UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UNIDAD ACADEMICA DE LOS CICLOS
PROFESIONAL Y DE POSGRADO DEL C.C.H.

PROBLEMAS DE CALENDARIZACION Y PROGRAMACION LOGICA

T E S I S

QUE PARA OBTENER EL TITULO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

FRANCISCO JAVIER MACIAS PEREZ

Director de Tesis :

Dr. David A. Rosenblueth L.

MEXICO, D. F., 1993

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Contenido

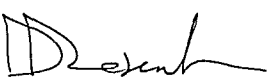
Resumen

El presente trabajo contrasta tres diferentes métodos de solución para problemas de satisfacción de restricciones. El primero de ellos se basa en Prolog, el segundo es una emulación de CHIP, y el tercero descansa en el Algoritmo de Relajación. La puesta en marcha de cada uno de los métodos se ejecuta en Prolog. Para llevar a cabo la comparación entre estos métodos, se escogieron problemas propios de la programación con restricciones, dando especial preponderancia a los problemas de calendarización. A través de dichas implantaciones se hace patente la utilidad de la programación lógica frente a los problemas orientados a restricciones. La comparación llevada a cabo mostró que a medida que el tamaño de la entrada crece, la implantación basada en el Algoritmo de Relajación invierte más tiempo en su ejecución, con respecto a las otras dos implantaciones, mientras que la implantación basada en PROLOG se ejecuta con mayor rapidez.

PROBLEMAS DE CALENDARIZACIÓN Y PROGRAMACIÓN LÓGICA

Resumen

El presente trabajo contrasta tres diferentes métodos de solución para problemas de satisfacción de restricciones. El primero de ellos se basa en Prolog, el segundo es una emulación de CHIP, y el tercero descansa en el Algoritmo de Relajación. La puesta en marcha de cada uno de los métodos se ejecuta en Prolog. Para llevar a cabo la comparación entre estos métodos, se escogieron problemas propios de la programación con restricciones, dando especial preponderancia a los problemas de calendarización. A través de dichas implantaciones se hace patente la utilidad de la programación lógica frente a los problemas orientados a restricciones. La comparación llevada a cabo mostró que a medida que el tamaño de la entrada crece, la implantación basada en el Algoritmo de Relajación invierte más tiempo en su ejecución, con respecto a las otras dos implantaciones, mientras que la implantación basada en PROLOG se ejecuta con mayor rapidez .



Vo. Bo. Dr. David A. Rosenblueth Laguette

Capítulo 1

Introducción

En este capítulo el lector encontrará la orientación del presente trabajo y se describirá el contexto en que se encuentran embebidos la programación lógica y los problemas de satisfacción de restricciones. También se mencionará el contenido de los capítulos subsecuentes.
Este trabajo está dedicado a mostrar en términos prácticos la utilidad y facilidades que la programación lógica proporciona a la resolución de problemas de satisfacción de restricciones, concretamente a los problemas de calendarización. Para ello se escogerá un problema de calendarización de actividades y se resolverá a través de tres diferentes herramientas derivadas de la programación lógica. Una de ellas es el lenguaje de programación PROLOG, otra es una emulación del lenguaje de programación CHIP, y la tercera se conoce como Aritmética de Intervalos. Para empezar ubiquemos a la programación lógica y luego a los problemas de satisfacción de restricciones.
La programación lógica es un paradigma de programación declarativo. Un paradigma, en programación, es un conjunto de conceptos que sirven de molde al diseño de un programa, y determinan la estructura de dicho programa [1].
Los programas contruidos bajo la óptica de la programación lógica se sustentan en la descripción de las características que debe reunir la solución buscada.
Este paradigma se dice lógico por estar orientado a la búsqueda de las conclusiones que siguen (o se construyen) de un conjunto de premisas.
Una característica que distingue a la programación lógica del resto de los paradigmas declarativos consiste en que al ver el programa como un con-

junto de axiomas, las respuestas que se obtienen son consecuencia lógica de dicho conjunto. Las respuestas así obtenidas pueden constituir un conjunto vacío o un conjunto con uno o más elementos. Si nuestro programa lógico y la estrategia de búsqueda son correctos y completos conseguiremos extender la búsqueda de soluciones a todas las posibles alternativas.
Algunas características importantes de los paradigmas declarativos, entre los que se encuentra la programación lógica, son: 1) el mecanismo de control principal es la recursión, 2) proporcionan al lector de un programa transparencia referencial y 3) sus implantaciones están libres de efectos laterales.
Decimos que un lenguaje soporta a un paradigma, cuando las implantaciones del lenguaje reflejan los principios y conceptos del paradigma. El lenguaje de programación PROLOG soporta como paradigma a la programación lógica.
En lo referente a los problemas de satisfacción de restricciones, estos se expresan a través de un conjunto de variables y un conjunto de restricciones. Cada restricción involucra a un subconjunto del conjunto de variables, estableciendo el modo en que los valores de dichas variables deben relacionarse, y las restricciones pueden expresarse como ecuaciones o como desigualdades [3].
Una característica propia de este tipo de problemas es que las variables no pueden instanciarse de modo independiente, ya que el valor que adopte una variable bajo una restricción debe adecuarse a los valores de las variables condicionadas por la misma restricción.
Los problemas de satisfacción de restricciones se encuentran emparentados con los problemas contemplados por la programación lineal y por la programación entera. El objetivo de la programación lineal y de la programación entera es encontrar el valor máximo o mínimo de una función sobre un conjunto de variables bajo alguna restricción, con la diferencia de que la programación lineal está orientada al manejo de valores continuos y la programación entera opera con valores discretos [2].
En el capítulo de antecedentes el lector encontrará una breve descripción de los problemas de satisfacción de restricciones y de la programación entera.
A continuación, en el mismo capítulo se justifica el uso de la programación lógica para problemas de satisfacción de restricciones, se describe el modo de operación de PROLOG, de CHIP y de la Aritmética de Intervalos.
En el capítulo de implantación se describe el modo en que se puso en marcha el problema de las ocho reinas desde dos perspectivas diferentes, la primera de ellas es PROLOG, y la segunda es una emulación de CHIP.

Además, en dicho capítulo se describe la implantación de un problema de calendarización a través de tres diferentes herramientas, en primer término una emulación de CHIP, en seguida a través de PROLOG, y por último se implanta el problema por medio de Aritmética de Intervalos.
El capítulo de conclusiones recoge los resultados más relevantes del trabajo, y el apéndice reune la totalidad de los programas descritos.

Notas Bibliográficas

[1] A. L. Ambler, M. M. Burnett, B, A. Zimmerman; Operational Versus Definitional: A perspective on Programming Paradigms; *Computer*, XXV(9):28-43; 1992.

[2] H. A. Taha; 1975; *Integer Programming. Theory, Applications, and Computations*; Academic Press; U.S.A.; p.t. 380.

[3] P. Van Hentenryck; 1989; *Constrain Satisfaction in Logic Programming*; MIT Press; U.S.A.; p.t. 224.

Capítulo 2

Antecedentes

A lo largo del presente capítulo, el lector encontrará una descripción intuitiva del problema de satisfacción de restricciones así como de programación entera. En éste se cita también la justificación al uso de la programación lógica para resolver problemas de satisfacción de restricciones, y un breve esbozo de la forma en que funcionan PROLOG, CHIP y la Aritmética de Intervalos.
El tratamiento computacional de un problema requiere que este último se especifique en un lenguaje propio para ser procesado por un ordenador.
Esto conlleva una descripción general de sus parámetros y la declaración de las propiedades que la respuesta (solución) debe reunir. De ser generada, la respuesta proporcionará valores para los parámetros correspondientes al problema planteado [2].
Un modelo particular de problemas es el de "satisfacción de restricciones".
En estos, además del conjunto de variables (parámetros), se establece un conjunto de restricciones, que deben ser satisfechas por la solución propuesta [4]. Este modelo surge de una técnica de resolución de problemas conocida como programación entera.
En la terminología propia de la programación entera se entiende por restricción una ecuación o desigualdad, la cual es impuesta por el problema y declara una condición a la que debe sujetarse la respuesta al problema.
La programación entera es una técnica usada para encontrar una solución máxima o mínima a un problema con restricciones, cuyas variables deben asumir valores discretos [4].
Un ejemplo de problema propio de la programación entera puede verse a través de la pregunta que se formula Alí Babá al encontrarse en la cueva de

los cuarenta ladrones. Alí Babá ha pronunciado la frase "Ábrete Sésamo" y una vez adentro de la cueva, puede apreciar muchas joyas de diferente valor, pero Alí Babá sólo puede cargar un peso limitado de joyas. Su problema es, entonces, ¿Qué joyas debe seleccionar de modo que el conjunto de éstas, posea el máximo valor? Pobre Alí Babá, ahora se enfrenta a un problema intrínsecamente difícil.

Otro ejemplo, que ilustra un caso de satisfacción de restricciones, puede plantearse de la siguiente manera: Athos, Porthos, Aramis y Artagnan, tienen, cada uno de ellos, una estatura propia. Si Porthos no es mayor que Athos, Aramis no es mayor que Artagnan, este último no es mayor que Porthos y Aramis es al menos tan alto como Athos; ¿Podrá el Cardenal Richelieu distinguir a cada uno de ellos tan solo por su estatura?

Ambos problemas encierran una característica común, la cual consiste en que no se conoce un método que permita descubrir, de manera "directa", una solución satisfactoria. En vez de ello, es necesario ensayar varias posibilidades de respuesta antes de encontrar la más acertada, esto es, generar un conjunto de instancias (valores para los parámetros o variables del problema) y probar si el conjunto satisface las condiciones impuestas.

En el tratamiento de los problemas descritos sería deseable contar con una máquina capaz de generar todas las posibles respuestas, y probar si alguna o algunas de ellas satisfacen las restricciones impuestas. Una herramienta que se adecúa al funcionamiento de dicha máquina, es la programación lógica, debido a que al mapear entre dos conjuntos (dominio y rango) a un solo elemento del dominio puede asociar varios elementos del rango, es decir, su comportamiento es relacional, o no determinístico (a diferencia de otros modelos de programación cuyo funcionamiento es funcional o determinístico).

La programación lógica, más allá de un estilo o una técnica para escribir programas, constituye una disciplina que puede ser igualmente útil para la escritura de programas, o para la definición inductiva de un problema. Esta disciplina propone el uso de la lógica como lenguaje de programación. Para ello, se sirve de un subconjunto de la lógica de primer orden conocido como cláusulas de Horn.

Las cláusulas de Horn proporcionan elementos para la especificación formal de problemas, a través de afirmaciones (hechos y reglas), preguntas, y una cláusula vacía. Un hecho denota una instancia de alguna relación, y posee una forma bien definida. Las cláusulas, a excepción de la cláusula vacía, poseen uno o varios átomos.

