
  - *No Condicionada*: La expresión de la preferencia no depende de ninguna condición evaluada sobre el Puesto (e.g. Preferencia por cercanía a Localización de Entrada)
  - *Condicionada Simple*: La expresión de la preferencia depende de una condición evaluada sobre el Puesto, y para calcular dicha condición sólo se necesita conocer el Puesto (ejemplo??)
  - *Condicionada Doble*: La expresión de la preferencia depende de una condición evaluada sobre el Puesto, y para calcular dicha condición se necesita conocer tanto el Puesto como el Agente de Asignación (ejemplo??)
- *Forma de Expresar la Preferencia*: Existen dos formas de expresar la preferencia:
  - *Enumerada*: Se expresa mediante una lista ordenada, para cada Agente, de los valores asociados a los Puestos de Asignación. Los Valores que vayan antes en dicha lista serán los preferidos por el Agente (e.g. Preferencia por Petición de Turnos).
  - *Inferida*: Se expresa mediante un orden sobre los valores obtenidos de los Puestos (e.g. Cercanía a Estación de Entrada). Este orden puede ser uno de estos tres:
    - ORDEN\_CRECIENTE: Se prefieren los valores en orden creciente
    - ORDEN DECRECIENTE: Se prefieren los valores en orden decreciente
    - ORDEN\_INDIFERENTE: Para el Agente no es importante esta preferencia
- **Prioridades**: Una prioridad es una clase que representa la *prioridad* de los Agentes de Asignación para acceder a un Puesto de Asignación con respecto a un aspecto concreto (Escalafón, Tipo de Agente, etc., ).

En general una Prioridad proporciona un valor para cada Agente, y una definición de la Prioridad de los Agentes con respecto al Puesto en función de dichos valores. Para ello, debe implementar una característica de cada una de las siguientes perspectivas:

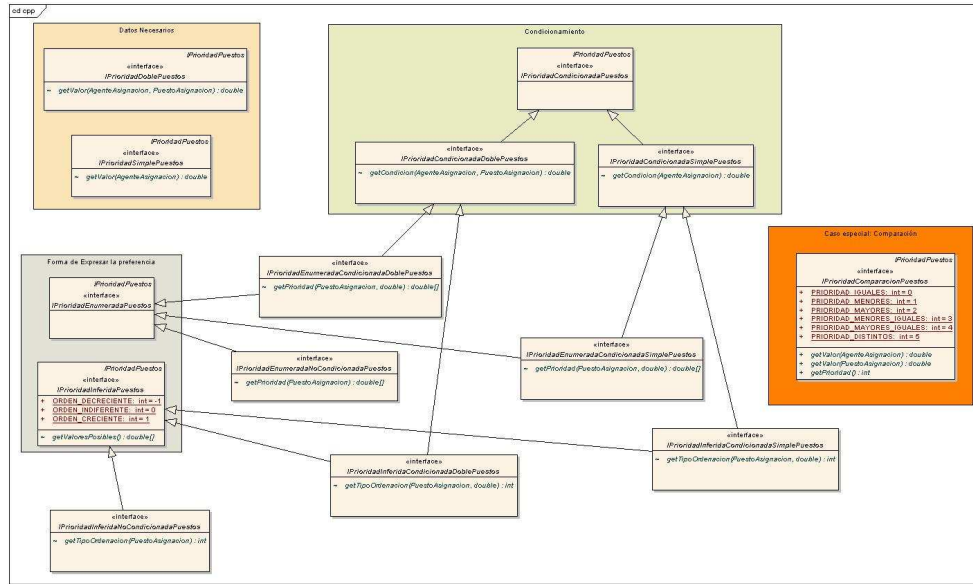
- *Información Necesaria para obtener el Valor*: Con respecto a esta perspectiva, tenemos las siguientes posibilidades:
  - *Simple*: Sólo necesita conocer el Agente de Asignación para conocer el valor (e.g. Prioridad por Escalafón).
  - *Doble*: Necesita conocer tanto el Puesto de Asignación como el Agente de Asignación para conocer el Valor (e.g. Prioridad por Condicionante de Vacante)
- *Condicionalidad*: Si la expresión de la preferencia depende de una condición evaluada sobre el Agente de Asignación. Aquí tenemos tres posibilidades:
  - *No Condicionada*: La expresión de la prioridad no depende de ninguna condición evaluada sobre el Agente (e.g. Prioridad por Escalafón)







- **Prioridades:**



## Modelos de Optimización

Los Modelos de Optimización son las representaciones en un formato comprensible por los Motores de Optimización disponibles del Modelo de Asignación a resolver. En la actualidad, usamos tres Motores de Optimización que necesitan de dos Modelos de Optimización distintos:

- Motor de Optimización CPLEX: Necesita de un Modelo de Optimización específico CPLEX (siar::pp::ModeloCPLEXCPX)
- Motor de Optimización Generación de Columnas: Necesita de un Modelo de Optimización específico para Generación de Columnas (usando ILOG CPLEX) (siar::pp::ModeloC-PLXGC)
- Motor de Optimización Solver: Necesita de un Modelo de Optimización específico Solver (siar::pp::ModeloConcertSolver).
- Motor de Optimización de Búsqueda Local: Necesita de un Modelo de Optimización específico Solver (siar::pp::ModeloConcertSolver).

A continuación se describen los distintos Modelos de Optimización necesarios, cada uno de ellos en la forma más apropiada en función de su composición.

## Descripción de los elementos contenidos en el Problema

En el Modelo de Optimización tendremos:

- A: Representando al conjunto de Agentes de Asignación del Modelo de Asignación



- $P$ : Representando al conjunto de Puestos de Asignación del Modelo de Asignación
- $C$ : Representando al conjunto de Restricciones de Compatibilidad contenidas en el Modelo de Asignación
- $CP$ : Representando al conjunto de Cardinalidades Puras contenidas en el Modelo de Asignación
- $CM$ : Representando al conjunto de Cardinalidades Mixtas contenidas en el Modelo de Asignación
- $RPP$ : Representando el conjunto de Restricciones de Partición de Puestos contenidas en el Modelo de Asignación
- $R$ : Representando al conjunto de Restricciones de Relación entre Agentes contenidas en el Modelo de Asignación

## Modelo CPLEX (siar::pp::ModeloCPLEXCPX)

Para la representación de este Modelo de Optimización utilizaremos su formulación matemática, ya que es la más apropiada para este tipo de Modelos.

Dividiremos la representación de este Modelo de Optimización en los siguientes apartados:

- Descripción de los elementos contenidos en el Problema
- Descripción de las Variables: Se definirán las variables a utilizar, así como su significado
- Descripción de las Restricciones: Por cada tipo de restricción, se mostrará su representación Matemática
- Descripción del Modelo completo: Se mostrará la representación del Modelo Matemático completo

### Descripción de las Variables del Problema

Se tendrá, para cada Agente de Asignación  $a$  y Puesto de Asignación  $p$  una variable  $x_{ap}$ . Esta variable cogerá el valor 1 si al Agente de Asignación  $a$  se le asigna el Puesto de Asignación  $p$ , y 0 en otro caso. Así,

$$x_{ap} \in \{0, 1\}, \forall a \in A, \forall p \in P$$

Además, para cada Agente, crearemos una variable más  $x'_a \in \{0, 1\}$  que valdrá 1 en el caso de que  $a$  no resulte asignado

### Restricciones: Unicidad

Cada Agente de Asignación sólo podrá ser asignado a un Puesto de Asignación

$$\forall a \in A, x'_a + \sum_{p \in P} x_{ap} = 1$$



### Restricciones: Compatibilidades

Las restricciones de compatibilidad se representarán del siguiente modo.

$$x_{ap} = 0, \text{ si } \exists \text{ comp} \in C, a \in A, p \in P \text{ t. } q. \text{ comp.Compatible}(a, p) = \text{NO}$$

### Restricciones: Cardinalidades Puras

Las restricciones de Cardinalidad Puras se representarán del siguiente modo:

$$\forall \text{cp} \in \text{CP}, \sum_{a \in A} \sum_{p \in \text{cp.CONJ}} x_{ap} \leq \text{cp.MaxCardinalidad}$$

$$\forall \text{cp} \in \text{CP}, \sum_{a \in A} \sum_{p \in \text{cp.CONJ}} x_{ap} \geq \text{cp.MinCardinalidad}$$

### Restricciones: Cardinalidades Mixtas

Las restricciones de Cardinalidad Mixtas se representarán del siguiente modo:

$$\forall \text{cm} \in \text{CP}, \sum_{a \in \text{cp.CONJA}} \sum_{p \in \text{cp.CONJP}} x_{ap} \leq \text{cp.MaxCardinalidad}$$

$$\forall \text{cm} \in \text{CP}, \sum_{a \in \text{cp.CONJA}} \sum_{p \in \text{cp.CONJP}} x_{ap} \geq \text{cp.MinCardinalidad}$$

### Restricciones: Restricciones de Partición de Puestos

Las restricciones de Particion de Puestos se representarán del siguiente modo:

$$\forall r \in \text{RPP}, \sum_{a \notin r.AG} \sum_{p \in r.PU} x_{ap} \leq M * \left( \sum_{a' \in r.AG} \sum_{p' \in r.PU} x_{a'p'} \right)$$

Esta linearización expresa claramente que si el lado izquierdo de la desigualdad es mayor que 0, entonces el lado derecho tiene que ser mayor que 0. Esto es, si existe un Agente  $a \notin r.AG$  asignado a un Puesto  $p \in r.PU$ , entonces tiene que existir algún Agente  $a' \in r.AG$  asignado a un Puesto  $p' \in r.PU$ .

### Restricciones: Restricción de Relación entre Agentes

Esta restricción se representa como una Linearización, por lo que, para representarla, será necesario añadir las siguientes restricciones lineales:

$$\forall \text{ra} \in \text{RA}, \forall a_1, a_2 \in \text{ra.A}, a_1 \neq a_2, \forall \text{cv} \in \text{CV}_{a_1}^{\text{ra}} \sum_{p \in P(\text{cv})} x_{a_1 p} \leq x'_{a_2} + \sum_{p \in P(\text{rel}(\text{cv}, a_1, a_2))} x_{a_2 p}$$



Notar que en cada una de estas restricciones estamos determinando que, si el lado izquierdo vale 1, entonces el lado derecho tiene que valer también 1. Si ahora examinamos el significado de cada uno de los dos lados, tendremos que:

- El lado izquierdo representa el número de veces que asignamos al Agente  $a_1$  a un Puesto  $p$  tal que  $p \in P(\text{cv})$
- El lado derecho representa, o bien la asignación del Agente  $a_2$  al Puesto NO\_ASIGNADO ( $x'_{a_2}$ ) o bien asignar al Agente a algún Puesto de asignación perteneciente a los relacionados con  $p$  y  $a_1$

Con esto representamos exactamente el significado de la restricción.