El carácter relacional de los átomos, proporciona a esta notación la posibilidad de ofrecer respuestas no determinísticas.
La programación lógica se ha implantado con muy buenos resultados en el lenguaje de programación PROLOG (programmation en logique).
PROLOG proporciona un mecanismo de retroceso (backtracking) que garantiza el tratamiento relacional de las cláusulas de Horn, es decir, PROLOG proporciona respuestas no determinísticas, y responde a una pregunta (cláusula de Horn) con tantas instancias como genere el programa lógico en cuestión, las cuales pueden ser, incluso, redundantes.
El poder de PROLOG reside en la posibilidad de ofrecer al usuario el algoritmo de unificación y la capacidad de retroceso.
Una herramienta alterna a PROLOG, que se ha considerado en el presente trabajo, es el lenguaje de programación CHIP.
El lenguaje de programación CHIP (Constrain Handling In Prolog) está orientado, como su nombre lo indica, al manejo de restricciones, bajo la perspectiva de la programación lógica. El espectro de aplicaciones para CHIP abarca una amplia gama de problemas combinatorios, entre los cuales podemos distinguir tres ramas básicas: dominios finitos, términos booleanos y términos racionales. Para cada uno de ellos, CHIP proporciona técnicas eficientes de manejo de restricciones, tales como técnicas de consistencia, unificación booleana, y un algoritmo simbólico similar al método Simplex. La unificación booleana se refiere al manejo de términos que involucran valores de verdad con el auxilio de los operadores lógicos.
El método Simplex es una técnica útil para encontrar un valor óptimo, propia de la programación lineal [5], de la que a su vez participa la programación entera. La parte de interés, en el caso que nos ocupa, reside en el manejo de técnicas de consistencia para dominios finitos.
El hecho de proporcionar a la programación lógica una extensión, tal como lo hace CHIP, ha impuesto, a su vez, una limitación importante, la cual consiste, para el caso del manejo de elementos en el dominio finito, en la necesidad de manejar de manera explícita cada uno de los elementos de que se compone un rango. Dada nuestra imposibilidad del manejo directo de CHIP, en el presente trabajo se emulará el funcionamiento de este lenguaje a través de PROLOG. Dicha emulación es muy simple, basta considerar que los dominios en CHIP son conjuntos finitos de constantes. De modo que nuestras implantaciones estarán basadas en enumeraciones.
Otra alternativa contemplada en el presente trabajo se conoce como Aritmé-

tica de Intervalos, la cual ha surgido como una respuesta a la necesidad de extender los beneficios de la programación lógica al cómputo que involucra aritmética de punto flotante. Con esta herramienta se busca eliminar el error inherente que resulta del uso de operaciones de punto flotante. El manejo aritmético de intervalos, se verifica a través de la sustitución de cada uno de los valores numéricos que limitan a un intervalo, por otros valores cuya representación en la máquina se encuentra libre de error. Las operaciones se manipulan como procesos de inferencia, los cuales reducen la longitud de cada intervalo, en un proceso de aproximaciones sucesivas; las operaciones son tratadas, en consecuencia, como restricciones.
El control de este proceso, es dirigido por el algoritmo de relajación.
Aún cuando la motivación original para la puesta en marcha de la Aritmética de Intervalos fue la eliminación de los errores por redondeo, nosotros aprovecharemos el poder de manipulación de restricciones.
El algoritmo de relajación permite reducir, con un solo proceso, un conjunto de operaciones (restricciones) sobre intervalos [1,3].

Notas Bibliográficas

[1] J. G. Cleary; Logical Aritmetic; *Future Computing Systems,* II(2): 125-149; 1987.

[2] M. R. Garey, D. S. Johnson; 1979; *Computers and Intractability. A guide to the theory of NP-Completeness*; W. H. Freeman and Co.; U.S.A.; p.t. 340.

[3] J. H. M. Lee, M. H. van Emden; Numerical Computation can be Deduction Deduction in CHIP; *Logical Programming Laboratory, Technical Report,* LP-19(DCS-184-IR); 1992.

[4] H. A. Taha; 1975; *Integer Programming. Theory, Applications, and Computations*; Academic Press; U.S.A.; p.t. 380.

[5] P. Van Hentenryck; 1989; *Constrain Satisfaction in Logic Programming*; MIT Press; U.S.A.; p.t. 224.

Capítulo 3

Implantación

El presente capítulo está orientado a la comparación entre tres implantaciones de un problema de satisfacción de restricciones, sobre tres diferentes herramientas de programación: PROLOG[2], CHIP[3] y Aritmética de Intervalos[1]. A modo de ilustración, se presenta el problema de las ocho reinas visto a través de PROLOG y de una emulación de CHIP. El problema que servirá para ponderar los tres métodos se refiere a la calendarización de un conjunto de actividades relacionadas entre sí.
Antes de presentar el problema de calendarización se mostrará el tratamiento del problema de las ocho reinas, para dar cuenta clara del modo en que CHIP trabaja con respecto a PROLOG. Con este fin, el problema se implantará en PROLOG "desnudo", es decir, además de la posibilidad de retroceso propia de PROLOG, la implantación se encontrará libre de toda heurística o método de búsqueda, y también se implantará con un algorítmo que emula a CHIP, y cuya puesta en marcha descansa a su vez sobre PROLOG. Este último proporcionará al modelo de emulación todas las características que la programación lógica le brinda a CHIP, en especial la posibilidad de generar y probar (estrategia que consiste en generar una instancia y probar si es acertada).

3.1 El problema de las ocho reinas

El problema de las ocho reinas constituye un buen ejemplo de problema de satisfacción de restricciones.

Este problema consiste en colocar ocho reinas de ajedrez en un tablero vacío de 64 escaques, distribuidas de modo que ninguna reina sea atacada por otra. Este problema exige que se ensayen diferentes posibilidades de respuesta antes de encontrar una que satisfaga las restricciones impuestas y pertenece a un conjunto de problemas para los cuales no existe, hasta hoy, un método que los resuelva "sin rodeos". A dicho conjunto de problemas se le conoce como Clase $\mathcal{NP}$-completa. La necesidad de probar diferentes posibilidades de solución queda perfectamente solventada por el mecanismo de retroceso de PROLOG (proporcionado también por CHIP).

3.1.1 PROLOG y el problema de las ocho reinas

Para dar solución al problema de las ocho reinas a través de PROLOG "desnudo" (recordemos que esto se refiere al método de generar y probar sin otra heurística), podemos solicitar la respuesta por medio de un predicado cuya única variable unifique con la lista de los escaques afectados, éste puede ser:

```
reinas(Y)
```

donde Y tendrá la forma de la lista `[(1,Y1), (2,Y2), (3,Y3), (4,Y4), (5,Y5), (6,Y6), (7,Y7), (8,Y8)]`, siendo el valor de Yn el número de la columna que corresponde a la reina que se encuentre en el enésimo renglón, de modo que cada pareja de valores (X,Y) represente la posición de un escaque del tablero ocupado por alguna reina. Para tal efecto, podemos proporcionar dicho formato y luego buscar que la respuesta sea instanciada sobre la retícula proporcionada:

```
reinas(Y) :- reticula(Y), solucion(Y).
reticula([(1,Y1),(2,Y2),(3,Y3),(4,Y4),
          (5,Y5),(6,Y6),(7,Y7),(8,Y8)]).
```

de este modo, la solución propuesta se adecuará al formato establecido por `reticula`. El predicado `solucion` debe satisfacer las siguientes condiciones:

- debe proporcionar un valor a la variable de cada pareja en la lista que recibe como parámetro,
- debe garantizar que cada reina (pareja coordenada) no ataque al resto de las reinas.

Dada la conveniencia de que este predicado revise, una por una, a cada pareja ordenada, lo definiremos de manera recursiva. El caso base para la recursión será la lista vacía, lo cual implica, en el momento de unificar, que ya han sido instanciadas y revisadas todas las parejas, en cuyo caso no se requiere satisfacer ninguna condición:

```
solucion([]).
```

El caso recursivo instancia y revisa la siguiente pareja ordenada, y en su llamada recursiva ignora a la pareja que ya fue revisada:

```
solucion([(X,Y)|Otras]) :- solucion(Otras), ...
```

La primera condición puede satisfacerse a través de una selección no determinística de posibles valores para la variable en cuestión. Para ello se puede utilizar un predicado que se cumple si un elemento es miembro de una lista:

```
miembro(X,[X|Y]).
miembro(X,[Y|Ys]) :- miembro(X,Ys).
```

El predicado `miembro` tiene éxito cuando el primer argumento se encuentra como cabeza de la lista del segundo argumento, según la primera cláusula, o bien si el primer argumento se encuentra contenido en algún lugar del resto de la lista, de acuerdo con la segunda cláusula, si esto no se cumple, el predicado fracasará. Pero ¿Qué ocurre si únicamente el segundo argumento se recibe instanciado? En ese caso, PROLOG unificará con la variable del primer argumento, cada uno de los elementos del segundo argumento, es decir, inicialmente unificará la variable con la cabeza de la lista, al hacer retroceso, su segunda respuesta unificará a la variable con la cabeza del resto de la lista original, nuevamente hará retroceso y su tercera respuesta unificará con la cabeza del resto del resto de la lista original, y así sucesivamente hasta alcanzar el último elemento de la lista. Entonces, si el predicado `solucion` utiliza la definición del predicado `miembro` para instanciar a la variable de cada pareja en la lista, con ello garantizará que dicha variable ensaye cada uno de los valores propuestos por `miembro`:

```
solucion([(X,Y)|Otras]) :- solucion(Otras),
                           miembro(Y,[1,2,3,4,5,6,7,8]),...
```

La segunda condición es que una reina no ataque a las otras, de ahí que debamos agregar un predicado `noataques` y definir su funcionamiento:

```
solucion([(X,Y)|Otras]) :- solucion(Otras),
                           miembro(Y,[1,2,3,4,5,6,7,8]),
                           noataques((X,Y),Otras).
```

El predicado noataques recibirá una pareja instanciada y una lista de parejas instanciadas (gracias a la previa llamada recursiva de solucion), y tendrá éxito si la reina posicionada en el escaque del primer argumento no ataca a ninguna reina colocada en algún escaque de la lista en el segundo argumento. Ello sólo se verificará si ninguna reina comparte la misma columna o alguna diagonal (gracias a la instancia del predicado reticula, sabemos que ningún par comparte el mismo renglón). El predicado noataques debe también revisar a cada pareja ordenada de la lista, por lo que debe definirse de manera recursiva. El caso base puede darse cuando ya no hay ninguna reina pendiente de comparación, es decir, cuando el segundo argumento es una lista vacía:

```
noataques((X,Y),[]).
```

En el caso recursivo el éxito de este predicado está sujeto a que la columna y diagonales del primer argumento sean distintos a los de la cabeza del segundo argumento:

```
noataques((X,Y),[(X1,Y1)|Otras]) :- Y =\= Y1,
                                    Y1 - Y =\= X1 - X,
                                    Y1 - Y =\= X - X1,
                                    noataques((X,Y),Otras).
```

El programa completo aparece en el apéndice con el nombre de **programa no. 1.**
En este programa se explota exclusivamente el retroceso de PROLOG como método de búsqueda. En seguida veremos cómo se resuelve el mismo problema, pero ahora emularemos, con PROLOG, la forma en que trabaja CHIP.

3.1.2 CHIP y el problema de las ocho reinas

Antes de describir la puesta en marcha cabe señalar que de los tres dominios de cómputo proporcionados por CHIP, nos interesa de manera especial el manejo de dominios finitos, una de cuyas características se refiere al manejo directo de cada elemento en los dominios discretizados presentes en los programas. Por este motivo se verá que en las emulaciones de CHIP, en alguna

parte del programa, se proporciona la "materia prima" con que éste ha de trabajar.
Para ver con mayor claridad la forma en que nuestra emulación de CHIP trata el problema de las ocho reinas veremos este mismo problema bajo una nueva perspectiva:
Se desea colocar en un tablero de ajedrez a ocho reinas. El tablero tiene ocho columnas y ocho renglones de escaques. Los escaques del tablero forman 15 diagonales en un sentido y 15 diagonales en sentido transversal al primero. Las reinas deben acomodarse de modo tal que ningún par de éstas compartan ni el mismo renglón, ni la misma columna, ni la misma diagonal en un sentido, ni la misma diagonal en el sentido transverso. Sólo por claridad en la exposición estableceremos la siguiente convención: El tablero es un sistema coordenado cuyo origen se encuentra en el ángulo inferior izquierdo. Hacia la derecha el valor de X se incrementa. Hacia arriba el valor de Y se incrementa. Las diagonales orientadas de abajo a la izquierda hacia arriba a la derecha serán numeradas del siguiente modo: La primera se encuentra abajo a la derecha, y la decimoquinta se encuentra arriba a la izquierda. Diremos que estas diagonales se encuentran en sentido normal. A las diagonales orientadas de arriba a la izquierda hacia abajo a la derecha las numeraremos de esta manera: La primera es la que se ubica abajo a la izquierda y la decimoquinta está arriba a la derecha. Al referirnos a estas diagonales les llamaremos diagonales inversas.
Cabe destacar que un escaque cualquiera ocupa un renglón, una columna, una diagonal normal y una diagonal inversa. Este cuarteto es único y no puede repetirse para ningún otro escaque. Al conocer el renglón Y y la columna X de un escaque, podemos calcular con facilidad las diagonales que corresponden a dicho escaque. Sea M una diagonal inversa y N una diagonal normal, tendremos que:

$$M = X + Y - 1$$

$$N = Y - X + 8$$

El problema planteado de este modo deja ver un hecho interesante. Cuando una reina ocupa un escaque, cancela a su renglón, la columna que ocupa, una diagonal normal y una diagonal inversa, las cuales no podrán ser ocupadas por ninguna otra reina. Regresemos ahora a nuesta implantación en *quasi*-CHIP.