### Descripción del Modelo de Optimización Completo

Así, el Modelo de Optimización completo quedaría de la siguiente forma:

$$\begin{array}{ll}
 x'_a + \sum_{p \in P} x_{ap} = 1 & \forall a \in A \\
 x_{ap} = 0 & \forall a \in A, p \in P, \exists c \in Ct.q. c.Compatible(a, p) = \text{NO} \\
 \sum_{a \in A} (\sum_{p \in \text{cp.CONJ}} (x_{ap} \leq \text{cp.MC})) & \forall \text{cp} \in \text{CP} \\
 \sum_{a \in \text{cp.CONJA}} \sum_{p \in \text{cp.CONJP}} x_{ap} \leq \text{cp.MC} & \forall \text{cm} \in \text{CP} \\
 \sum_{p \in P(\text{cv})} x_{a_1 p} \leq x'_{a_2} + \sum_{p \in P(\text{rel}(\text{cv}, a_2))} x_{a_2 p} & \forall \text{ra} \in \text{RA}, \forall a_1, a_2 \in \text{ra}.A, a_1 \neq a_2, \forall \text{cv} \in \text{CV}_a^{\text{ra}} \\
 \sum_{a \notin r.AG} \sum_{p \in r.PU} x_{ap} \leq M * (\sum_{a' \in r.AG} \sum_{p' \in r.PU} x_{a' p'}) & \forall r \in \text{RPP}
 \end{array}$$

### Modelo CPLEX con Generación de Columnas (siar::pp::ModeloGCCPX)

La descripción matemática de este modelo es idéntica a la del modelo CPLEX ya que en realidad lo que varía es la forma de introducir las variables al modelo cpx de Ilog Cplex así como el uso que desde fuera se hace del modelo como consecuencia de la técnica de Generación de Columnas.

### Modelo Solver (siar::pp::ModeloConcertSolver)

El Modelo Solver está basado en la teoría de la Programación por Restricciones. En este sentido, cada una de las restricciones que aparecen en el Modelo de Asignación quedará representada como una restricción en el Modelo Solver.

Así, el procedimiento que vamos a seguir para describir este Modelo será el de:

1. Describir las variables existentes en el problema:
  - i. Descripción de las Variables fundamentales del problema (no de las auxiliares utilizadas en restricciones concretas)



- ii. Descripción de las Restricciones utilizadas para mantener la coherencia entre estas variables
2. Descripción de la Modelización de cada una de las restricciones presentes en el Modelo Solver.

Se puede observar, pues, que los dos tipos de objeto que vamos a definir en este apartado son Variables y Restricciones. Para cada una de ellas, se informará de las siguientes características:

- **Variables**
  - *Tipo de Variable* (Entera, Conjunto de Enteros, etc.), junto con la clase Solver utilizada para su representación
  - *Significado de la Variable*: Una breve descripción de lo que significa la variable de cara al proceso de optimización
  - *Dominio de la Variable*: El conjunto de valores posibles que inicialmente contiene la variable
- **Restricciones**:
  - *Significado de la restricción*: Una breve descripción del sentido de la restricción con respecto al Modelo de Optimización
  - *Invariante*: Descripción de la propiedad que la restricción pretende mantener a lo largo de todo el árbol de búsqueda
  - *Métodos de Conservación del Invariante*:
    - Conservación Inicial: Descripción del proceso mediante el cual la restricción consigue que el invariante se respete al principio de la asignación
    - Conservación Dinámica: En función de lo que vaya ocurriendo a lo largo de la búsqueda de soluciones, descripción del proceso que permite a la restricción conservar la propiedad definida por el invariante
  - *Propiedades especiales de nuestra implementación*: Describiendo, en el caso de que la restricción haya sido implementada por nosotros, las propiedades especiales que conseguimos.

Con objeto de obtener *siempre* soluciones, vamos a crear un Puesto Ficticio, llamado  $F$ , que representa la *no asignación* de un Agente.

En cualquier Modelo de Optimización escrito en ILOG Solver, no es tan importante la definición del Modelo como la calidad de las propagaciones contenidas en el mismo. Siendo esta calidad tan importante, dedicamos la siguiente sección a describir las técnicas de propagación utilizadas.

### Técnicas de Propagación utilizadas



Conviene aquí introducir una serie de conceptos teóricos que serán útiles para definir la calidad de una restricción:

**JLG TODO: CHEQUEAR SI EL SUPERINDICE  $V^R$  se usa**

**Definition 1.** Dada una variable  $V$ , se dice que  $D(V)$  es el conjunto de valores posibles que puede tomar dicha variable (es decir, su dominio). Cabe notar que este conjunto de valores puede variar dependiendo del momento de la búsqueda en que nos encontremos.

**Definition 2.** Una Restricción  $R(V_1^R, \dots, V_n^R)$  sobre un conjunto de variables  $\{V_1^R, \dots, V_n^R\}$  queda definida por un conjunto de  $t$ -uplas  $T^R = \{T_1^R, \dots, T_m^R\}$  tales que  $T_i^R = \{v_1^i, \dots, v_n^i\}$ . Es decir, una Restricción sobre un conjunto de variables se define como un conjunto de  $t$ -uplas de valores para esas variables, y define que los valores que reciban esas variables en una solución deben corresponder a una de esas Tuplas

**Definition 3.** Se dice que una Restricción  $R(V_1^R, \dots, V_n^R)$  es **Consistente** si existe al menos una  $t$ -upla  $T \in T^R$  tal que  $\forall i \in \{1, \dots, n\}, T_i \in D(V_i^R)$ . Es decir, una Restricción es consistente si existe una solución que cumpla dicha restricción

**Definition 4.** Se dice que una Restricción  $R(V_1^R, \dots, V_n^R)$  es **Arc-Consistent** si para todo  $i \in \{1, \dots, n\}$ , y para todo  $v_i^j \in D(V_i^R)$  existe una tupla  $T \in R$  tal que:

- $v_i^j = T_i$
- $T_k \in D(V_k^R)$ , para todo  $k \in \{1, \dots, n\}, k \neq i$

Es decir, que cada valor del dominio de cada variable de la restricción esté **justificado** por una  $t$ -upla.

**Definition 5.** Se dice que un Modelo de Optimización es **Arc-Consistent** si para toda variable  $V$  del Modelo, y para todo  $v_i \in D(V)$ , existe una  $t$ -upla  $T$ , tal que:

- $T \in R$ , para todas las restricciones  $R$  del Modelo.
- $T_i = v_i$
- $T_k \in D(V_k)$ , para todo  $k \neq i$

Es decir, que para todo  $v_i \in D(V)$ , existe una solución factible del Modelo de Optimización en la que  $V$  tiene el valor  $v_i$ .

Se puede observar que si definimos un Modelo de Optimización que sea *Arc-Consistent*, entonces, independientemente del orden de instanciación de variables o valores, nunca tendríamos un *fallo* (fail) durante el proceso de búsqueda.

Aunque es en general **falso** que un Modelo de Optimización cuyas restricciones sean todas *Arc-Consistent* sea él mismo *Arc-Consistent*, es cierto que resulta muy interesante, con respecto a la velocidad de búsqueda, así como a la calidad de las soluciones encontradas, que todas ellas lo sean, o al menos se aproximen a serlo.



Así, y siempre que desde el punto de vista de la eficiencia resulta *rentable*, o bien usamos Restricciones existentes en ILOG Solver que mantengan estas propiedades, o bien construimos las nuestras propias de forma que, o bien se mantengan estas propiedades, o bien estemos próximos a mantenerlas. Para ello, utilizamos una técnica basada en *soportes*.

**Definition 6.** Se dice que un valor  $v_i \in D(V)$  está **soportado** por la restricción  $R(V_1^R, \dots, V_n^R)$  si:

- O bien  $R$  no afecta a la variable  $V$ .
- O bien  $R$  afecta a la variable  $V$  (digamos que es la variable con índice  $j$ ) y existe una Tupla  $T \in R$  tal que:
  - $T_j = v_i$ , y
  - Para todo  $k \in \{1, \dots, n\}, k \neq i, T_k \in D(V_k^R)$

Es decir, existe una solución factible con respecto a la restricción en la que la variable  $V$  toma el valor  $v_i$ .

Una restricción que esté implementada usando la técnica de soportes, tiene que garantizar que TODOS los valores existentes en los dominios de las variables a las que afectan están *soportados*, o lo que es lo mismo, es *arc-consistent*.

Para cada una de las restricciones que usen esta técnica, lo especificaremos y describiremos la forma en que esta propiedad queda mantenida.

## Variables del Problema

En el Modelo Solver, tendremos dos tipos fundamentales de variables:

- $x_a, \forall a \in A$ : Representa el Puesto de Asignación asignado al Agente de Asignación  $a$ . Sus características son las siguientes:
  - *Tipo de Variable*: Es una variable de tipo Entera, representada con una instancia de la clase Solver **IloIntVar**.
  - *Significado de la Variable*: Esta variable representa el Puesto de Asignación que va a recibir el Agente  $a$ . Al ser de tipo Entera, en realidad contendrá el índice del Puesto de Asignación asignado al Agente.
  - *Dominio de la Variable*: La variable contendrá inicialmente en su dominio los siguientes valores:
    - Todos los índices de los Puestos de Asignación existentes en el Modelo de Asignación
    - Un índice especial, que representará el hecho de que el Agente no ha sido asignado (es decir, que ha sido asignado al Puesto  $F$ ).
- $w_p, \forall p \in P \cup F$ : Representa los Agentes de Asignación que han sido asignados a cada uno de los Puestos, incluido  $F$ :
  - *Tipo de Variable*: Es una variable de tipo Conjunto de Enteros, representada con una instancia de la clase Solver **IloIntSetVar**.



- *Significado de la Variable*: Representa los Agentes que están asignados al Puesto de Asignación  $p$ .
- *Dominio de la Variable*: En un principio, la variable contendrá
  - En su Conjunto de valores Posibles, todos los Agentes de Asignación existentes
  - En su Conjunto de valores requeridos, el conjunto vacío.
- $y_p, \forall p \in P \cup F$ : Representa el número de Agentes de Asignación asignados a cada uno de los Puestos, incluido  $F$ .
  - *Tipo de Variable*: Es una variable de tipo Entera, representada con una instancia de la clase Solver **IloIntVar**.
  - *Significado de la Variable*: Representa el número de Agentes que están asignados al Puesto de Asignación  $p$ . Equivale a la expresión  $\text{IloCard}(w_p)$ .
  - *Dominio de la Variable*: En un principio, la variable contendrá valores entre 0 y el número de Agentes de Asignación existentes.

Para mantener la coherencia entre estos dos conjuntos de variables, se usa una restricción preexistente en ILOG Solver **IloDistribute**, que garantiza, como mínimo, en todo momento, que:

- $\forall p \in P, y_{p.\min} = |\{a \in A: x_a = p\}|$
- $\forall p \in P, y_{p.\max} = |\{a \in A: p \in D(x_a)\}|$

La restricción **IloDistribute**, en su modo de propagación *IloExtended*, tiene una implementación *Arc-Consistent*.

### Restricciones: Compatibilidades (siar::pp::CompatibilidadI)

Las compatibilidades, en Solver, se representan mediante la restricción de Compatibilidad. Esta restricción tiene las siguientes características:

- *Significado de la Restricción*: Esta restricción mantendrá la propiedad de que no exista en el dominio de ninguna variable  $x_a$  un Puesto que sea incompatible con el Agente de Asignación  $a$ .
- *Invariante*: El invariante que se pretende mantener con esta restricción es el definido por la siguiente fórmula:

$$\text{Si } \exists c \in C, a \in A, p \in P, t.q. c.\text{Compatible}(a, p) = \text{NO}, \text{ entonces } p \notin D(x_a)$$

- *Métodos de Conservación del Invariante*: Este invariante sólo requiere de un proceso de conservación inicial, por lo que sólo se describirá este mismo.
  - *Conservación Inicial*: Para cada Agente de Asignación  $a$  se iterará sobre todos los valores contenidos en  $D(x_a)$ , y eliminará del mismo los valores de los Puestos de Asignación tales que  $\exists c \in C, a \in A, p \in P, t.q. c.\text{Compatible}(a, p) = \text{NO}$ .



- Conservación Dinámica: No procede.
- *Propiedades especiales de la restricción*: Se identifican dos propiedades interesantes en esta restricción:
  - Esta restricción se satisface durante la Conservación Inicial, por lo que es en sí *Arc-Consistent*, ya que cualquier valor de una variable que fuera susceptible de romper esta propiedad habría sido eliminado por este proceso
  - Si el Modelo de Optimización es en sí *Arc-Consistent*, añadir esta restricción no elimina esta propiedad del modelo, ya que la restricción en sí se limita a eliminar los valores imposibles desde el principio.

### Restricciones: Cardinalidades Puras

Las Cardinalidades Puras se representan mediante una restricción simple sobre las cardinalidades de los Puestos. Así, si  $cp \in CP$ , se añadirá la siguiente restricción (en este caso lineal)

$$\sum_{p \in cp.CONJP} y_p \leq cp.MaxCardinalidad$$

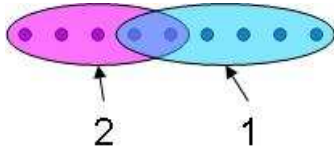
Siendo esta restricción Lineal, se añadirá a ILOG Solver, usando la propagación interna que éste tiene para este tipo de restricciones.

En cuanto a las *propiedades* de esta restricción, cabe definir dos casos:

- $|p.CONJP| = 1$ : En este caso, se mantiene la propiedad de *Arc-Consistency*, ya que sencillamente estamos acotando el dominio de una variable ( $y_p$ ), que a su vez está incluida en una restricción (*IlcDistribute*) que es *Arc-Consistent*. Añadiendo este tipo de restricciones no alteramos la propiedad de *Arc-Consistency*.
- $|p.CONJP| > 1$ : En este caso, la restricción deja de ser *Arc-Consistent*, con lo que elimina esta propiedad del Modelo de Optimización, si este la tuviera.

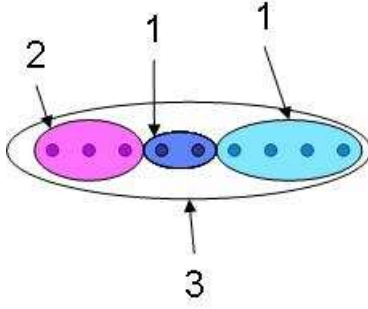
Para resolver los casos en los que las cardinalidades *rompen* la propiedad de *Arc-Consistency* del Modelo ( $|p.CONJP| > 1$ ), seguimos la siguiente técnica:

Supongamos que tenemos nueve Puestos, y dos restricciones de cardinalidad sobre ellos tal y como indica la figura:



Es decir, una Restricción de Cardinalidad sobre los primeros cinco, y con cardinalidad máxima 2, y otra sobre los últimos seis con cardinalidad máxima 1. Estas dos restricciones juntas rompen la propiedad *Arc-Consistency* del Modelo. Con unas deducciones sencillas, podemos ver que estas dos Restricciones de Cardinalidad inducen las siguientes:





Estas restricciones de cardinalidad inducida no son exactamente equivalentes a las originales, pero nos permiten plantear dos técnicas posibles de resolución del problema:

- Añadiendo nuevas restricciones *IlcDistribute*
- Usando ILOG Cplex para asegurar la propiedad de *Consistencia* del Modelo de Asignación

### Resolución mediante IlcDistribute

Podemos aumentar la calidad de la propagación mediante la introducción de nuevas restricciones de tipo *IlcDistribute*. Para ello, dividimos las cardinalidades inducidas en niveles, siguiendo el siguiente algoritmo:

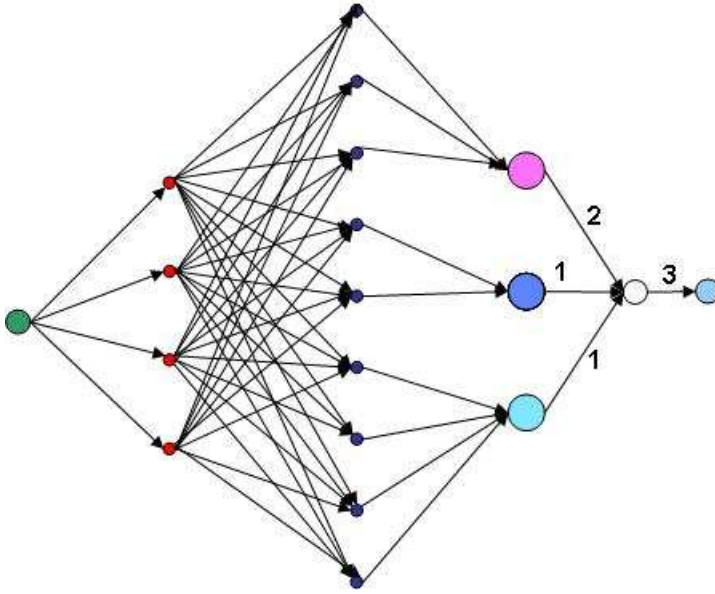
1. Llamamos  $C = \{C_1, \dots, C_n\}$  al conjunto de Cardinalidades inducidas.
2. Hacemos  $N := \emptyset$
3. Para cada  $C_i \in C$ , si  $C_i$  no contiene a ninguna otra Cardinalidad contenida en  $C$  ( $C \subseteq C'$  si todos los Puestos de  $C$  están en  $C'$ ), entonces hacemos  $N := N \cup \{C_i\}$ .
4. Si  $N \neq \emptyset$ , (es decir,  $N = \{C'_1, \dots, C'_k\}$ ), entonces, si  $\forall a \in A$ ,  $V_a$  es la variable restringida que representa el Puesto asignado a  $A$  :
  - i. Para cada Agente  $a \in A$ , creamos la variable restringida  $V'_a$ , con  $D(V'_a) = \{1, \dots, n\}$
  - ii. Definimos una restricción arc-consistent que mantiene que  $v'_a \in D(V'_a) \Leftrightarrow \exists v_a \in D(V_a)$  tal que  $v_a \in C_{v'_a} \text{PU}$ .
  - iii. Definimos una restricción *IlcDistribute* que defina que el número de veces que la variable  $V'_a$  de un Agente coge el valor  $v_a$  es menor o igual que  $C_{v'_a} \text{MAX}$ .
5. En otro caso ( $N = \emptyset$ ), acabamos.
6. Eliminamos las cardinalidades contenidas en  $N$  del conjunto de cardinalidades  $C$ .
7. Volvemos al paso 2.

De esta manera, definimos una serie de restricciones *Arc-Consistent* que mejoran la propagación que teníamos por defecto

### Resolución mediante el uso de ILOG CPLEX

En el caso de la resolución mediante el uso de ILOG CPLEX, la técnica consiste en plantear el siguiente grafo:





Este Grafo está dividido en varias columnas de nodos, cada una significando lo siguiente:

1. La primera (conteniendo un nodo verde), representa la entrada al grafo. Es un nodo auxiliar, y de él salen arcos con capacidad 1.
2. Los Nodos rojos corresponden a los Agentes existentes en el Modelo de Asignación. De ellos salen arcos con capacidad 1.
3. Los Nodos azules representan los Puestos. De ellos salen arcos con capacidad 1
4. Los Nodos del siguiente nivel representan las cardinalidades que están coloreadas en el dibujo anterior. De cada uno de ellos sale un arco con capacidad igual a la de la cardinalidad máxima.
5. El Nodo transparente del siguiente nivel representa la cardinalidad *grande* descrita anteriormente, con la capacidad igual a su cardinalidad máxima.
6. El Nodo final es un nodo auxiliar que representa el *sumidero* del grafo.

Resolviendo un problema de *Flujo Máximo* sobre este Grafo, tenemos una solución que satisface las restricciones de cardinalidad inducidas, teniendo que, si existe flujo entre el nodo que representa a un Agente y el que representa a un Puesto, entonces el Agente estará asignado al Puesto.

Un sencillo algoritmo de propagación que garantice que,  $\forall a \in A$ , si  $V_a$  es la variable restringida que representa:

- Si  $p \in P$ , y  $p \notin D(V_a)$ , entonces hacemos 0 la capacidad del arco que une  $a$  con  $p$
- Si  $p \in P$ , y  $D(V_a) = \{p\}$ , entonces hacemos 1 la capacidad del arco que une  $a$  con  $p$

Así que si el problema planteado por el grafo tiene solución entonces esta solución es una solución al problema con las cardinalidades inducidas. Si no la tiene, entonces el Problema con las cardinalidades inducidas no tiene solución, por lo que el problema en su totalidad tampoco la tiene.