Como en el caso del programa anterior, podemos solicitar la respuesta adecuada a través del predicado **reinas**, con una única variable que unifique con la lista de las posiciones requeridas:

```
reinas(X)
```

en donde la variable X puede instanciarse con la lista [X1,X2,X3,X4,X5, X6,X7,X8], donde el valor de Xn representa el número de columna que hace pareja con el n-ésimo renglón, el índice (posición en la lista de cada valor) representará, entonces, el número de renglón.
El predicado **reinas** puede condicionar su éxito a otro predicado que, además de instanciar la lista de posiciones, proporcione la "materia prima" con que trabajará el programa. Dicha "materia prima", según nuestro análisis, debe componerse de cuatro listas, dos de ellas con ocho elementos cada una, para los renglones y las columnas respectivamente, y otras dos con 15 elementos cada una, para las diagonales:

```
reinas(X) :- solucion( X,
                       [1,2,3,4,5,6,7,8],  [1,2,3,4,5,6,7,8],
                       [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
                       [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]).
```

Ahora bastará que el predicado **solucion** asigne a cada reina un elemento de cada lista. Para ello se le dará a la siguiente reina (de la primera a la última) el siguiente elemento (la cabeza) de la primera lista, y algún elemento de la segunda lista. Con estos dos elementos se calcularán los valores de las diagonales que corresponden al escaque (columna,renglón) asignado, y si estos aún no han sido retirados de su lista respectiva (es decir, no han sido asignados a otra reina) entonces se borran los cuatro valores de sus respectivas listas. El elemento borrado de la segunda lista (columnas) se incluye en la lista de respuesta, y se llama recursivamente al predicado **solucion** con los elementos que no fueron borrados, para que sean repartidos entre las reinas restantes. El caso base para la recursión será cuando todas las columnas y renglones ya hayan sido repartidos, es decir, cuando estas listas estén vacías:

```
solucion([],[],[],M,N).
```

en el caso recursivo, como ya se describió:

```
solucion([R|Rs],[X|Xs],Y,M,N):- borra(R,Y,Ys),
                                K is X + R - 1,
                                borra(K,M,Ms),
                                L is R - X + 8,
                                borra(L,N,Ns),
                                solucion(Rs,Xs,Ys,Ms,Ns).
```

ahora sólo resta definir el predicado **borra**, cuyo primer argumento sea un elemento incluido en la lista del segundo argumento, y en el tercer argumento la misma lista luego de haber eliminado de ella al primer argumento, y si el primer argumento no se encuentra contenido en el segundo, el predicado debe fracasar. El predicado **borra** debe definirse también de manera recursiva, para que pueda buscar al elemento de interés a lo largo de toda la lista, y su caso base será cuando el elemento buscado aparezca como cabeza de la lista:

```
borra(X,[X|Xs],Xs).
```

en el caso recursivo, bastará con lanzar la búsqueda al resto de la lista cuando el elemento buscado no se encuentre en la cabeza de la lista (es decir que el caso base no se satisfaga):

```
borra(X,[Y|Ys],[Y|Xs]) :- borra(X,Ys,Xs).
```

Con todos estos elementos se integra el programa completo, que aparece en el apéndice con el nombre de **programa no. 2.**
El programa implantado a través de PROLOG desnudo, presenta una ejecución más lenta que el programa recién citado, debido a que, en el primero, cada reina ensaya en su propio renglón cada una de las ocho posiciones que corresponden a las ocho columnas (en total 8^8 casos), mientras que en el caso de la emulación de CHIP, cada nueva reina ensaya sobre su renglón una posición menos que la reina anterior (en total 8! intentos).
Aquí podría parecer que CHIP es mejor (más rápido) en algunos casos, pero hay que recordar que PROLOG trabajó "desnudo", es decir sin más técnica que generar y probar. En el problema que a continuación se cita, veremos que a través de PROLOG puede reducirse en medida considerable el trabajo ejecutado por CHIP. Demos paso, entonces, a un problema de calendarización.

3.2 Un problema de calendarización

El problema que ahora consideraremos solicita ordenar (secuencializar) un conjunto de actividades, entre las cuales pueden existir condiciones de precedencia, ya sea por la naturaleza de las actividades, o bien porque comparten algún recurso y no pueden utilizarlo simultáneamente. El problema que trataremos es una modificación del que aparece descrito por Van Hentenryck [3], modificado a su vez de Bartusch. Las actividades por ordenar corresponden a una construcción. La mayor parte de las restricciones obedece a condiciones propias de las actividades. Sabemos, por ejemplo, que para plantar los cimientos antes debemos excavar las cepas para ese fin, o para vaciar el concreto se deben tener listas las cimbras, pero si se cuenta con una sola revolvedora, aún cuando se encuentren varias cimbras listas para recibir el concreto, primero deberá hacerse el vaciado de una, y sólo hasta terminar el vaciado en ésta, se podrá continuar con la siguiente.
El problema que atacaremos es una etapa de la construcción de un puente. Se trata de construir el almacén, practicar las excavaciones para asentar las columnas del puente, dar el acabado interior de ladrillo a las excavaciones, hacer la cimbra y el vaciado de cada columna (fuera de su fosa), remover las cimbras y quitar el almacén.
Puesto que debe atenderse a la precedencia de actividades y a la distribución de recursos, nuestra lista de actividades debe incluir, además del nombre y duración de la actividad, una lista de todas las actividades que preceden a la actividad en cuestión, y los recursos requeridos por ésta.
En las tablas 3.1 y 3.2 aparecen las actividades con sus nombres, duraciones, antecedentes y recursos respectivamente. Cabe mencionar que algunas actividades son ficticias y sirven sólo como puntos de referencia.
Éstas usan recursos también ficticios. Su existencia obedece a la necesidad de dar tiempos de espera entre una actividad y otra. Cada actividad será incluida como un hecho en una base de datos, que luego será consultada por los programas que generemos. La forma de los hechos que describirán a cada actividad será la siguiente:

```
actividad(nombre,duración,antecedentes,recursos).
```

donde **nombre** es el nombre que ya se citó para cada actividad, **duración** será una lista de elementos, en nuestro caso los números naturales, desde uno hasta el número que proporcione el valor de la duración de la actividad,

Actividad	*Nombre*	*Duración*
Inicio de la obra	pa	0
Excavaciones	a1, a2, a3, a4, a5, a6	4, 2, 2, 2, 2, 5
Construcción del almacén	ue	10
Tiempo de espera	fic	6
Cimbras	s1, s2, s3, s4, s5, s6	8, 4, 4, 4, 4, 10
Vaciados	b1, b2, b3, b4, b5, b6	1 c/u
Fraguados	ab1, ab2, ab3, ab4, ab5, ab6	1 c/u
Albañilería	m1, m2, m3, m4, m5, m6	16, 8, 8, 8, 8, 15
Quitar almacén	l	2

Tabla 3.1: Actividades de una construcción

Nombre	*Antecedentes*	*Recursos*
pa	ninguno	aux1
a1, a2, a3, a4, a5, a6	pa	exc
ue	pa	aux2
fic	pa	aux3
s1,s2,s3,s4,s5,s6	fic	ca1,ca2,ca3,ca4,ca5,ca6
b1,b2,b3,b4,b5,b6	s1,s2,s3,s4,s5,s6	rev
ab1,ab2,ab3,ab4,ab5,ab6	b1,b2,b3,b4,b5,b6	au4,au5,au6,au7,au8,au9
m1,m2,m3,m4,m5,m6	ab1,ab2,ab3,ab4,ab5,ab6	ma1,ma2,ma3,ma4,ma5,ma6
l	m1,m2,m3,m4,m5,m6	aux10

Tabla 3.2: Antecedentes y Recursos de las Actividades

antecedentes será una lista de las actividades que preceden a la actividad en cuestión, y **recursos** se refiere a los recursos requeridos por la actividad. Para que la base de datos quede completa debemos además agregar un predicado que contenga los recursos y su disponibilidad. Éste tendrá la forma:

```
dominios([ rec(nombre_1, duracion_1),
           rec(nombre_2, duracion_2),  ...
           rec(nombre_n, duracion_n) ]).
```

La base de datos completa se encuentra en el apéndice con el nombre de base **no. 1.**
A continuación veremos cómo opera la emulación de CHIP para obtener respuestas al problema de calendarización, y luego ese mismo programa lo modificaremos para obtener un programa en PROLOG, el cual, en principio, mejorará la primera ejecución.

3.2.1 CHIP y un problema de calendarización

El problema, como fue planteado, puede dividirse en otros dos subproblemas más elementales, los cuales son: uno, ordenar las activivdades según su precedencia, y el otro, distribuir de manera adecuada los recursos entre las actividades; por este motivo dividiremos la implantación en dos partes y luego integraremos las partes para obtener un solo programa. Es importante notar que un subproblema se ocupa de un conjunto de restricciones y otro subproblema se ocupa de otras restricciones. El problema original exige que todas las restricciones se encuentren satifechas, por lo que no basta con encontrar un respuesta que satisfaga un conjunto de restricciones y luego modificar esta respuesta para que satisfaga a un segundo conjunto de restricciones. Dicha modificación puede provocar que la respuesta viole el primer conjunto de restricciones, de modo que debe encontrarse una respuesta que satisfaga simultáneamente a todas las restricciones. Por este motivo se debe obtener un solo programa que resuelva el problema y no varios que intenten resolverlo por partes.

Subproblema de precedencia

Empezaremos con la parte que ordena las actividades según su precedencia. El programa consultará la base de datos que ya escribimos, y solicitará la

respuesta a través de un predicado cuya única variable unificará con la lista de actividades ordenadas por su precedencia, y responderá una, ninguna o tantas veces como posibles formas de ordenar las actividades según su precedencia. Su nombre será `pert`:

```
pert(Lista)
```

y su éxito estará condicionado al éxito de otros dos predicados. El primero de estos se llamará `todas`, y su único argumento unificará con la lista de los nombres de las actividades por ordenar:

```
todas([pa,a1,a2,a3, ...]).
pert(Lista) :- todas(AN), ...
```

El segundo predicado se llamará `programa`, recibirá la lista consultada por `todas`, la variable con que debe unificar la respuesta, y una lista vacía que servirá para acarrear las respuestas parciales:

```
pert(Lista) :- todas(AN), programa(AN,[],Lista).
```

El predicado `programa` tendrá éxito en dos casos. En el primer caso cuando la lista de los nombres de actividades esté vacía, es decir que no quede ninguna actividad pendiente, en cuyo caso diremos que ya terminó, y procederemos a unificar la lista de acarreo con la lista de respuesta:

```
programa([],Acts,Acts).
```

En el segundo caso tendrá éxito si es factible lo siguiente: borra un nombre de la lista de nombres, consulta la base de datos para conocer el conjunto de actividades que preceden a la actividad "borrada", observa si esas actividades precedentes ya fueron incluidas en la lista de acarreo, lo que simbólicamente significará que ya fueron ejecutadas, y al final se llamará recursivamente con las actividades restantes luego de haber borrado, y con la actividad borrada agregada a la lista de acarreo. Si la actividad borrada (seleccionada de manera no determinística) no posee un conjunto de antecedentes que se hayan ejecutado en su totalidad, el retroceso permitirá borrar (seleccionar) otra actividad:

```
programa(AN,AP,R) :- borra(A,AN,Acts), ...
```

El predicado **borra** es el mismo que se utilizó para el problema de las ocho reinas con la emulación de CHIP, y opera de igual manera.
Acto seguido, se consulta la base de datos con el nombre de la actividad "borrada", con la finalidad de conocer la lista de actividades que le preceden:

```
programa(AN,AP,R):-borra(A,AN,Acts),actividad(A,_,Pred,_),...
```

luego de conocerlas, el predicado **subconj** observará si esta lista de actividades (vista como conjunto) está incluida en la lista de actividades que fueron previamente borradas (analizadas) con éxito:

```
programa(AN,AP,R) :- borra(A,AN,Acts), actividad(A,_,Pred,_),
                     subconj(Pred,AP), ...
```

El predicado **subconj** tiene éxito si su primer argumento es un subconjunto de su segundo argumento. Para ello, verifica que cada uno de los elementos de la primera lista esté incluido en la segunda lista. Este predicado es recursivo, y su caso base se da cuando el primer argumento es la lista vacía, ya que todo conjunto posee como subconjunto al conjunto vacío. El caso en que esta definición se satisfaga luego de varias llamadas recursivas, implica que todos los elementos de la primera lista fueron encontrados en la segunda:

```
subconj([],_).
```

Para el caso recursivo, basta con buscar la cabeza de la primera lista en la segunda lista, y luego llamar recursivamente al predicado **subconj** con el resto de la primera lista y con la segunda lista tal como se recibió. La tarea de búsqueda se hará a través del predicado **miembro** que fue descrito en la implantación del problema de las ocho reinas en PROLOG desnudo:

```
subconj([X|Xs],Y) :- miembro(X,Y), subconj(Xs,Y).
```

Finalmente, el predicado **programa**, debe llamarse recursivamente con la lista de las actividades que no fueron borradas, la actividad borrada incluida en la lista (de acarreo) que contiene a las actividades que ya tuvieron éxito, y la variable auxiliar que proporcionará la respuesta. El programa completo se encuentra en el apéndice con el nombre de **programa no. 3.**
Cabe mencionar que este programa integra una permutación, ya que proporciona en su conjunto de respuestas a todas las posibles permutaciones que respetan el orden de precedencia. Con éste, resolvemos el primero de los subproblemas. En seguida atacaremos el segundo subproblema.

Subproblema de distribución de recursos

Ahora nuestro objetivo es buscar una distribución adecuada de los recursos entre las actividades (en realidad queremos todos los posibles modos de distribución aceptables).
El programa que diseñaremos para distribuir los recursos entre las actividades, es muy similar al que ya desarrollamos para satisfacer las necesidades de precedencia.
Utilizaremos un predicado `pert` cuya única variable unificará con la respuesta. Éste a su vez dependerá del éxito del predicado `todas`, el cual opera de igual modo que en el programa anterior.
Luego se consulta la base de datos a través del predicado `dominios`, que proporciona la lista de recursos con sus respectivas duraciones. Al último predicado del que depende (en forma inmediata) `pert`, le llamaremos `programa`, y recibe como parámetros a la lista de los nombres de todas las actividades, la lista de recursos y sus duraciones, una lista vacía que servirá como acumulador, y la variable del predicado `pert` con la cual se instanciará la respuesta:

```
pert(Lista) :- todas(AN), dominos(Doms),
               programa(AN,Doms,[],Lista).
```

Puesto que el predicado `programa` debe proporcionar a cada actividad la parte que requiera de algún recurso, se definirá de modo recursivo y actualizará, además, la lista de recursos, de modo que el recurso en cuestión sólo disponga de la parte que no haya sido solicitada por alguna actividad.
Así, el caso base de este predicado debe definir (exigir) que la lista de actividades haya sido procesada en su totalidad. Dicha lista debe estar vacía debido a que el procesamiento de cada actividad implicará que ésta sea borrada de la lista original.
En ese momento, la lista de acarreo contendrá a todas las actividades, cada una con su respectivo segmento de tiempo de recurso utilizado, entonces se instancia con ella la variable de respuesta:

```
programa([],_,Acts,Acts).
```

La definición recursiva, como ya se mencionó, toma un nombre de actividad de la lista respectiva a través del predicado `borra` (fue definido en el programa anterior), y consulta la base datos para conocer la duración y el recurso utilizado por dicha actividad:

```
programa(AN,Doms,AP,R) :- borra(A,AN,Acts),
                          actividad(A,Durac,_,Rec), ...
```

El siguiente predicado debe conocer el estado actual de la lista de recursos, asignar a la actividad en turno el segmento de tiempo del recurso (el nombre del recurso lo da la variable Rec) que le corresponda según su duración, y proporcionar la lista de recursos actualizada (luego de haber entregado un segmento de su tiempo), justo como si se tratase de un mazo de naipes (recurso) antes, durante, y después de darle cartas al jugador (actividad) en turno. Este predicado se llama clc_en_dom, y recibe como parámetros, el nombre del recurso utilizado por la actividad, la duración de la misma, la lista de recursos tal como la recibió el predicado programa, y dos variables, la primera para proporcionar a la actividad el segmento de tiempo del recurso que le corresponde, y la segunda para la actualización de la lista de recursos:

```
programa(AN,Doms,AP,R) :- borra(A,AN,Acts),
                          actividad(A,Durac,_,Rec),
                          clc_en_dom(Rec,Durac,Doms,
                                     Lst_tpo_us,Doms1),...
```

Este último predicado no es recursivo, y su éxito depende del éxito de otros dos predicados que sí lo son. Su definición es la siguiente:

```
clc_en_dom(Rec,Durac,Doms,[Rec|Restante],[rec(Rec,X)|DomAux]):-
                          borra(rec(Rec,Y),Doms,DomAux),
                          saca(Y,Durac,[],Restante,X).
```

El predicado borra (anteriormente citadó) se usa aquí de modo que sirve para dejar aparte los recursos que no están involucrados y también nos permite conocer el tiempo de disponibilidad del recurso solicitado. El segundo predicado se llama saca y recibe como parámetros el tiempo disponible del recurso involucrado por la actividad, la duración de la propia actividad, una lista vacía que servirá como acumulador y dos variables, la primera para unificar los recursos (el tiempo) que consumió la actividad, y la segunda para regresarle al recurso el segmento de tiempo que no fue consumido por la actividad.
El predicado saca quitará, en cada paso de la recursión, la cabeza tanto a la lista de tiempo disponible del recurso como a la lista de duración de la actividad, e incluye la cabeza del recurso como cabeza del acumulador. El

caso base se da cuando la lista de duración de la actividad ya está vacía, en ese momento se copia en una de las variables (la primera) el contenido del acumulador que ahora tiene la parte de tiempo disponible del recurso consumida por la actividad, y en la última variable se copia el segmento de tiempo de recurso no consumido por la actividad:

```
saca(X,[],Restante,Restante,X).
```

En el caso recursivo, como ya se mencionó, se recortarán simultáneamente tanto la lista de recursos como la lista de duración de la actividad, pero los elementos suprimidos al recurso se irán integrando en el acumulador al ejecutarse la llamada recursiva:

```
saca([A|B],[C|D],Rest,Rsp,X) :- saca(B,D,[A|Rest],Rsp,X).
```

Si el tiempo disponible del recurso se agota antes que la duración de la actividad el predicado `saca` fracasará, haciendo a su vez fracasar a `clc_en_dom`, y ello obligará a `programa` a retroceder para buscar otra actividad. Si la duración de la actividad se agota antes que el tiempo disponible del recurso, nos encontraremos en el caso base.
Si `clc_en_dom` tiene éxito, el predicado `programa` se llamará recursivamente con los siguientes argumentos: la lista de actividades restantes luego que una de ellas fue borrada, la lista de recursos actualizada luego que `clc_en_doms` la modificó y actualizó, la actividad que fue borrada con el segmento de tiempo de recurso que utilizó, pegada al acumulador, y la variable para respuesta. El programa completo se incluye en el apéndice con el nombre de **programa no. 4.**
Ahora ya tenemos el programa que ordena las actividades según su precedencia y el programa que distribuye los recursos entre las actividades. Con ellos procederemos a generar el programa que resuelva nuestro problema de calendarización.

Integración de subproblemas

Es importante observar que los dos programas generados para resolver cada uno de los subproblemas planteados son muy similares entre sí; de hecho comparten algunos predicados. Debemos recordar también que para dar solución a nuestro problema de calendarización, es necesario generar un solo programa que involucre las restricciones contempladas por uno y otro programa.

Podemos unir ambos programas y obtener la emulación de CHIP para nuestro problema de calendarización. Para ello aprovecharemos la similitud entre dichos programas para obtener uno solo. Al observar ambos programas encontraremos que, por su forma, el primero de ellos se aproxima a un fragmento del segundo. La diferencia más notable es el predicado `subconj` y su predicado auxiliar `miembro`. Bajo estas circunstancias podemos agregar dichos predicados al segundo programa para obtener el que nos interesa. El problema ahora es dónde colocar dichos predicados dentro del segundo. Para dilucidarlo retomemos nuestro análisis desde el inicio.
En ambos programas el predicado `pert` consulta al predicado `todas`, el segundo programa consulta, además, al predicado `dominios`, y ambos consultan luego a un predicado `programa`, con la diferencia de que el segundo programa incluye en su llamado a una variable más, la que ha resultado del llamado a `dominios`. Podemos dejar estas condiciones tal como aparecen en el segundo programa sin que se afecte al primero. Luego, en ambos casos el predicado `programa` llama al predicado `borra` con el mismo argumento, e igualmente se consulta a la base de datos con el predicado `actividad` y el mismo argumento instanciado por `borra`, acto seguido cada programa llama a un predicado para que proporcione diferente información, refirámonos brevemente a ambos predicados. Aquí debemos resaltar que las transformaciones que cada uno efectúa y la información que aportan son totalmente excluyentes, por lo que es factible hacerlos intervenir uno tras otro, incluso sin importar el orden, y la llamada recursiva a `programa` puede permanecer sin cambios. Ahora tenemos un programa para resolver nuestro problema de calendarización, emulando el funcionamiento de CHIP. Dicho programa aparece en el apéndice bajo el nombre de **programa no. 5.**
Ahora nos interesa generar un programa que resuelva el mismo problema pero sin emular a CHIP.