### Restricciones: Cardinalidades Mixtas (siar::pp::CtRestriccionCardinalidadAgentePuestoPI)

Esta restricción se representará en Solver mediante el uso de Restricciones ya existentes en la librería. Así, si *varsAgentes* es una clase del tipo *IlcIntArray* conteniendo las variables de Puesto de cada uno de los agentes contenidos en *CM.CONJ\_A*, y *valsPuestos* conteniendo los índices e los Puestos contenidos en *CM.CONJ\_P*, tendremos:

```
IlcIndex i(solver);
solver.add(IlcCard(i, IlcMember(varsAgentes[i], valsPuestos)) <= CM.MAX);
```

Esta restricción representa exactamente lo descrito en la Restricción de Cardinalidad Mixta. Se contemplarán estas restricciones como restricciones de cardinalidad normales durante el proceso de inducción de cardinalidades descrito con anterioridad, ya que, en concreto una restricción de cardinalidad mixta *CM* induce una restricción de cardinalidad  $C'_{CM}$ , tal que  $C'_{CM}.CONJ\_A = CM.CONJ\_A$ , y  $C'_{CM}.MAX = CM.MAX$

### Restricciones: Restricción por Partición de Puestos

La Restricción por Partición de Puestos se mantiene mediante una restricción *custom*, es decir, implementada por nosotros. Esta restricción tendrá las siguientes propiedades:

- *Significado de la Restricción:* Esta restricción (rpp) tendrá que satisfacer que, si un Agente  $a \notin rpp.AG$ , y  $D\{V_a\} \subseteq rpp.PU$ , entonces existe  $a' \in rpp.AG$  tal que  $D\{V_{a'}\} \subseteq rpp.PU$
- *Invariante:* El invariante a conservar será el siguiente: Si no existe  $a' \in rpp.AG$  tal que  $D(V_{a'}) \cap rpp.PU \neq \emptyset$ , entonces tampoco existe  $a \notin rpp.AG$  tal que  $D(V_a) \cap rpp.PU \neq \emptyset$ . Es decir, si no existe ningún Agente en *rpp.AG* que pueda cubrir un puesto de *rpp.PU*, entonces ningún Agente puede cubrir un Puesto de *rpp.PU*.
- *Método de Conservación del Invariante*
  - Conservación Inicial: Si al comenzar la búsqueda no existe ningún Agente  $a' \in rpp.AG$  tal que  $D(V_{a'}) \cap rpp.PU \neq \emptyset$ , eliminamos del dominio de todos los Agentes los puestos pertenecientes a *rpp.PU*.
  - Conservación Dinámica: En cualquier momento en que tengamos que el número de Agentes de *rpp.AG* se vuelve 0, eliminamos del dominio de todos los Agentes los puestos pertenecientes a *rpp.PU*.
- *Propiedades especiales:* Esta restricción rompe la propiedad de *Arc-Consistency* del Modelo de Optimización, pero al ser una restricción de escasa aparición, y compleja de mejorar, nos limitamos a contar con la técnica de *Backtrack Inteligente*, más el preproceso ejecutado en cada Goal, contados con posterioridad en este documento, para solucionar los problemas que generara.

### Restricciones: Restricción de Relación entre Agentes (siar::pp::CtRestriccionRelacionAgentesPI)



La Restricción de Relación entre Agentes se representa en Solver mediante una restricción de tipo *custom*, es decir, una restricción implementada por nosotros. Así, procederemos a detallar las peculiaridades de la misma:

- *Significado de la Restricción*: Esta restricción tendrá que satisfacer que los Agentes de Asignación implicados en una Restricción de Relación entre Agentes sean asignados a Puestos de Asignación que respeten dicha restricción, como queda explicado en la definición del Modelo de Asignación.
- *Invariante*: El invariante a conservar será el siguiente: Si existe un Agente  $a \in \text{ra.AG}$ , y existe  $t \in \text{CV}_a^{\text{ra}}$  tal que para todo  $p \in D(V_a), p \in P(t)$ , entonces, para cada  $a' \in \text{ra.AG}, a' \neq a$ , en  $D(a')$  deben quedar sólo Puestos pertenecientes a  $\text{rel}(t, a, a')$ .
- *Método de Conservación del Invariante*
  - Conservación Inicial: Si al comenzar a propagar existe un Agente  $a \in \text{ra.AG}$ , y existe  $t \in \text{CV}_a^{\text{ra}}$  tal que para todo  $p \in D(V_a), p \in P(t)$ , entonces, para cada  $a' \in \text{ra.AG}, a' \neq a$ , eliminamos de  $D(a')$  todos los puestos que no pertenezcan a  $\text{rel}(t, a, a')$ .
  - Conservación Dinámica: En cualquier momento durante la búsqueda que se produzca que exista un Agente  $a \in \text{ra.AG}$ , y exista  $t \in \text{CV}_a^{\text{ra}}$  tal que para todo  $p \in D(V_a), p \in P(t)$ , entonces, para cada  $a' \in \text{ra.AG}, a' \neq a$ , eliminamos de  $D(a')$  todos los puestos que no pertenezcan a  $\text{rel}(t, a, a')$ .
- *Propiedades especiales*: Esta restricción rompe la propiedad de *Arc-Consistency* del Modelo de Optimización, pero al ser una restricción de escasa aparición, y compleja de mejorar, nos limitamos a contar con la técnica de *Backtrack Inteligente*, contada con posterioridad en este documento, para solucionar los problemas que generara.

## Algoritmos de Optimización

Los Algoritmos de Optimización son los encargados de resolver el Problema de Optimización producido por el Modelo de Asignación. Para ello, podrán hacer uso de Modelos de Optimización particulares para su técnica de resolución.

En particular, existen dos tipos de Algoritmos:

- *Algoritmos Genéricos*: Resuelven el Problema de Optimización en su totalidad, es decir, son capaces de tener en cuenta todos los Criterios de Evaluación existentes en el Modelo de Asignación
- *Algoritmos Específicos*: Resuelven un (o varios) tipos de Criterios de Evaluación, pero no todos ellos. Son normalmente usados por los Algoritmos Genéricos para resolver el Problema completo

### Algoritmos Genéricos (`siar::pp::AlgoritmoGenerico`)

En la actualidad tenemos tres Algoritmos Genéricos:

- *Algoritmo CPLEX Genérico* (`siar::pp::AlgoritmoPCPLEXGenerico`): Se usa cuando los Criterios de Evaluación existentes en el Modelo de Asignación son todos por Coeficiente, y utiliza una serie de algoritmos específicos de tipo CPLEX para la resolución global del problema



- *Algoritmo Fixings* (*siar::pp::AlgoritmoFixingsP*): Se usa cuando existe algún Criterio de Evaluación por Criterio de Ordenación. Su peculiaridad es que, a la hora de comunicar información entre algoritmos de distinto tipo, utiliza los *Fixings*, que son restricciones sencillas que son asimilables por todos los algoritmos.
- *Algoritmo Generación de Columnas Genérico* (*siar::pp::AlgoritmoPGCGenerico*): Se usa cuando los Criterios de Evaluación existentes en el Modelo de Asignación son todos por Coeficiente, y utiliza una serie de algoritmos específicos de tipo GENERACION DE COLUMNAS para la resolución global del problema.
- *Algoritmo LNS* (*siar::pp::AlgoritmoPOrdenacionSolverSwapLNSOne*): Utiliza técnicas de Búsqueda Local para mejorar la solución global. Al contrario que los otros dos, no utiliza algoritmos específicos, al ser autosuficiente para contemplar todos los Criterios de Evaluación.

En el caso de los tres primeros, la técnica usada es muy similar, y corresponde a este proceso:

1. Seleccionar el siguiente Criterio de Evaluación ( $ce := siguienteCriterio()$ )
2. Seleccionar el Algoritmo Específico a utilizar ( $alg := seleccionaAlgoritmoEspecifico(ce)$ ):
  - En el caso del Algoritmo CPLEX Genérico, siempre serán algoritmos de tipo CPLEX
  - En el caso del Algoritmo Fixings, podrá ser cualquier tipo de algoritmo específico
  - En el caso del Algoritmo Generación de Columnas Genérico, siempre serán algoritmos de tipo GENERACION DE COLUMNAS.
3. Con el algoritmo, resolver el Criterio de Evaluación seleccionado
4. Añadir restricciones al Modelo de Optimización que establezcan que el siguiente algoritmo *no puede empeorar la solución encontrada con respecto a éste Criterio de Evaluación*:
  - En el caso del Algoritmo CPLEX Genérico o el de Generación de Columnas, serán restricciones lineales
  - En el caso del Algoritmo Fixings, serán Fixings.
5. Volver al paso 1, hasta que se acaben los Criterios de Evaluación.

### **Algoritmo CPLEX Genérico (*siar::pp::AlgoritmoPCPLEXGenerico*)**

Este algoritmo se utiliza cuando todos los Criterios de Evaluación son por coeficiente. Hay fundamentalmente dos razones para utilizar este Algoritmo en este caso

- Un Criterio de Evaluación por Coeficiente produce automáticamente una función objetivo *lineal*, por lo que Algoritmos de tipo CPLEX son muy eficientes
- El conocimiento de que todos los Algoritmos utilizados son de Tipo CPLEX permite definir sistemas de comunicación más efectivos, ya que se puede directamente introducir una restricción lineal en el siguiente Algoritmo para respetar el *coste* obtenido con el anterior.



Así, este algoritmo iterará sobre los Criterios de Evaluación existentes *por orden de prioridad* y:

1. Resolverá el Criterio de Evaluación mediante el uso de un Algoritmo Específico CPLEX
2. Comunicará al siguiente Algoritmo Específico, mediante una restricción lineal, la imposibilidad de empeorar el objetivo obtenido. Así, si la función objetivo del anterior Criterio de Evaluación era  $\text{obj}$ , y el valor obtenido en la solución encontrada era  $\text{val}$ , se añadirá una restricción de la forma  $\text{obj} \leq \text{val}$  (si el criterio es de minimización) u  $\text{obj} \geq \text{val}$  (si el criterio es de maximización) al siguiente Algoritmo específico CPLEX. Esta restricción es *lineal*, por lo que encaja a la perfección dentro de las que maneja CPLEX.

### Algoritmo Generación de Columnas Genérico (siar::pp::AlgoritmoPGCGenerico)

Este algoritmo se utiliza cuando todos los Criterios de Evaluación son por coeficiente y las dimensiones del problema son muy grandes. La razón para utilizar este algoritmo se debe exclusivamente a que el número de puestos generados sea muy grande y que el Algoritmo CPLEX Genérico anterior sea incapaz de resolverlo (por tiempo o por memoria).

Así, este algoritmo iterará sobre los Criterios de Evaluación existentes *por orden de prioridad* y:

1. Resolverá el Criterio de Evaluación mediante el uso de un Algoritmo Específico Generación de Columnas.
2. Comunicará al siguiente Algoritmo Específico, mediante una restricción lineal, la imposibilidad de empeorar el objetivo obtenido de la misma forma que se hace en el Algoritmo CPLEX genérico.

### Algoritmo Fixings (siar::pp::AlgoritmoFixingsP)

El Algoritmo Fixings se usa cuando tenemos de todos los tipos de Criterios de Evaluación. Por lo tanto, usará también varios tipos de Algoritmos Específicos. Esto crea un *problema de comunicación entre Algoritmos* que no existía en el caso anterior, ya que todos eran del mismo tipo.

Para solucionar este problema, aparece el concepto de **Fixing**. Un Fixing es una restricción con las siguientes propiedades:

- *Es Simple*: Es decir, **todos** los Algoritmos Específicos son capaces de asimilarla sin bajar su rendimiento.
- *Determina que los siguientes algoritmos no empeoren la solución presente*, pero posiblemente sea aún más restrictivo. En este sentido, los Fixings añadidos podrán eliminar soluciones factibles, aunque, por contra, permitirán encontrar *buenas* soluciones de forma muy eficiente.

Existen en la actualidad cuatro tipos de Fixing:

- FixingCubrirAgentePuestos:
  - Contiene un Agente de Asignación y un conjunto de Puestos de Asignación.



- Determina que al Agente de Asignación sólo le podrá ser asignado uno de los Puestos de Asignación contenidos en el conjunto
- FixingNoAsignarAgente:
  - Contiene un Conjunto de Agentes de Asignación.
  - Determina que *ningún* Agente de Asignación del conjunto será asignado
- FixingCubrirPuesto:
  - Contiene un Conjunto de Puestos de Asignación
  - Determina que *todos* los Puestos de Asignación contenidos en el conjunto deben de ser cubiertos
- FixingNoCubrirPuesto
  - Contiene un Conjunto de Puestos de Asignación
  - Determina que *ninguno* de los Puestos de Asignación contenidos en el Conjunto debe de ser cubierto
- FixingCardinalidadPuestos:
  - Contiene un Conjunto de Puestos de Asignación y una Cardinalidad Máxima
  - Determina que el número máximo de veces que se deben cubrir los Puestos de Asignación contenidos en el conjunto es la Cardinalidad Máxima.
- FixingCardinalidadPuestosAgentes:
  - Contiene un Conjunto de Agentes de Asignación y una Cardinalidad Máxima
  - Determina que el número máximo de veces que se deben asignar los Agentes de Asignación contenidos en el conjunto es la Cardinalidad Máxima.

Así, este algoritmo iterará sobre los Criterios de Evaluación existentes *por orden de prioridad* y:

1. En función del Criterio de Evaluación, seleccionará el Algoritmo a utilizar para su resolución
2. Comunicará al siguiente Algoritmo Específico, mediante un Fixing, la imposibilidad de empeorar el objetivo obtenido.

Para todo ello, va a hacer uso de las siguientes clases:

- *Selector Algoritmos* (*siar::pp::SelectorAlgoritmo*): Es una clase que, en función de un Criterio de Evaluación, determina el Algoritmo que va a ser utilizado. En la actualidad el criterio para seleccionar el algoritmo será el siguiente:
  - Si el Criterio de Evaluación es por Coeficiente, se selecciona el Algoritmo Específico CPLEX o el Algoritmo Específico de Generación de Columnas, dependiendo del tamaño del Modelo de Asignación.



- Si el Criterio de Evaluación es por Ordenación, se selecciona el Algoritmo Específico Solver por Ordenación
- *Generador Fixings (siar::pp::GeneradorFixings)*: Es la clase encargada de generar los fixings en función de la solución encontrada y del Criterio de Evaluación para el que se encontró dicha Solución. En la actualidad, se generan los siguientes Fixings:
  - *Si el Criterio de Evaluación es por Coeficiente sobre Agentes*: Si  $a \in A$  es el Agente con menor coeficiente que ha sido asignado en la solución:
    - Se añadirá un Fixing de tipo FixingNoAsignarAgente para todos los agentes con coeficiente *menor* que el de  $a$  para este criterio
    - Se añadirá un Fixing de tipo FixingCardinalidad para todos los Agentes con coeficiente *igual* al de  $a$ , y Cardinalidad Máxima igual al número de veces que estos agentes han sido asignados en la solución.
  - *Si el Criterio de Evaluación es por Coeficiente sobre Puestos*: Si  $p \in P$  es el Puesto con menor coeficiente que ha sido asignado en la solución:
    - Se añadirá un Fixing de tipo FixingNoCubrirPuesto para todos los puestos con coeficiente *menor* que el de  $p$  para este criterio
    - Se añadirá un Fixing de tipo FixingCardinalidadAgentes para todos los Agentes con coeficiente *igual* al de  $a$ , y Cardinalidad Máxima igual al número de veces que estos agentes han sido asignados en la solución.
  - *Si el Criterio de Evaluación es por Coeficiente sobre Agentes - Puestos*:
  - *Si el Criterio de Evaluación es por Ordenación*: Para cada  $a \in A$ , si  $p_a \in P$  es el Puesto de Asignación asignado a  $a$ , entonces se creará un fixing de tipo FixingCubrirAgentePuesto que contenga todos los Puestos de Asignación que  $a$  prefiere *igual* que  $p_a$ , es decir, todos los puestos  $p \in P$  que sean *iguales* a  $p_a$  con respecto a las preferencias de  $a$ .

#### **Algoritmo LNS (Búsqueda Local) (siar::pp::AlgoritmoPOrdenacionSolver-SwapLNSOne)**

Este algoritmo es un algoritmo de Búsqueda Local. Un algoritmo de Búsqueda Local se puede definir en función de dos dimensiones:

- Los Movimientos que permiten modificar la solución en curso para llegar a otra solución
- Los mecanismos de aceptación de soluciones nuevas.

Se podría definir el proceso de ejecución de un Algoritmo de Búsqueda Local mediante los siguientes pasos:

1. Dada una solución, intentar explorar las soluciones *vecinas*, es decir, las soluciones a las que es posible llegar aplicando los Movimientos sobre la solución actual.
2. Para cada una de las soluciones vecinas, aceptarla o no en función de los Mecanismos de Aceptación de Nuevas Soluciones



en este sentido, consiste en, a partir de una solución inicial, intentar mejorar la misma realizando pequeños cambios. Estos cambios se llaman *vecindario*.

En nuestro algoritmo de Búsqueda Local, existen dos tipos de *vecindario*:

- Vecindario ***Swap*** (siar::pp::NHoodIntercambiarAsignacionesPI): Dada una solución, produce otra solución intercambiando las asignaciones de dos Agentes
- Vecindario ***LNS*** (Large Neighborhood Search): Dada una solución, fija la asignación de un conjunto de Agentes a la que tienen en la solución, y libera a los demás Agentes de modo que se realice una búsqueda exhaustiva en los Agentes *libres* para intentar mejorar la solución.  
**TODO JLG SELECCIÓN DE AGENTES A LIBERAR**

En cuanto a los Mecanismos de Aceptación de Nuevas Soluciones, usamos en la actualidad dos:

- ***IloImprove***: Este mecanismo sólo acepta soluciones que mejoren la actual. El problema de este tipo de mecanismos suele ser que existe la posibilidad de que pare en mínimos locales, es decir, en puntos donde los *vecindarios* próximos no mejoren la solución, pero que haya vecindarios más lejanos que sí la mejorarían
- ***IloTabuSearch***: Este mecanismo acepta soluciones peores en cuanto al objetivo, con objeto de evitar mínimos locales. Para dirigir la búsqueda y no empeorar continuamente las soluciones, mantiene una lista de los  $N$  últimos movimientos realizados, de modo que se prohíben soluciones a las que se llegue, desde la actual, deshaciendo dichos movimientos  
**(JLG madurar)**

El problema de este tipo de Algoritmos es que suelen ser de convergencia lenta, de modo que es muy interesante haber encontrado una buena solución a partir de la cual ejecutar este algoritmo. Esta solución se obtiene con alguno de los otros Algoritmos Genéricos, por lo que este algoritmo se ejecutará al final.

## Algoritmos Específicos

Los algoritmos específicos son Algoritmos especializados en la resolución del Modelo de Asignación para uno o varios Criterios de Evaluación, pero no para todos ellos. Esto nos permite resolver dichos criterios de forma muy eficiente, pero nos hace perder visión sobre el Modelo de Asignación visto de forma Global, por lo que estos algoritmos siempre son utilizados desde otros Algoritmos Genéricos que sí tienen esta visión.

Cada Algoritmo específico utiliza un Modelo de Optimización, pudiendo ser dicho Modelo distinto para cada algoritmo. En la descripción de cada uno de los Algoritmos Específicos existentes se determinará el Modelo de Optimización que utilizan.

En la actualidad tenemos tres Algoritmos Específicos:

- Algoritmo CPLEX por Coeficiente (siar::pp::AlgoritmoPCPLEXCPX): Este Algoritmo está basado en la tecnología ILOG CPLEX, y resuelve los Criterios de Evaluación por Coeficiente
- Algoritmo Solver por Ordenación (siar::pp::AlgoritmoCriterioOrdenacion): Este Algoritmo está basado en la tecnología ILOG Solver, y resuelve los Criterios de Evaluación por Ordenación.



- Algoritmo Generación de Columnas por Coeficiente (siar::pp::AlgoritmoPGCCPX): Este Algoritmo está basado en la tecnología ILOG CPLEX, y resuelve los Criterios de Evaluación por Coeficiente utilizando la técnica de Generación de Columnas.

A continuación se describen en más detalle ambos algoritmos.

### Algoritmo CPLEX por Coeficiente (siar::pp::AlgoritmoPCPLEXCPX)

Este Algoritmo, mediante el uso de un Modelo de Optimización basado en ILOG CPLEX, establecerá una función objetivo Lineal representando los costes de realizar cada una de las posibles asignaciones entre Agentes y Puestos.