3.2.2 PROLOG y un problema de calendarización

Para resolver el problema de calendarización planteado en este mismo capítulo a través de PROLOG, consideraremos el programa generado para emular a CHIP y lo modificaremos. La modificación consistirá en representar la duración de actividades y recursos a través de intervalos en lugar de listas.
La base de datos usada por la emulación de CHIP proporciona a actividades y recursos una lista de números naturales para representar con ella los tiem-

pos de duración (para actividades) o de disponibilidad (para recursos).
Ahora en lugar de enumerar (de modo discreto) los elementos de la duración de cada actividad o recurso, utilizaremos un término para representar su duración a través de un intervalo (continuo). El símbolo funcional (o functor) del término que utilizaremos será `i`, y sus argumentos serán dos enteros positivos, el inicio y el final del intervalo. Así, la lista de elementos

```
[1,2,3]
```

será representado por el intervalo

```
i(0,3)
```

Es importante notar que hasta este momento hemos trabajado con valores discretos, por ejemplo, en la lista `[1,2,3]` se ve con claridad que contiene tres elementos (estas listas no representan intervalos). Ahora trabajaremos con valores continuos, y la lista mencionada será representada con el intervalo `i(0,3)` el cual puede dividirse en tres segmentos, el primero `i(0,1)`, el segundo `i(1,2)` y el tercero `i(2,3)`. Es válido preguntarse ¿Estos intervalos son abiertos (excluyen sus extremos) o son cerrados (incluyen sus extremos)? La respuesta es que son cerrados. Si no lo fueran ¿Cómo podríamos representar un punto? Nuestra base de datos también se modificará para que sea aceptada por el nuevo programa. Esta base se encuentra en el apéndice con el nombre de base no. 2.
Esta base de datos está lista para ser operada por nuestro programa en PROLOG. Puesto que el programa generado para emular a CHIP opera básicamente discriminando actividades y asociándolas con recursos, y sólo una pequeña parte del programa se concreta a operar directamente con los intervalos, bastará con modificar esa parte para tener el nuevo programa.
En nuestra emulación de CHIP, el predicado que se encarga de la manipulación de intervalos es `saca`, y lo hace a través de `clc_en_dom`.
Sus primeros tres argumentos sirven como datos para la mencionanda manipulación, y sólo el tercero de ellos requiere de un tratamiento intermedio, efectuado por `borra`. El cuarto argumento, una lista, unifica en su cabeza el nombre del recurso utilizado por la actividad, y en su resto la porción del intervalo requerido por la actividad. Puesto que ahora no representaremos los intervalos con listas sino con términos, sustituiremos la lista por la pareja de datos involucrada. El último argumento no requiere cambios, ya que la

única variable que en el programa anterior unificaba con una lista, puede igualmente unificar con un intervalo. Cambiaremos entonces los argumentos de `clc_en_dom`:

```
clc_en_dom(Rec,Durac,Doms,[Rec|Restante],
                          [rec(Rec,X)|Dom_Aux]) ...
```

por

```
clc_en_dom(Rec,Durac,Doms,(Rec,Restante),
                          [rec(Rec,X)|Dom_Aux]) ...
```

El predicado `borra` queda tal cual y el predicado `saca` debe reescribirse por completo. Originalmente, este predicado manipula listas, pero ahora queremos que manipule intervalos. El objetivo que se persiguió al definir `saca` de manera recursiva, era darle poder para manejar la totalidad de la lista que contiene la duración de la actividad; como ahora no usamos listas, redefiniremos `saca` como un predicado no recursivo, vamos a prescindir del acumulador, y la nueva definición tendrá sólo cuatro argumentos. La definición de `saca` que ya fue elaborada toma dos listas, y su modo de operación es análogo a tomar ambas listas, calcular su cardinalidad y restar la duración de la actividad de la duración del recurso. La nueva definición de `saca` hará algo similar. Su primer argumento sigue siendo una parte o todo el recurso, el segundo argumento es la duración de la actividad, el tercer argumento es la parte del recurso que le corresponde a la actividad, y el último es la parte restante de la actividad. Para ejecutar dicho cálculo, primero se resta el límite inferior del límite superior de la actividad (para conocer su duración), luego este resultado se suma al límite inferior del intervalo del recurso, para obtener el alcance de la actividad sobre el recurso. El segmento proporcionado a la actividad irá desde el límite inferior del recurso hasta el alcance mencionado, y la parte restante del recurso irá desde dicho alcance hasta el límite superior del recurso:

```
saca(i(A,B),i(C,D),i(A,F),i(F,D)) :- F is A + D - C.
```

con esto damos por terminada la transformación. El programa completo aparece en el apéndice con el nombre de **programa no. 6**.
La única diferencia entre los dos últimos programas es que la emulación de CHIP opera enumerando los elementos de los valores numéricos mientras

que el segundo programa manipula los valores numéricos como intervalos. En caso de valores muy grandes la diferencia en el tiempo de ejecución entre ambos puede ser igualmente grande. A continuación atacaremos el mismo problema pero con un modelo de ejecución distinto a los que ya vimos. Este modelo se conoce como aritmética de intervalos.

3.2.3 Aritmética de intervalos y un problema de calendarización

La aritmética de intervalos [1] se generó con la finalidad de buscar una convergencia entre la manipulación numérica de punto flotante y el cómputo como deducción. Un resultado importante de la aritmética de intervalos es la eliminación de los errores por redondeo. Con este fin, las operaciones aritméticas son tratadas como restricciones y cada operando en lugar de representarse con un número se representa con un intervalo, es decir con un par de números, el primero de ellos necesariamente menor o igual que el segundo. Este par de números son los valores representables en la máquina más cercanos al valor del operando. El primero de ellos es menor o igual que el operando y el segundo es mayor o igual que el operando. Antes de pasar a la implantación veremos en que consiste el método para conocer su mecánica.

El método

Las operaciones aritméticas sobre intervalos están orientadas a reducir intervalos, siempre que esto se pueda hacer. La reducción de un intervalo se logra disminuyendo su longitud, es decir aumentando el valor de su límite inferior y disminuyendo el valor de su límite superior, tanto como sea posible. Siempre se procura que el valor búscado se encuentre entre ambos límites (dentro del intervalo). El método no sólo reduce los intervalos de los resultados en las operaciones aritméticas sino también los intervalos de los operandos. Con este fin, el método recibe una operación y los parámetros de dicha operación. Supongamos que la operación en cuestión es la suma, y que sus parámetros son los sumandos X y Y y la suma Z, lo que expresaremos como $suma(X, Y, Z)$. Al conjunto de intervalos $\{X, Y, Z\}$ le llamaremos vector de entrada. Para reducir los intervalos seguiremos el siguiente método (después de la descripción ilustraremos con un ejemplo, descrito por Cleary [1]):

1° Recalcular cada intervalo del vector de entrada a través de los intervalos restantes y del uso (de la definición matemática) de la operación. Así generaremos un nuevo vector al que le llamaremos vector intermedio.
2° Obtener la intersección entre cada uno de los intervalos del vector de entrada y su homólogo en el vector intermedio. Si alguna de las intersecciones es el conjunto vacío, todo el proceso de reducción fracasa y decimos que el vector (de entrada) bajo esa operación es *inconsistente*. Al nuevo vector le llamaremos vector de intersecciones.
3° Calcular la función de redondeo externo para cada intervalo del vector de intersecciones. Este último vector será el vector de salida.
4° Si el vector de salida no es igual al vector de entrada, repetir el proceso, tomando ahora el vector de salida como vector de entrada. En este último caso diremos que el vector de entrada bajo esa operación es inestable. Si el vector de entrada es igual al vector de salida el proceso termina con éxito y decimos que el vector es estable bajo esa operación.
Para ilustrar el método daremos valores a los parámetros X, Y, y Z de la suma. Sean $X = [0,2]$, $Y = [1,3]$, y $Z = [4,6]$. El primer paso solicita recalcular cada intervalo, para ello usaremos la definición de suma:

$$Z' = X + Y$$

$$Z' = \{z : \exists\ x \epsilon X,\ y \epsilon Y,\ x + y = z\}$$

$$X' = Z - Y$$

$$X' = \{x : \exists\ z \epsilon Z,\ y \epsilon Y,\ z - y = x\}$$

$$Y' = Z - X$$

$$Y' = \{y : \exists\ z \epsilon Z,\ x \epsilon X,\ z - x = y\}$$

El nuevo intervalo Z' es el resultado de sumar cada uno de los puntos del intervalo en X con cada uno de los puntos del intervalo en Y. De modo análogo, X' es el resultado de restar cada punto en Y de cada punto en Z, y Y' resulta de restar cada punto en X de cada punto en Z. Podemos ver que los límites del intervalo Z' son $[0+1, 2+3]$, los del intervalo X' son $[4-2, 6-1]$ y los del intervalo Y' son $[4-2, 6-0]$. El segundo paso solicita encontrar las intersecciones entre X y X', entre Y y Y' y entre Z y Z'. Para

$X \cap X'$ tenemos $[0,2] \cap [1.5] = [1,2]$, para $Y \cap Y'$ tenemos $[1,3] \cap [2,6] = [2,3]$, y para $Z \cap Z'$ tenemos $[4,6] \cap [1,5] = [4,5]$. A continuación se debe aplicar una función de redondeo externo a cada uno de los intervalos del vector de intersecciones. Dicha función consiste en decrementar al límite inferior hasta el valor más próximo representable en la máquina, e incrementar el límite superior hasta el más próximo valor representable por la máquina. En nuestro caso la aplicación de esta función no es necesaria dado que trabajamos con números enteros. Puesto que el vector de salida $< X', Y', Z' >$ es diferente al vector de entrada $< X, Y, Z >$, repetiremos el proceso:

1° $X = [1,2]$, $Y = [2,3]$, $Z = [4,5]$
$X' = [4-3,5-2]$, $Y' = [4-2,5-1]$, $Z' = [1+2,2+3]$
2° $X \cap X' = [1,2] \cap [1,3] = [1,2]$
$Y \cap Y' = [2,3] \cap [2,4] = [2,3]$
$Z \cap Z' = [4,5] \cap [3,5] = [4,5]$

Desde este punto podemos advertir que el vector de salida será igual al vector de entrada. Con ello la tarea ha terminado.
Al proceso descrito le llamaremos reducción de intervalos, y al conjunto del vector y la operación asociada le llamaremos restricción.
En la Aritmética de Intervalos existe un conjunto mínimo de operaciones, y de dicho conjunto pueden derivarse todas las demás operaciones aceptables. A este conjunto se la ha llamado "operaciones primitivas" [1].
Las operaciones que nosotros utilizaremos en este trabajo son *suma*, *menor o igual y minimiza*.
Debe mencionarse que nosotros aprovecharemos las posibilidades de manejo de intervalos que este método brinda, aunque sólo utilizaremos valores enteros.
El proceso descrito reduce una restricción individual, pero nuestro caso es el de un conjunto de ellas, es decir, es una red de restricciones. En ésta existen restricciones (relaciones) que comparten intervalos entre sus parámetros, de modo que al reducir el conjunto de intervalos de una relación, las relaciones que compartan uno o varios de estos intervalos pueden pasar de ser estables a ser inestables, y en consecuencia debe volver a reducirse hasta ser nuevamente estable. Uno de los problemas que podemos encontrar es la posibilidad de ejecutar muchas reducciones innecesarias. Para el tratamiento de las redes de restricciones Cleary proporciona un algoritmo [1], conocido como algoritmo de relajación, el cual usa dos listas, la primera de ellas contiene la totalidad

de las restricciones, y la segunda se encuentra vacía al iniciar el proceso.
Toma una restricción de la lista original, la reduce hasta que sea estable y la mete en la segunda lista, cuando la segunda lista no se encuentra vacía, cada vez que reduce una restricción, busca si en la segunda lista existe alguna restricción que comparta algún intervalo, y todas las que encuentre, serán reintegradas en la primera lista para que se reduzcan de nuevo. Así continúa hasta que todas las restricciones sean estables y se encuentren en la segunda lista.
Los requerimientos del algoritmo de relajación son dos listas A y P, y la definición de parejas de variables (p,$\vec{V}$), donde 'p' representa a la relación, y $\vec{V}$ representa al vector de intervalos que parametrizan a 'p'.
El algoritmo es el siguiente:

1.- Mete en A todas las restricciones de la red
2.- Inicializa P como la lista vacía
3.- Mientras A no esté vacía:
Empieza
4.- Saca una restricción (p,$\vec{V}$) de A
5.- Aplica reducción de intervalos a (p,$\vec{V}$) para obtener $\vec{V}'$
6.- Si la reducción fracasa termina con error
7.- Si_No Si $\vec{V} \neq \vec{V}'$:
Empieza
8.- $\vec{V} \leftarrow \vec{V}'$
9.- Para cada restricción (q,$\vec{Y}$) en P:
10.- Si $\vec{V}$ y $\vec{Y}$ comparten variables:
11.- Saca (q,$\vec{Y}$) de P y mételo en A
Termina
12.-Pega (p,$\vec{V}$) al final de P
Termina.

El programa que nosotros generaremos para resolver el problema de calendarización planteado deberá ser capaz de reducir intervalos, y manejar una red de estos.