Existen dos casos de función objetivo, en función de si el Criterio de Evaluación tiene en cuenta la Reserva Enumerada o no la tiene en cuenta:

- En caso de que no la tenga en cuenta, la Función Objetivo quedará como sigue:

$$\sum_{a \in A} \sum_{p \in P} (x_{ap} * C^c.coef(a, p))$$

- En caso de que sí se tenga en cuenta, la Función Objetivo quedará como sigue

$$\sum_{a \in A} \sum_{p \in P} (x_{ap} * C^c.coef(a, p)) + \sum_{p \in RE} (C^c.offset * \sum_{a \in A} (x_{ap}))$$

Así, los Modelos de Optimización, una vez añadido el Objetivo, quedarán:

#### Si no se tiene en cuenta la Reserva Enumerada

$$\min / \max \sum_{a \in A} \sum_{p \in P} (x_{ap} * C^c.coef(a, p))$$

sujeto a:

$$\begin{array}{ll} x'_a + \sum_{p \in P} x_{ap} = 1 & \forall a \in A \\ x_{ap} = 0 & \forall a \in A, p \in P, \exists c \in Ct.q. c.Compatible(a, p) = NO \\ \sum_{a \in A} (\sum_{p \in cp.CONJ} (x_{ap} \leq cp.MC)) & \forall cp \in CP \\ \sum_{a \in cp.CONJA} \sum_{p \in cp.CONJP} x_{ap} \leq cp.MC & \forall cm \in CP \\ \sum_{p \in P(cv)} x_{a_1 p} \leq x'_{a_2} + \sum_{p \in P(rel(cv, a_2))} x_{a_2 p} & \forall ra \in RA, \forall a_1, a_2 \in ra.A, a_1 \neq a_2, \forall cv \in CV_a^{ra} \\ \sum_{a \notin r.AG} \sum_{p \in r.PU} x_{ap} \leq M * (\sum_{a' \in r.AG} \sum_{p' \in r.PU} x_{a' p'}) & \forall r \in RPP \end{array}$$

#### Si se tiene en cuenta la Reserva Enumerada

$$\sum_{a \in A} \sum_{p \in P} (x_{ap} * C^c.coef(a, p)) + \sum_{p \in RE} (C^c.offset * \sum_{a \in A} (x_{ap}))$$



sujeto a:

$$\begin{array}{ll}
x'_a + \sum_{p \in P} x_{ap} = 1 & \forall a \in A \\
x_{ap} = 0 & \forall a \in A, p \in P, \exists c \in Ct.q. c.Compatible(a, p) = NO \\
\sum_{a \in A} (\sum_{p \in cp.CONJ} (x_{ap} \leq cp.MC)) & \forall cp \in CP \\
\sum_{a \in cp.CONJA} \sum_{p \in cp.CONJP} x_{ap} \leq cp.MC & \forall cm \in CP \\
\sum_{p \in P(cv)} x_{a_1 p} \leq x'_{a_2} + \sum_{p \in P(rel(cv, a_2))} x_{a_2 p} & \forall ra \in RA, \forall a_1, a_2 \in ra.A, a_1 \neq a_2, \forall cv \in CV_a^{ra} \\
\sum_{a \notin r.AG} \sum_{p \in r.PU} x_{ap} \leq M * (\sum_{a' \in r.AG} \sum_{p' \in r.PU} x_{a' p'}) & \forall r \in RPP
\end{array}$$

### Algoritmo Generación de Columnas por Coeficiente (siar::pp::AlgoritmoPGCCPX)

Este algoritmo usa la técnica de Generación de Columnas para encontrar una solución. El uso de esta técnica se basa en el proceso iterativo de ir resolviendo uno tras otro un modelo de CPLEX no entero (esto es, sin usar variables enteras), cada uno de ellos con las mismas restricciones pero con un número de variables diferentes o empleando otro juego de variables. Se deben cumplir dos condiciones:

\* Que el coste de resolver cada modelo sea mejor al coste de resolver el modelo anterior.

\* Que cada modelo no contenga todas las variables del problema original, esto es, cuando el número de variables de algún modelo sea igual al número de variables que plantea el problema entonces el algoritmo acaba.

La heurística que plantea pues el algoritmo es la siguiente:

1. Se construye un primer modelo no entero con la totalidad de restricciones del problema y una serie de variables iniciales que se escogen de forma diagonal, esto es, se pretende que todos los agentes y el mayor numero de puestos sean recogidos por las variables del primer modelo.

Por ejemplo, si se tienen 3 agentes y 3 puestos se escogerían las variables  $x_{11}, x_{22}, x_{33}$ .

2. Se resuelve este primer modelo añadiendo los costes de la función objetivo asociados a las variables que usa este modelo y cuya formulación es idéntica a la del Algoritmo anterior.

3. Se generan las nuevas variables a añadir al modelo de manera que el nuevo modelo presente un coste mejor que el actual (Generador).

La técnica de construcción de estas nuevas variables no se va a explicar por requerir conocimientos matemáticos más profundos.

4. Se accede al paso 6 en el caso de que se den alguna de las siguientes circunstancias:

- Que no se puedan generar más variables.
- Que se haya superado un tiempo límite máximo de ejecución.



- Que se haya repetido el mismo valor de la función objetivo un número máximo de veces.

5. Se resuelve el modelo, resultado de añadir a las variables ya existentes las nuevas variables generadas y añadiendo los costes de las nuevas variables que se obtienen del criterio de evaluación del algoritmo. Se vuelve al paso 3.

6. Se resuelve el último modelo obtenido del paso 5 pero tratando ahora las variables como enteras y se devuelve la solución obtenida de resolver este modelo.

Los modelos de generación de columnas usan la misma formulación matemática que los modelos CPLEX.

### **Algoritmo Solver por Ordenación (siar::pp::AlgoritmoCriterioOrdenacion)**

Este Algoritmo plantea una heurística con objeto de realizar una asignación óptima en cuanto al Criterio de Evaluación por Ordenación se refiere. Se puede observar que, dado el caso en que *todos* los Puestos de Asignación tuviesen exactamente la misma lista de Agentes de Asignación ordenados con respecto a las prioridades, se podría utilizar la siguiente heurística:

1. Ordenamos los Agentes de Asignación en función de su orden en la lista única existente
2. Vamos asignando un Puesto de Asignación a cada Agente por el orden inferido. Cada vez que se asigna uno de estos Puestos de Asignación, se *propagan* las restricciones de modo que se eliminen los Puestos de Asignación que ya no son asignables.

De esta forma, garantizaríamos que, siempre que se produjera un conflicto, este sería *justificable*, ya que si el Agente que entra en conflicto no puede obtener un Puesto de Asignación es por una de estas dos razones:

- O bien un Agente de Asignación más prioritario se lo ha llevado antes
- O bien el hecho de que un Agente más prioritario se llevara otro Puesto de Asignación produjo que el Puesto de Asignación preferido no se haya podido asignar.

En cualquiera de estos dos casos, se consideraría que el conflicto es *justificable*, por lo que no sería un problema a la hora de determinar la *bondad* de la solución.

La realidad es, en general, otra, que hace que esta técnica no funcione directamente. Ocurren dos cosas que lo impiden:

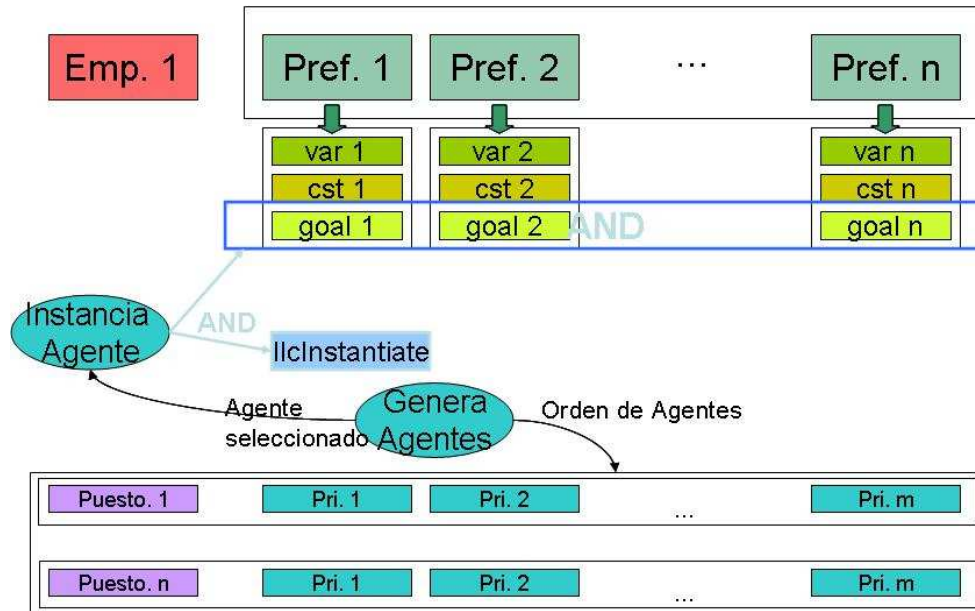
- No siempre las Listas de Agentes de Asignación para los Puestos de Asignación son iguales. Es decir, dependiendo de la naturaleza del Puesto de Asignación, pueden aparecer Listas de Agentes distintas.
- Aunque es importante no generar conflictos en la solución, es más importante el no dejar Agentes de Asignación sin asignar, por lo que, en cualquier caso, la solución ofrecida por este algoritmo no será óptima si deja Agentes de Asignación sin Asignar.



Por ello, es necesario tomar medidas que gestionen estos dos problemas, usando de todas formas un algoritmo heurístico. Estas medidas son:

- Saltos atrás *inteligentes* a lo largo de la búsqueda, para solucionar el problema de las listas de Agentes distintas (y eventualmente para solucionar el problema de los Agentes no asignados)
- Propagación más potente, para solucionar el problema de los Agentes no Asignados.

Todas estas técnicas serán expuestas posteriormente. Por ahora, se introduce un gráfico que muestra la técnica seguida para solucionar el Modelo de Optimización y, a continuación, se entrará en más detalle.



#### 4.2.2.1 Algoritmo de Búsqueda (Genérico)

En general, se puede dividir el Algoritmo de Búsqueda en dos fases principales:

- **Ordenación de los Agentes** de Asignación con respecto a las Prioridades de los Puestos
- **Instanciación de cada Agente** de Asignación con respecto a sus Preferencias

##### 4.2.2.1.1 Ordenación de los Agentes

El objetivo de esta fase consiste en conseguir una ordenación única de Agentes de Asignación de modo que la Asignación que conseguiríamos con el método explicado al principio de esta sección genere el menor número de conflictos. De este modo, las técnicas de reparación tendrán un impacto mínimo sobre el rendimiento del Algoritmo.

Así, el proceso se divide en dos pasos:

- Conseguir una colección de conjuntos de Puestos de Asignación con ordenaciones similares.
- Mezclar las ordenaciones producidas por dichos conjuntos de forma que sea suficientemente buena en el sentido expresado con anterioridad.



Para conseguir una colección de conjuntos de Puestos de Asignación se sigue el siguiente algoritmo:

1. Sea  $pri$  la primera prioridad del conjunto de Prioridades contenido en el Criterio de Evaluación
  - i. Hacemos  $pri\_actual := pri$
  - ii. Hacemos  $conj\_inicio := P$ , siendo  $P$  el conjunto de todos los Puestos de Asignación
  - iii. Hacemos  $C := \{conj\_inicio\}$
2. Mientras  $pri\_actual \neq nil$ 
  - i. Para cada conjunto  $conj$  perteneciente a  $C$ 
    - a) Si  $pri\_actual$  induce distintas ordenaciones  $(O_1, ..., O_n)$  sobre los Puestos de  $conj$ , entonces creamos tantos conjuntos de Puestos  $(C_1, ..., C_n)$  como ordenaciones, siendo  $C_i$  el conjunto de Puestos de Asignación pertenecientes a  $conj$  tales que  $pri\_actual$  induce la ordenación  $O_i$  sobre el Puesto.
  - ii. Hacemos  $C := \{C_1, ..., C_n\}$
  - iii. Hacemos  $pri\_actual := pri\_actual.siguiente$ .
3. Devolvemos la colección  $C$  resultante.

Sabemos ahora que esta colección de Conjuntos de Puestos de Asignación en realidad nos está representando una colección de Ordenaciones *distintas* de Agentes de Asignación con respecto a Puestos de Asignación para conjuntos de Puestos de Asignación. Ahora tenemos que inferir una Ordenación de Agentes a partir de estas Ordenaciones inferidas. Para ello, disponemos en la actualidad de dos técnicas:

- O bien escogemos directamente la Ordenación cuyo Conjunto de Puestos de Asignación  $C_i$  tenga una Cardinalidad mayor, de modo que estamos escogiendo la ordenación *más representativa*.
- O bien damos un valor a cada Agente  $a$  igual a  $\sum_{i \in \{0, ..., n\}} posicion(a, O_i)$ , y ordenamos los Agentes por este valor, de modo que estamos escogiendo la ordenación *más equitativa*.

La elección de cuál de estas dos técnicas se utiliza vendrá marcada por un parámetro.

Una vez conseguida esta lista ordenada de Agentes, procedemos a la instanciación de cada uno de ellos en función de sus preferencias, en el orden marcado por la Lista. De esto se encarga el Goal *GenerarAgentesI*, cuya estructura es de este estilo:

```
IlcGoal GenerarAgentesI::execute() {
  Agente *agente = NULL;

  // Seleccionar siguiente Agente sin instanciar, siguiendo el orden
  // inferido y guardarlo en 'agente'

  return IlcAnd(InstanciarAgente(getSolver(), _algoritmo, agente),
```



```

        this);
    }

```

#### 4.2.2.1.2 Instanciación de cada Agente

Instanciar un Agente consiste en asignarle un Puesto de Asignación, o bien decidir que se queda como NO\_ASIGNADO. Esta asignación se realiza de forma indirecta, a través de las preferencias. Así, se van seleccionando las Preferencias del Agente, y, por cada una de ellas, dependiendo de su tipo, se crea una *Variable Restringida*, una *Restricción* y un *Goal*. A continuación se describe cada uno de estos objetos:

- La *Variable Restringida* que se crea depende del modo en que se expresa la preferencia del Agente:
  - Si la preferencia es *Inferida*, se crea una *Variable Restringida* cuyo dominio es el resultado de llamar al método *getValoresPosibles* de la preferencia.
  - Si la preferencia es *Enumerada*, se crea una *Variable Restringida* cuyo dominio son todos los valores de la preferencia del Agente.
  - Si la preferencia es *Enumerada por Conjuntos*, se crea una *Variable Restringida* cuyo dominio es la unión de los Conjuntos que expresan la preferencia del Agente.
- La *Restricción* que se crea sencillamente mantiene la relación entre los valores de la *Variable Restringida* y los posibles Puestos de Asignación para asignar al Agente. Esta restricción queda definida por:
  - *Invariante*:
    - Si no existe ningún Puesto en el dominio del Agente tal que el valor de la Preferencia con respecto a ese puesto sea  $v$ , entonces  $v$  no puede estar en el dominio de la *Variable Restringida* representando a la preferencia
    - Si no existe el valor  $v$  en el dominio de la *Variable Restringida* que representa la preferencia del Agente, entonces no puede existir ningún Puesto en el dominio del Agente cuyo valor con respecto a la preferencia sea  $v$ .
  - *Métodos para mantener el invariante*:
    - Mantenimiento Inicial (post): Inicialmente, se revisan los valores de los dominios tanto de la preferencia como del Agente, y se eliminan los que son inconsistentes
    - Mantenimiento Dinámico (propagate): El Mantenimiento dinámico se hace mediante un sistema de *soportes*. Este sistema consiste en mantener una justificación (soporte) para cada uno de los valores presentes en el dominio de la preferencia. Una *justificación* para un valor  $v$  del Dominio de la preferencia es un Puesto  $p$  en el dominio del Agente cuyo valor con respecto a la preferencia sea  $v$ . Para conseguir esto, se realizan dos acciones:
      1. Cuando se elimina un Puesto  $p$  del dominio de Puestos del Agente, se chequea si este Puesto era soporte de algún valor  $v$  de la preferencia.:
        - i. Si es así, se busca otro Puesto  $p'$  en el dominio de Puestos del Agente cuyo valor con respecto a la preferencia sea  $v$ :
          - a) Si existe este Puesto  $p'$ , se pone como soporte de  $v$ , y se continúa



- b) Si no existe este Puesto, se elimina  $v$  del dominio de la preferencia y se continúa
  - ii. Si no, no se hace nada
- 2. Cuando se elimina un valor  $v$  del dominio de la preferencia, se eliminan todos los Puestos de Asignación del dominio del Agente tales que su valor con respecto a la preferencia sea  $v$ .
- El *Goal* que se crea pretende instanciar la *Variable Restringida* asociada a la preferencia, en el orden de valores preferido por el Agente. Así, dependiendo de la forma de expresar la preferencia, tenemos tres Goals:

- *InstanciaVariableInferido*: Este Goal se encarga de instanciar la Variable Restringida asociada a la preferencia de por un orden inferido

```
IlcGoal InstanciaVariableInferidoI::execute() {
    IlcInt valPref = IloIntMax;

    if ( !_varPref.isBound() ) {
        // Seleccionamos, el valor más pequeño o más grande posible de
        // la Variable Restringida representando a la preferencia, en
        // en función de si la preferencia del Agente es por
        // ORDEN_CRECIENTE o ORDEN_DECRECIENTE, respectivamente

        return IlcOr ( _varPref == valPref,
                       IlcAnd(_varPref != valPref, this) );
    }
}
```

- *InstanciaVariableEnumerado*: Este Goal se encarga de instanciar la Variable Restringida asociada a la preferencia a partir de una enumeración de valores

```
IlcGoal InstanciaVariableEnumeradoI::execute() {
    IlcInt valPref = IloIntMax;

    if ( !_varPref.isBound() ) {
        // Seleccionamos el primer valor de la preferencia del
        // Agente que esté en el dominio de la variable _varPref

        return IlcOr ( _varPref == valPref,
                       IlcAnd(_varPref != valPref, this) );
    }
}
```

- *InstanciaVariableEnumeradoConjuntos*: Este Goal se encarga de instanciar la Variable Restringida asociada a la preferencia a partir de una enumeración de Conjuntos de valores.

```
IlcGoal InstanciaVariableInferidoEnumeradoConjuntosI::execute() {
    IlcIntArray conjPref;

    if ( !_varPref.isBound() ) {
        // Seleccionamos el primer conjunto de la lista de conjuntos
        // Agente que esté en el dominio de la variable _varPref

        return IlcOr ( IlcMember(_varPref, conjPref),
                       IlcAnd(IlcNotMember(_varPref, conjPref),

```



```

        this) ) );
    }
}

```

En última instancia, por si después de evaluar todas las preferencias quedara más de un Puesto compatible con todas las elecciones realizadas, se ejecuta un *IlcInstantiate* sobre la variable de los Puestos del Agente.

Estos Goals, restricciones y variables representan la heurística básica que se define al principio de esta sección. A este respecto hemos identificados dos problemas fundamentales:

1. Existencia de ordenaciones distintas de Agentes de Asignación en función de características del Puesto de Asignación concreto (perfil FSC, Escalafón condicionado, etc., ): Para este caso, utilizamos técnicas de *Backtrack Inteligente*, que se describirán posteriormente.
2. Dominancia del Criterio de no dejar Agentes sin asignación. Utilizamos dos técnicas distintas para resolver este problema:
  - i. Técnicas de propagación más completas
  - ii. *Backtrack Inteligente*

En las siguientes dos secciones se describirán las técnicas utilizadas. Cabe decir, en cualquier caso, que estas dos técnicas son en cierto modo equivalentes. Es decir, en el momento en que se realiza un *Backtrack Inteligente* es porque la propagación anterior no ha sido capaz de detectar que este *Backtrack* iba a hacerse, y una propagación más completa lo hubiese evitado.

#### 4.2.2.2 Técnica de *Backtrack Inteligente*

Durante el proceso de asignación, en cualquiera de los Goals que se ejecutan, podemos encontrarnos con una situación *incorrecta*. En la actualidad nos encontramos con dos situaciones posibles:

- Existe un Puesto que el Agente  $A_1$ , que se está instanciando actualmente, prefiere a cualquiera de los que se le pueden asignar en este momento, y este Puesto ha sido asignado a otro Agente  $A_2$  que fue instanciado con anterioridad en este proceso de Asignación. Resulta, además, que  $A_1$  es prioritario a  $A_2$  para ese Puesto.
- Detectamos que el número máximo de Agentes no asignados va a exceder el máximo permitido (inferido de una solución anterior)

En estos dos casos, intentaríamos deshacer la decisión que llevó a esta incorrección y continuar la búsqueda.

Dependiendo del caso, la forma de deshacer la incorrección es distinta.

##### 4.2.2.2.1 Técnica de *Backtrack Inteligente*: Prioridades

A la hora de introducir esta técnica es conveniente introducir serie de objetos que van a entrar en juego:

- Llamaremos  $A$  al Agente que estamos asignando en la actualidad



- Llamaremos  $P_1, \dots, P_m$  a las preferencias tenidas en cuenta en el Criterio de Evaluación
- $A_1, \dots, A_n$  serán los Agentes, ordenados por orden de instanciación, que han sido asignados *antes* que  $A$  en este proceso de asignación
- Llamaremos  $L(A_i, P_j)$  al Label asignado al Punto de Decisión (IlcOr) que se ejecutó al instanciar la preferencia  $P_j$  para el Agente  $A_i$ .
- Llamaremos  $L(A_i)$  al Label asignado al Punto de Decisión (IlcOr) que se ejecutó al llamar al *IlcInstantiate* sobre el Agente  $A_i$ .