La implantación

Cabe mencionar que nuestra base de datos no. 2 se adapta de manera natural a las exigencias de la aritmética de intervalos, ya que la notación corresponde

a intervalos con un límite inferior y un límite superior. Nuestra implantación del algoritmo de relajación requerirá algunas adecuaciones a PROLOG. En esta tarea específicamente, es deseable tener un metaintérprete que se encargue del algoritmo de relajación y hacer fuera de dicho metaintérprete lo que no corresponda al mismo.
Nuestro programa transformará la base de datos, para que la manipulación de ésta por el algoritmo de relajación sea adecuada.
La idea intuitiva del funcionamiento del programa es la siguiente:
Inicialmente se consulta la base de datos y se reescribe cada una de las actividades en un formato adecuado para ser manipulada por el algoritmo de relajación, luego las actividades se agrupan en conjuntos de elementos que utilizan el mismo recurso.
Hasta aquí el proceso se ha limitado a preparar la base de datos para ser recibida por el algoritmo de relajación. En seguida se elige una actividad cuyos antecedentes ya hayan sido procesados (al iniciar, la única actividad factible es la de inicio ya que no tiene antecedentes) y se elige de modo no determinístico una de las permutaciones del conjunto en que se encuentre dicha actividad y esta permutación se pasa al algoritmo de relajación. El proceso de elegir una actividad cuyos antecedentes se satisfagan, seleccionar una permutación y relajar, se repite hasta agotar la totalidad de las actividades.
Debe mencionarse aquí que la base de datos, además de ser transformada por el programa, requiere otra modificación previa, la que consistente en sustituir el recurso de cada actividad por una lista de recursos, de modo que las actividades con más de un recurso puedan optar por el uso de uno u otro recurso. La modificación se hará en todas las actividades aún cuando exista sólo un recurso; de este modo, donde decía:

```
actividad(Nombre,i(Inicio,Fin),[Antecedentes],Recurso).
```

ahora dirá:

```
actividad(Nombre,i(Inicio,Fin),[Antecedentes],[Recursos]).
```

Por este motivo requeriremos un programa más general que los anteriormente descritos.
La implantación del programa se describe a continuación.
El predicado `ejecuta` instanciará, en su única variable, el resultado de la evaluación:

```
ejecuta(M) ...
```

Inicialmente este predicado solicitará una consulta a la base de datos:

```
ejecuta(M) :- dominios(Q), todas(L), ...
```

Con esta información, se reescribirán las actividades de modo que la los datos contenidos en cada actividad sean suficientes, para posteriormente asignarle los recursos y para que el algoritmo de relajación pueda, de manera adecuada, operar con dicha actividad. La reescritura corre a cargo del predicado **reesc**, y tiene tres argumentos: la lista de recursos, la lista de todas las actividades, y una variable en que regresará la lista reescrita de actividades:

```
ejecuta(M) :- dominios(Q), todas(L), reesc(Q,L,L1), ...
```

Cabe mencionar que luego del éxito de este predicado nunca más se consultará la base de datos. La transformación de reescritura cambia el contenido de los datos de cada actividad, del formato antes presentado:

```
actividad(Nombre,i(Inicio,Fin),[Antecedentes],[Recursos]).
```

al siguiente:

```
act(Nombre, Recurso, si(i(Inicio,Fin),
                        d(Duracion,Duracion),
                        f(Inicio,Fin)), [Antecedentes])
```

En contraste con la primera fórmula, en la que **actividad** es el símbolo de predicado de un átomo, en la segunda fórmula será considerada como un término, del que **act** es el símbolo funcional (o functor). Los términos de este tipo se usarán para construir listas de actividades. Los argumentos del término cuyo functor es **act** son, primero, el nombre de la actividad, segundo, un solo recurso de la lista de recursos original de la actividad, tercero, un término cuyo functor es **si**, que a su vez contiene tres términos, que representan, cada uno, los intervalos de inicio, duración, y final, y cuarto, la lista de antecedentes propios de la actividad. El predicado **reesc** opera recursivamente recorriendo la lista de todas las actividades (esta última proporcionada por el predicado **todas**), reescribiendo e integrando cada una de ellas a la lista indicada por el tercer argumento. Cuando la lista de actividades se agota, el predicado llega al fondo de la recursión y en ese caso no hay transformación por lo que únicamente regresa una lista vacía:

```
reesc(_,[],[]).
```

En el caso recursivo, este predicado unifica, en la tercera variable, diferentes datos que obtiene a lo largo de su transformación y que van encontrando su lugar en la estructura proporcionada por la misma variable:

```
reesc(Q,[A|T],[act(A,Rec,si(i(Y,Z),d(X,X),f(Y,Z)),P)|L]) ...
```

El nombre de la actividad lo obtiene de manera inmediata a través de la lista de actividades, el nombre del recurso lo obtiene de manera no determinística al borrarlo, con el predicado borra, de la lista original de recursos propios de la actividad:

```
reesc(Q,[A|T],[act(A,Rec,si(i(Y,Z),d(X,X),f(Y,Z)),P)|L]) :-
                actividad(A,i(_,X),P,Rs), borra(Rec,Rs,_),...
```

esta lista de recursos se obtuvo a través de la consulta al predicado actividad correspondiente, el predicado borra es el que ha venido utilizándose en los programas anteriores, y es precisamente aquí donde reside la importancia del predicado reesc. Recordemos que el predicado borra proporciona respuestas no determinísticas y lo hace tantas veces como elementos pueda borrar de la lista. Esto en nuestro programa se traduce en la posibilidad de que cada actividad opte por el uso de un recurso, luego otro, y así sucesivamente hasta llegar al último. El tercer argumento del término con functor act es a su vez un término con functor si cuyos tres argumentos son a su vez términos que representan intervalos, estos son:

```
si(i(Y,Z),d(X,X),f(Y,Z))
```

el primer intervalo, de inicio, expresa la porción de recurso en que la actividad puede empezar, el segundo intervalo expresa la duración de la actividad, y el tercero expresa la porción de recurso en que la actividad puede terminar. Cabe notar que el segundo intervalo es un punto, esto es, la duración de la actividad no está sujeta a restricción alguna y no hay modo de alterarla, mientras que el primer intervalo y el tercero son iguales, y su extensión es, al iniciar el programa, tan grande como el recurso que la actividad requiere, según podemos apreciar en el siguiente segmento de programa:

```
reesc(Q,[A|T],[act(A,Rec,si(i(Y,Z),d(X,X),f(Y,Z)),P)|L]) :-
                actividad(A,i(_,X),P,Rs), borra(Rec,Rs,_),
                miembro(rec(Rec,i(Y,Z)),Q), ...
```

Debido a al forma de su primer argumento, el predicado `miembro`, que ya conocemos, busca la duración del recurso dentro de la lista de recursos, la cual es el primer argumento de `reesc`. La utilidad que se deriva de copiar la duración del recurso en su totalidad en el principio y fin de cada actividad que use dicho recurso, consiste en proporcionar al algoritmo de relajación la libertad suficiente para dar a cada actividad el segmento de recurso más adecuado.
Finalmente el predicado `reesc` se llama recursivamente con la misma lista de recursos `Q`, el resto de la lista con nombres de actividades, y el resto de la lista de actividades por reescribir:

```
reesc(Q,[A|T],[act(A,Rec,si(i(Y,Z),d(X,X),f(Y,Z)),P)|L]) :-
                actividad(A,i(_,X),P,Rs), borra(Rec,Rs,_),
                miembro(rec(Rec,i(Y,Z)),Q), reesc(Q,T,L).
```

De regreso en nuestro predicado `ejecuta`, una vez que se ha cambiado la estructura de la información incluida en las actividades, se debe proceder a agrupar en sublistas a las actividades que compartan los mismos recursos. Es decir, de la lista original de actividades entre las que se distribuyen 'n' recursos, generaremos 'n' sublistas de actividades, y en cada sublista todas las actividades compartirán el mismo recurso, de modo que no habrá sublistas vacías ni habrá dos o más sublistas con elementos en común. La finalidad de esta reagrupación es proporcionar al algoritmo de relajación bloques de actividades que compartan el mismo recurso. Esta tarea se pone en marcha a través del predicado `separa` cuyos argumentos son la lista de actividades reescritas, y una variable en la que unificará la reagrupación de dichas actividades. El predicado `separa` recorre toda la lista de actividades reescritas, por lo que se encuentra definido de manera recursiva. En cada paso de la recursión busca una sublista adecuada para incluir a cada actividad. La recursión toca fondo cuando la lista se agota, en cuyo caso regresa la lista vacía:

```
separa([],[]).
```

En su cláusula recursiva, `separa` hace uso del predicado `acomoda` que es a su vez recursivo y se declara a través de tres cláusulas. El predicado `acomoda` toma como primer parámetro una actividad y la mete en una sublista de actividades que comparten el mismo recurso. Si no existe una sublista

que contenga actividades con el mismo recurso de la actividad en cuestión, dicha actividad se integra en una nueva sublista. Esto último ocurre cuando el predicado ha recorrido todas las sublistas buscando el recurso sin éxito. Definiendo el caso base de la recursión:

```
acomoda(H,[],[[H]]).
```

El argumento en que el predicado acomoda hace la búsqueda es el segundo, si la busqueda no tiene éxito, esta continúa hasta tener éxito o hasta encontrar el caso base:

```
acomoda(act(A,B,C,P),[[act(D,E,F,P1)|T]|Ts],
                     [[act(D,E,F,P1)|T]|L]) :-
                        B \= E,  acomoda(act(A,B,C,P),Ts,L).
```

Cuando la búsqueda tiene éxito, la actividad se integra en la lista correspondiente y la recursión se detiene:

```
acomoda(act(A,B,C,P),[[act(D,B,E,P1)|T]|Ts],
                     [[act(A,B,C,P),act(D,B,F,P1)|T]|Ts]).
```

Con auxilio del predicado acomoda, el predicado separa podrá revisar cada actividad de la lista original, y en cada paso de recursión incluir dicha actividad en la sublista correspondiente:

```
separa([H|T],L) :- separa(T,L1), acomoda(H,L1,L).
```

Una vez que hemos reescrito las actividades, y que las hemos agrupado en función de los recursos que utilizan, podemos proceder a transformar según el algoritmo de relajación. Para ello utilizaremos el predicado ejec_aux con cuatro parámetros, el primero de ellos un acumulador, el segundo la lista de listas proporcionada por separa, el tercero es otro acumulador, y el último es una variable con la que unificará el resultado final, esta variable es la misma que parametriza al predicado ejecuta:

```
ejecuta(M) :- dominios(Q), todas(L), reesc(Q,L,L1),
              separa(L1,L2), ejec_aux([],L2,[],M).
```

La tarea de ejec_aux es recorrer la lista de sublistas de actividades, seleccionar un conjunto cuyas condiciones de precedencia estén satisfechas y le

permitan reducirlas a continuación. Una vez encontrado el conjunto calcula una permutación del conjunto, la relaja y la mete en el segundo acumulador y finalmente se llama de modo recursivo hasta agotar la lista de sublistas, en cuyo caso copia el segundo acumulador en la variable que ha sido reservada para unificar la respuesta, terminando así la recursión. La definición de `ejec_aux` es:

```
ejec_aux(_,[],R,R).
ejec_aux(I,L,N,R) :- selecciona(I,L,K,M), perm(K,H),
                     relaja(I,I1,H,[],J), pega(J,N,O),
                     ejec_aux(I1,M,O,R).
```

El primer paso de la cláusula recursiva, encargada de revisar las condiciones de precedencia, se verifica a través del predicado `selecciona`, cuyos argumentos son: el primer acumulador, que en un principio está vacío, la lista de sublistas de actividades, y dos variables, la primera para unificar la sublista cuyos antecedentes se satisfagan, y la segunda para instanciar a todas las sublistas restantes. Este predicado está definido de manera recursiva y opera de la siguiente manera. En el caso base, la recursión toca fondo cuando la cabeza del segundo argumento, que es una lista de actividades, contiene actividades cuyos antecedentes están incluídos en el primer argumento, lo que se verifica a través del predicado `revisa` que analizaremos un poco más adelante. Si esta definición tiene éxito, el siguiente predicado es un símbolo de corte, lo cual motivará que, una vez encontrado un conjunto que satisfaga la restricción de precedencia, no se buscará más:

```
selecciona(I,[L|Ls],L,Ls):- revisa(I,L), !.
```