Dicho esto, en *todos* los Goals de instanciación de preferencias se incluirá un chequeo previo como el que sigue, suponiendo que estamos instanciando la preferencia  $P_i$ :

1. Hacer  $A_S := \text{nil}$  ( $A_S$  es el Agente Seleccionado para hacer el Backtrack)
2. Hacer  $V_S^p :=$  el valor de la preferencia  $P_i$  que el Agente  $A$  prefiere entre todos los que se le pueden asignar en este momento.
3. Para cada Agente  $A' \in \{A_n, \dots, A_1\}$  (notar el orden inverso):
  - i. Obtenemos el Puesto  $P'$  que le fue asignado al Agente  $A'$ .
  - ii. Si:
    - a) El Puesto  $P'$  es compatible en todos los sentidos al Agente  $A$ . En este sentido debe de ser compatible con respecto a todas las restricciones definidas en el Modelo.
    - b) El Puesto  $P'$  tiene los mismos valores con respecto a las preferencias  $P_1, \dots, P_{i-1}$  que los seleccionados por el Agente  $A$  hasta la preferencia  $P_i$  (que es la que estamos instanciando)
    - c) El valor del Puesto  $P'$  con respecto a la preferencia  $P_i$  es preferido a  $V_S^p$  por el Agente  $A$ .
    - d)  $A$  es más prioritario que  $A'$  para el Puesto  $P'$ .

Entonces hacemos:

- a)  $A_S := A'$
- b)  $V_S^p := P_i.\text{valor}(P')$

iii. Si no, continuamos al Agente anterior en la lista.

4. Si  $A_S \neq \text{nil}$ , entonces hacemos un Backtrack al Label  $L(A_S, P_i)$ .

En resumen, lo que hacemos con esta técnica de backtrak es, cada vez que decidimos una preferencia para un Agente, ver si existiese otro Agente que hubiera sido asignado antes que este y al que le hubiéramos asignado un Puesto que el nuestro prefiriera a cualquiera de los que le podemos dar ahora, y además fuera más prioritario para el mismo.



De entre todos los Agentes que cumplen esta propiedad, elegiremos el que tenga el Puesto que más prefiera, y de entre ellos, el que haya sido asignado el último.

De esta manera, evitamos los conflictos que pudieran haber sido generados por una ordenación incorrecta de los Agentes.

#### 4.2.2.2.2 Técnica de Backtrack Inteligente: NO ASIGNADO

Aunque no podemos asegurar que nunca se va a producir que un Agente resulte NO\_ASIGNADO, en muchos casos podemos recibir información sobre el máximo número de no asignados que podemos conseguir. Esto ocurre, en general, cuando, durante la ejecución del Algoritmo Fixings, y anteriormente a la ejecución de este Algoritmo Específico, hemos ejecutado otro algoritmo, que nos ha dado una solución con un número de NO\_ASIGNADOS.

En los Casos de Uso que tenemos en la actualidad, este Algoritmo ha sido el correspondiente a la resolución del Criterio de Evaluación por Coeficientes (Prioridad de Cubrimiento).

Entonces, el objetivo de esta técnica va a ser el de evitar aumentar el número de Agentes No Asignados con respecto a la última solución obtenida. A la hora de introducir esta técnica es conveniente introducir serie de objetos que van a entrar en juego:

- Llamaremos  $A$  al Agente que estamos asignando en la actualidad
- Llamaremos  $E_{NA}$  a la estimación sobre el número de Agentes que van a quedar No Asignados en el momento de instanciar este Agente. Este número será obtenido mediante el chequeo de la cota inferior de  $D(y_F)$ , es decir, la variable que representa el número de Agentes asignados al Puesto  $F$  (ficticio o NO\_ASIGNADO).
- $A_1, \dots, A_n$  serán los Agentes, ordenados por orden de instanciación, que han sido asignados *antes* que  $A$  en este proceso de asignación
- Llamaremos  $L(A_i, P_j)$  al Label asignado al Punto de Decisión (IlcOr) que se ejecutó al instanciar la preferencia  $P_j$  para el Agente  $A_i$ .
- Llamaremos  $L(A_i)$  al Label asignado al Punto de Decisión (IlcOr) que se ejecutó al llamar al *IlcInstantiate* sobre el Agente  $A_i$ .

Dicho esto, en *todos* los Goals de instanciación de preferencias se incluirá un chequeo previo como el que sigue, suponiendo que estamos instanciando la preferencia  $P_i$ : Si en ese momento identificamos que  $E_{NA}$  es mayor que el número máximo de Agentes No Asignados, entonces:

1. Hacer  $A_S := \text{nil}$  ( $A_S$  es el Agente Seleccionado para hacer el Backtrack)
2. Si el Puesto Asignado al Agente  $A$  es  $F$ , hacer  $A_{NA} := A$ , en otro caso, hacer  $A_{NA} := \text{nil}$ .
3. Para cada Agente  $A' \in \{A_n, \dots, A_1\}$  (notar el orden inverso):
  - i. Obtenemos el Puesto  $P'$  que le fue asignado al Agente  $A'$ .
  - ii. Si el  $P' = F$ , entonces hacer  $A_{NA} := A'$
  - iii. En otro caso, si:
    - a)  $A_{NA} \neq \text{nil}$ , y



- b) El Puesto  $P'$  es compatible en todos los sentidos al Agente  $A$  ( en este sentido debe de ser compatible con respecto a todas las restricciones definidas en el Modelo)

hacer  $A_S := A'$

4. Si  $A_S \neq \text{nil}$ , entonces hacemos un Backtrack al Label  $L(A_S)$ .

En resumen, lo que hacemos con esta técnica de backtrak es, cada vez que decidimos una preferencia para un Agente, chequeamos si en este momento estamos ya excediendo el número de NO\_ASIGNADOS permitido. En este caso, buscamos el primer Agente no Asignado en la Lista de Agentes, tal que exista un Puesto que ha sido asignado con anterioridad para el que este es compatible, y hacemos un *backtrack* a ese Agente, con objeto de intentar darle otro.

#### 4.2.2.2.3 Técnicas de Propagación utilizadas

Con respecto a la parte de Búsqueda, y con objeto de propagar mejor, hacemos dos cosas:

- Implementamos las restricciones adicionales usando técnicas de propagación eficientes, como es el caso de la técnica de *soportes* utilizada en la restricción que mantiene la correspondencia entre los valores de las preferencias y los de los Puestos asignados a los Agentes (descrita con anterioridad)
- Usamos preprocesos para las Restricciones de Partición de Puestos.
- Usamos técnicas de *mirar hacia adelante*.

Las técnicas de mirar hacia adelante consisten básicamente en, a la hora de decidir el valor que se asigna a una variable (ya sea esta la correspondiente a una Preferencia o a un Puesto), se prueban varias posibilidades, examinando sus efectos y decidiendo a posteriori.

En nuestro caso, utilizamos estas técnicas con objeto de ver si la asignación que produzco provoca que se deje sin asignar a un Agente

En cuanto a los ***preprocesos para las Restricciones de Partición de Puestos***, consisten en, antes de instanciar a un Agente, visitar todas las Restricciones de Partición de Puestos. Para cada una de ellas, y para cada una de ellas (rpp):

1. Si ya existe un Agente  $a \in \text{rpp.AG}$  tal que  $D(V_a) \subseteq \text{rpp.PU}$ , no hacemos nada
2. En otro caso, eliminamos del dominio del Agente a instanciar los puestos pertenecientes a rpp.PU.

Aunque este preproceso sea un poco más *fuerte* que la restricción de Partición de Puestos en sí, el objetivo de este Algoritmo es encontrar una solución *suficientemente buena*, con el objeto de mejorarla con posterioridad con Algoritmos de Búsqueda Local.

#### 4.2.2.2.4 Conclusiones



Con esto, los Goals que creamos para instanciar las preferencias quedarían como sigue:

- *InstanciaVariableInferido*

```
IlcGoal InstanciaVariableInferidoI::execute() {
    IlcInt valPref = IloIntMax;

    // Chequeamos si podemos hacer un backtrack inteligente, ya sea
    // por una preferencia o por un NO_ASIGNADO. En caso de poder
    // hacerlo, se salta desde aquí, así que la ejecución de este
    // Goal se interrumpe.

    if ( !_varPref.isBound() ) {
        // Para cada valor de '_varPref', ordenado por preferencia del
        // Agente, chequeamos si produce que otro Agente se quede sin
        // asignar:
        // - Si la respuesta es NO, seleccionamos este valor
        // - En otro caso, continuamos
        // Si después de intentarlo con todos los valores, todos ellos
        // dejan algún Agente sin asignar, elegimos el más preferido.

        return IlcOr ( _varPref == valPref,
                       IlcAnd(_varPref != valPref, this), label );
    }
}
```

- *InstanciaVariableEnumerado*

```
IlcGoal InstanciaVariableEnumeradoI::execute() {
    IlcInt valPref = IloIntMax;

    // Chequeamos si podemos hacer un backtrack inteligente, ya sea
    // por una preferencia o por un NO_ASIGNADO. En caso de poder
    // hacerlo, se salta desde aquí, así que la ejecución de este
    // Goal se interrumpe.

    if ( !_varPref.isBound() ) {
        // Para cada valor de '_varPref', ordenado por preferencia del
        // Agente, chequeamos si produce que otro Agente se quede sin
        // asignar:
        // - Si la respuesta es NO, seleccionamos este valor
        // - En otro caso, continuamos
        // Si después de intentarlo con todos los valores, todos ellos
        // dejan algún Agente sin asignar, elegimos el más preferido.

        return IlcOr ( _varPref == valPref,
                       IlcAnd(_varPref != valPref, this), label );
    }
}
```

- *InstanciaVariableEnumeradoConjuntos*

```
IlcGoal InstanciaVariableInferidoEnumeradoConjuntosI::execute() {
    IlcIntArray conjPref;

    // Chequeamos si podemos hacer un backtrack inteligente, ya sea
    // por una preferencia o por un NO_ASIGNADO. En caso de poder
    // hacerlo, se salta desde aquí, así que la ejecución de este
    // Goal se interrumpe.
```



```

if ( !_varPref.isBound() ) {
    // Para cada valor de la lista de conjuntos, tal que interseca con
    // el dominio de '_varPref', ordenado por preferencia del
    // Agente, chequeamos si produce que otro Agente se quede sin
    // asignar:
    // - Si la respuesta es NO, seleccionamos este valor
    // - En otro caso, continuamos
    // Si después de intentarlo con todos los valores, todos ellos
    // dejan algún Agente sin asignar, elegimos el más preferido.

    return IlcOr ( IlcMember(_varPref, conjPref),
                   IlcAnd(IlcNotMember(_varPref, conjPref),
                           this) ) );
}
}

```