En el caso en que `revisa` no tenga éxito, se aplicará la cláusula recursiva del predicado `selecciona`, la cual discrimina en su búsqueda a la cabeza (que ya fue considerada), y continúa buscando:

```
selecciona(I,[L|Ls],K,[L|M]) :- selecciona(I,Ls,K,M).
```

En el caso en que el predicado `selecciona` continúe su búsqueda sin éxito hasta vaciar el segundo argumento, esto motivará un retroceso en el predicado `ejec_aux`.
El predicado `revisa`, que es utilizado por `selecciona`, presenta una definición

recursiva, su tarea es verificar si en la lista I que recibe como primer argumento (es una lista de nombres de actividades), se encuentran contenidas las listas de antecedentes de cada actividad que recibe en la lista del segundo argumento. Puesto que revisa la totalidad de las actividades, el caso base para la recursión se da cuando recibe vacía la lista de actividades:

```
revisa(I,[]).
```

En su caso recursivo, el predicado `revisa` llama a ejecución al predicado `subconj` que ya conocemos, con la lista de antecedentes de la actividad en curso y la lista (la misma lista I) de actividades ya procesadas, para que verifique si el primero es subconjunto del segundo:

```
revisa(I,[act(N,R,S,P)|L]) :- subconj(P,I), revisa(I,L).
```

Toda vez que el predicado `selecciona` proporciona en su tercer argumento una lista de actividades que comparten el mismo recurso y cuyos requisitos de precedencia ya están satisfechos, podemos tomar dicha lista y dársela al metaintérprete para que le aplique el algoritmo de relajación. Pero antes de ello, se aplica una transformación más a dicha lista. Esta transformación consiste en calcular de modo no determinístico todas las posibles permutaciones. Al generar las permutaciones se garantiza que una actividad tome diferentes segmentos de un recurso, pero sin que se repita ninguno. Ahora podemos referirnos a la utilidad de agrupar en sublistas la totalidad de las actividades. Si enviásemos todas las actividades al metaintérprete que implanta al algoritmo de relajación, este último invertiría más tiempo buscando las actividades que comparten su recurso con la última actividad reducida, pero más costoso aún, en tiempo de ejecución, sería calcular las permutaciones de las n actividades ya que al repartirlas en k sublistas m, tales que $|m_1| + |m_2| + \ldots + |m_k| = n$, al calcular las permutaciones de cada sublista m, el total de permutaciones generadas es menor que el número de permutaciones que pueden generarse a partir de la lista completa, esto es:

$$(|m_1|!) * (|m_2|!) * \ldots * (|m_k|!) \leq n!$$

Al calcular las permutaciones de cada sublista, garantizamos que el conjunto de respuestas proporcionadas por PROLOG sea completo y además no redundante.
Las permutaciones se generan a través del predicado `perm` cuya definición

es recursiva y opera del siguiente modo: perm tiene dos parámetros, en el primero recibe instanciada la lista sobre la que hará las permutaciones, y el segundo es una variable en la que unificará cada permutación calculada. perm toma la cabeza de su primer argumento (una lista) y calcula recursivamente cada una de las permutaciones del resto. El multicitado borra aquí es usado de manera ingeniosa para realizar la operación inversa a "borrar" (aprovechando las virtudes del algoritmo de unificación), pues al recibir instanciados su primero y su tercer argumento, en el segundo argumento construye cada una de las listas resultantes de insertar el elemento del primer argumento en la lista del tercer aregumento, que no son sino las permutaciones de la lista original. En cada llamada recursiva, perm va eliminando la cabeza de la lista que recibe por lo que el caso base recibe la lista vacía, cuya permutación es a su vez la lista vacía:

```
perm([],[]).
```

En el modo recursivo, como ya se mencionó, calcula cada una de las permutaciónes del resto de la lista que recibe instanciada, les inserta la cabeza, a través de borra, en todas las posiciones posibles:

```
perm([X|Xs],Y) :- perm(Xs,Z), borra(X,Y,Z).
```

Puesto que borra proporciona respuestas no determinísticas, perm igualmente ofrecerá respuestas no determinísticas.
Conforme perm calcula cada permutación de la sublista en turno, dicha permutación es recibida, entre otros parámetros, por el predicado relaja cuya tarea es proporcionar el mecanismo de control utilizado por el algoritmo de relajación.
El predicado relaja recibe cinco argumentos, el primero de ellos, I, es el mismo acumulador que recibió como primer argumento ejec_aux, en éste se almacenan los nombres de las actividades cuyos intervalos ya fueron reducidos y han alcanzado su estabilidad. El segundo argumento es una variable que unifica con una lista de los nombres de todas las actividades que ya habían sido reducidas, más las que se agregarán a la lista luego de que relaja termine su ejecución. El tercer argumento es una de las permutaciones descritas en los parrafos anteriores, o sea una lista que contiene todas las actividades que comparten algún recurso, el cuarto es un acumulador, en donde se van introduciendo las actividades luego de que han sido reducidas, y el quinto es

una variable donde unificará la lista de actividades que comparten un recurso cuando éstas han alcanzado su estabilidad. El modo en que opera `relaja` es el siguiente: De la lista que recibe como tercer argumento, reduce todos los intervalos de actividades contenidos en ésta, hasta que todas son estables, es decir hasta que todas las actividades han abandonado dicha lista, y entran, reducidas y estables, al acumulador (cuarto argumento). Entonces la lista del primer argumento, que ha estado creciendo, se copia en el segundo argumento, y el acumulador, con todas las actividades, se copia en la última variable del predicado:

```
relaja(S,S,[],M,M).
```

Mientras este caso no se de, el predicado `relaja` toma las dos primeras actividades inestables (del tercer argumento) y las envía al metaintérprete `goal` para que ejecute una reducción con la primera, y el primer paso de reducción con la segunda, si los intervalos de la primera actividad son estables, entonces esta última se integra al acumulador (cuarto argumento), el nombre de esta actividad se integra también en la lista de nombres de actividades (primer argumento), y luego se hace la llamada recursiva:

```
relaja(S,S1,[act(N1,R,A,P),act(N2,R,B,P1)|C],D,M) :-
  goal([ rest(sum,R,A,_,[]),
         rest(sum,R,B,si(_,Y,Z),_),
         rest(me,R,_,si(_,X),[]),
         rest(min,R,_,A1,[]), _] ),
  relaja([N1|S],S1,[act(N2,R,si(X,Y,Z),P1)|C],
                   [act(N1,R,A1,P)|D],M).
```

Pero si esta primera actividad no resulta estable, entonces el predicado `traslada`, saca de la lista de actividades estables a aquellas que compartan el mismo recurso con la que resulto inestable, luego se hace una llamada recursiva integrando la actividad reducida en la lista de actividades estables, y el nombre de tal actividad se integra en la lista respectiva (primer argumento):

```
relaja(S,S1,[act(N1,R,A,P),act(N2,R,B,P1)|C],D,M) :-
       goal([ rest(sum,R,A,_,_),
              rest(add,R,B,si(_,Y,Z),_),
              rest(me,R,_,si(_,X),_),
```

```
            rest(min,R,_,A1,_),
            parcial(_,_,_,[H|T]) ] ),
        traslada(H,[act(N2,R,si(X,Y,Z),P1)|C],D,E,F),
        relaja([N1|S],S1,E,[act(N1,R,A1,P)|F],M).
```

En un tercer tercer caso se contempla la posibilidad de que sólo reste una actividad en la lista de actividades inestables, en cuyo caso, se reduce utilizando dos de los predicados del metainterprete, ya reducida se integra al acumulador, y se hace una llamada a la cláusula básica de la definición de relaja incorporando a la vez el nombre de la actividad en la lista respectiva (primer argumento):

```
relaja(S,S1,[act(N,R,A,P)],C,M) :- rest(sum,R,A,A1,_),
       rest(min,R,A1,A2,_), pega(C,[act(N,R,A2,P)],D),
       relaja([N|S],S1,[],D,M).
```

El metaintérprete goal en su argumento espera una lista de términos estructurada en una forma especial para que, previos ajustes realizados mediante la invocación al predicado red, dichos términos puedan ser interpretados como predicados a través de la llamada a proceso:

```
goal(X) :- red(X), proceso(X).
red([ rest(sum,Rec,si(X1,Y1,Z1),si(X1p,Y1p,Z1p),Q),
      rest(sum,Rec,si(X2,Y2,Z2),si(X2p,Y2p,Z2p),_),
      rest(me,Rec,si(Z1p,X2p),si(Z1q,X2q),S),
      rest(min,Rec,si(X1p,Y1p,Z1q),si(X1r,Y1r,Z1r),T),
      parcial(Q,S,T,U) ]).
```

El predicado proceso se define recursivamente. De la lista de términos que recibe como parámetro, toma el primero de ellos, y lo interpreta como condición para su propio éxito, interpretándolo para ello como predicado, luego se llama recursivamente con el resto de la lista hasta agotarla:

```
proceso([]).
proceso([X|Y]) :- X, proceso(Y).
```

El predicado red define una lista de cinco términos, los primeros cuatro con functor rest y el último con functor parcial, estos son los términos que el predicado proceso interpreta como átomos.

La definición del predicado rest consta de siete cláusulas e implanta las restricciones que transformarán a los intervalos de cada actividad. En cada caso, el primer argumento señala el nombre de la restricción que se está implantando, las cuales son suma (se denominó sum), menor o igual (se denominó me), y minimización (se llama min). En consecuencia, el predicado proceso dirigido por la estructura del predicado red dice lo siguiente: ejecuta reducción de intervalos según la restricción suma (sum) a ambas ternas de intervalos, acto seguido toma ambos resultados y calcula la reducción menor o igual con el intervalo de terminación de la primera terna y el intervalo de inicio de la segunda, finalmente incorpora el nuevo intervalo calculado para el final de la primera terna y restringe ésta al mínimo espacio posible de modo que se coloque, de principio a fin, en la parte más cercana a cero.
El predicado rest, bajo la restricción de suma tiene dos cláusulas, pero no es recursiva. La primera cláusula tiene éxito cuando la restricción es estable y sus intervalos ya no sufren cambios bajo dicha restricción. Este predicado tiene cinco parámetros, el primero es el nombre de la restricción, el segundo el nombre del recurso que utiliza la actividad a la que pertenece el intervalo, el tercero es la terna de intervalos original, el cuarto es la terna de intervalos donde se instancia la respuesta (el intervalo reducido), y el quinto es una lista que regresa vacía cuando la restricción es estable. Lo que hace el predicado es calcular el valor de las proyecciones de la restricción sobre cada coordenada y luego calcula la intersección de ésta con el valor del intervalo, si los valores de la reducción así calculada son iguales a los de la terna original, esta definición tiene éxito. En el caso en que la reducción calculada sea diferente a la original, entonces tiene éxito la segunda definición, la cual regresa en su último argumento el nombre del recurso, para denotar que el intervalo recibido es inestable y para proporcionar el nombre del recurso:

```
rest(sum,Rec,si(i(A,B),d(C,D),f(E,F)),
             si(i(M,N),d(C,D),f(Q,R)),[]) :-
     K is E - D, L is F - C, inter(i(A,B,i(K,L),i(M,N)),
     O is A + D, P is B + C, inter(i(E,F),i(O,P),i(Q,R)),
     [A,B,E,F] == [M,N,Q,R].
rest(sum,Rec,si(i(A,B),d(C,D),f(E,F)),
             si(i(M,N),d(C,D),f(Q,R)),[Rec]) :-
     K is E - D, L is F - C, inter(i(A,B,i(K,L),i(M,N)),
     O is A + D, P is B + C, inter(i(E,F),i(O,P),i(Q,R)),
```

```
[A,B,E,F] \= [M,N,Q,R].
```

El predicado `inter` calcula la intersección entre intervalos, se define a través de dos átomos sin recursión. El primero de ellos compara el límite superior del primer intervalo (primer argumento) con el límite inferior del segundo intervalo (segundo argumento), y regresa en el tercer intervalo el punto definido por ambos límites, en caso de que sean iguales:

```
inter(i(_,E),i(E,_),i(E,E)).
```

El otro caso considera la situación en que los intervalos no son contiguos y la intersección no es vacía. Para ello compara nuevamente el límite superior del primer intervalo con el límite inferior del segundo intervalo, si el primero es mayor que el segundo, entonces define los límites del nuevo intervalo a través del predicado `comp`:

```
inter(i(A,B),i(C,D),i(E,F)):-C < B,comp(A,C,E,_),comp(B,D,_,F).
```

El predicado `comp` se define en dos cláusulas de cuatro argumentos. Lo único que hace es cambiar el orden de sus dos primeros argumentos, regresandolos en los dos últimos, si el primer argumento es menor que el segundo:

```
comp(A,B,B,A) :- A < B.
```

y regresarlos sin cambio en caso de que el primero sea mayor o igual que el segundo:

```
comp(A,B,A,B) :- A >= B.
```

Por ello, el predicado `inter` consulta al predicado `comp` dos veces, la primera con los dos límites inferiores de los intervalos y de entre estos toma el que será el límite inferior del nuevo intervalo, y en la segunda llamada a `comp` envía como argumentos los límites superiores de los intervalos que recibe, y de entre estos selecciona el que será el límite superior del nuevo intervalo. Para el caso en que la intersección es vacía, el predicado `inter` fracasa.
Las siguientes tres cláusulas del predicado `rest` corresponden a la restricción menor o igual (`me`). En estos casos tampoco hay definiciones recursivas, y el predicado tiene también cinco argumentos, el primero de los cuales es el nombre de la restricción (`me`), el segundo es el nombre del recurso compartido por

las actividades involucradas, el tercero es un par de intervalos que el predicado recibe instanciado, el cuarto argumento es la estructura que contiene el par de intervalos que serán regresados, el último argumento es una lista que puede estar vacía o bien tener un elemento, el que de ser instanciado será el nombre del recurso. Del par de intervalos que recibe en el tercer argumento, el primero de ellos corresponde al intervalo final de una actividad, y el segundo es el intervalo inicial de otra actividad. Esta restricción puede tener éxito cuando ocurre uno de los siguientes tres casos: El primero, cuando la intersección entre ambos intervalos es un punto y el primer intervalo es mayor que el segundo, es decir que el límite inferior del primer intervalo sea igual al límite superior del segundo intervalo, lo cual motiva que ambos intervalos se reduzcan a dicho punto, lo que se expresa de la siguiente manera:

```
rest(me,Rec,si(f(E,F),i(M,N)),si(f(E,E),i(N,N)),[Rec]):- E = N.
```

La lista del quinto argumento, que contiene al recurso, expresa que los intervalos no eran estables al ser recibidos. El segundo caso se da cuando la intersección entre ambos intervalos es vacía o es un punto, pero el primer intervalo es menor que el segundo, en cuyo caso se asume que los argumentos recibidos son estables y no se reducen bajo esta restricción, por lo que se regresa una lista vacía en el quinto argumento:

```
rest(me,Rec,si(f(E,F),i(M,N)),si(f(E,F),i(M,N)),[]) :- F =< M.
```

El tercer caso se da cuando la intersección no es vacía y tampoco es un punto, la acción indicada en este caso es calcular la intersección e instanciar ambas reducciones con dicha intersección:

```
rest(me,Rec,si(f(E,F),i(M,N)),si(f(G,H),i(G,H)),[Rec]) :-
                E < N,  F > M,  inter(i(E,F),i(M,N),i(G,H)).
```

La última variable se instancia como una lista que contiene al recurso, lo cual hace suponer que los intervalos se recibieron siendo inestables. Las últimas dos cláusulas de definición del predicado `rest` implantan la restricción de minimización, la cual modifica un intervalo para reducirlo al mínimo posible. Esto se logra calculando nuevamente la restricción de suma para el caso en que el intervalo de inicio de la actividad sea un punto. El modo en que opera esta restricción es comparando los dos límites del intervalo de inicio, si estos son iguales la restricción se considera satisfecha y no opera ningún cambio:

```
rest(min,Rec,si(i(A,A),d(C,D),f(E,F)),
             si(i(A,A),d(C,D),f(E,F)),[]).
```

Si los límites son diferentes, se calcula la restricción de suma reduciendo a un punto el primer intervalo, y se copia el resultado de esta reducción como resultado de la restricción solicitada:

```
rest(min,Rec,si(i(A,B),d(C,D),f(E,F)),X,Y) :- A \= B,
        rest(sum,Rec,si(i(A,A),d(C,D),f(E,F)),X,Y).
```

Una vez calculadas las reducciones que sobre los intervalos ejecuta el predicado `rest`, el predicado `proceso` interpreta al functor `parcial` como predicado y lo pone en marcha. Este predicado tiene cuatro argumentos, los tres primeros los recibe instanciados y el último es una variable en la que regresa el resultado de su operación. Sus tres primeros argumentos corresponden a cada caso del último argumento del predicado `rest` luego de haber reducido los intervalos de una actividad. Si estos intervalos resultaron estables ante cada una de las reducciones las tres listas estarán vacías, pero si al menos una reducción modificó a algún intervalo, una o más de estas listas no estará vacía, y en consecuencia el cuarto argumento del predicado `parcial` será una lista diferente de la vacía, ya que esta variable se instancia con el resultado de pegar las tres listas, para ello, primero se pega a la primera lista con la segunda, y el resultado obtenido se pega con la tercera lista:

```
parcial(Q,S,T,U) :- pega(Q,S,M), pega(M,T,U).
```

El predicado `pega` utilizado para pegar dos listas posee una definición recursiva. Tiene tres argumentos, las dos listas que debe pegar, y una variable donde unificará el resultado de su operación. El modo en que opera consiste en recorrer la primera lista hasta llegar al final, y luego pegar la segunda lista al final de la primera, de ahí que el caso base considere como primer argumento a la lista vacía:

```
pega([],Y,Y).
```

En el caso recursivo, como ya se mencionó, se recorre la primera lista en su totalidad y cada elemento nuevo se añade al resultado:

```
pega([W|X],Y,[W|Z]) :- pega(X,Y,Z).
```

Cuando el predicado **proceso** termina de recorrer la lista proporcionada por el predicado **red**, finaliza la tarea del metaintérprete **goal**, y la siguiente condición que debe satisfacer la cuarta cláusula de la definición del predicado **relaja** es la impuesta por el predicado **traslada** el cual tiene cinco parámetros y opera a través de otros dos predicados. El primer argumento del predicado **traslada** es el nombre del recurso compartido por las actividades con que trabajó el metaintérprete, los siguientes dos argumentos son la lista de actividades pendientes de reducción y la lista de actividades reducidas, y los últimos dos argumentos son dos variables en las que se unificarán las dos listas mencionadas luego de ser modificadas, es decir, son la nueva lista de actividades pendientes y la nueva lista de actividades reducidas. El predicado **traslada** opera a través del predicado **miembros** y a través del predicado **pega**:

```
traslada(H,A,C,F,G) :- miembros(H,C,B,G), pega(B,A,F).
```

El predicado **miembros** tiene una definición recursiva, y sus cuatro argumentos son: el nombre de un recurso, una lista de actividades pendientes de reducción, y dos variables no instanciadas al principio. Este predicado opera recorriendo la primera lista y repartiendo cada elemento de dicha lista en alguna de las otras dos listas. De esta manera, el caso base del predicado en cuestión se verifica cuando la primera lista se ha vaciado, lo que motiva que las otras listas toquen también su fin:

```
miembros(_,[],[],[]).
```

El caso recursivo requiere dos opciones, la primera cuando el recurso recibido por el predicado es el mismo que requiere la actividad en turno, la actividad se incorpora en la primera variable, y se hace la llamada recursiva:

```
miembros(Rec,[H|T],[H|R],S) :- H = act(_,Rec,_,_),
                               miembros(Rec,T,R,S).
```

Si por el contrario, la actividad en turno no precisa del recurso involucrado, dicha actividad se incluye en la segunda variable y se hace la llamada recursiva:

```
miembros(Rec,[H|T],R,[H|S]) :- H \= act(_,Rec,_,_),
                               miembros(Rec,T,R,S).
```

Con el resultado obtenido por el predicado **miembros** se tendrá parte de la respuesta, el predicado **pega** tomará el penúltimo argumento de **miembros** y lo pegará con el segundo argumento de **traslada** para obtener la nueva lista de actividades pendientes, y el cuarto argumento del predicado **miembros** es la nueva lista de actividades estables. Con las nuevas listas calculadas, el predicado **relaja** hará su llamada recursiva.
De regreso con **ejec_aux**, cuando el predicado **relaja** termina su tarea, nuevamente interviene el predicado **pega** que ya conocemos, para pegar el resultado arrojado por **relaja** con la lista de acarreo, entonces se tendran todos los elementos para hacer la llamada recursiva de **ejec_aux**:

```
ejec_aux(I,L,N,R) :- selecciona(I,L,K,M), perm(K,H),
                     relaja(I,I1,H,[],J), pega(J,N,O),
                     ejec_aux(I1,M,O,R).
```

Al llegar al fondo de la recursión, **ejec_aux** proporciona en su cuarta variable la respuesta solicitada por **ejecuta**.
El programa completo se encuentra en el apéndice con el nombre de **programa no. 7**.

3.2.4 Pruebas de velocidad

Una vez puestos en marcha los programas para resolver el problema de calendarización por cada uno de los tres métodos, se procedió a compararlos usando para ello entradas del mismo tamaño, y se obtuvieron los resultados que se muestran en la tabla 3.3. En ésta, el tiempo invertido por cada implantación se expresa en segundos, el número de actividades se refiere al total de actividades con que se corrió cada aplicación, y duración se refiere a la duración de cada una de las actividades. En los casos en que hay asteriscos el tiempo no se pudo medir (al aumentar el número de permutaciones en la Aritmética de Intervalos la memoria de la máquina se agota). Las corridas se llevaron a cabo en una máquina SUN-SPARCstation.
La implantación a través de Aritmética de Intervalos, fue la más lenta de las tres. A medida que el tamaño de la entrada crece, dicha implantación consume mucho más tiempo que las otras dos implantaciones. Se observó además que el tiempo de ejecución de esta implantación no depende del tamaño de las actividades sino del número de éstas.
Por otro lado se observó que las puestas en marcha de PROLOG y de la

Número de actividades	*Duración de c/actividad*	*Tiempo invertido*		
		CHIP	PROLOG	Arit. Int.
2	1	0.6	0.5	0.7
	5	0.7	0.6	0.9
	10	0.8	0.8	1.2
4	1	10.3	12.6	178.0
	5	15.6	13.4	187.9
	10	22.3	13.7	200.2
6	1	453.7	557.8	*
	5	694.7	590.0	*
	10	1005.0	597.3	*

Tabla 3.3: Tiempos de ejecución

emulación de CHIP no difieren mucho en el tiempo que invierten cuando el tamaño de duración de las actividades es de una unidad. Al incrementar el número de actividades, el tiempo invertido por estas dos implantaciones aumentó, pero la diferencia sigue siendo despreciable si el tamaño de las actividades es de una unidad, pero si dicho tamaño se incrementa, la emulación de CHIP, requiere de más tiempo para ejecutarse. Cuando la duración de las actividades es muy grande, la diferencia entre los tiempos requeridos por una y otra implantación se hace más significativa. Se hace evidente que el tiempo de ejecución requerido por la implantación en PROLOG depende sólamente del número de actividades, mientras que la emulación de CHIP además depende del tamaño de éstas.

Notas Bibliográficas.

[1] J. G. Cleary; Logical Aritmetic; *Future Computing Systems*, II(2): 125-149; 1987.

[2] L. Sterling, E. Shapiro; 1986; *The Art of Prolog*; MIT Press; U.S.A.; p.t. 437.

[3] P. Van Hentenryck; 1989; *Constrain Satisfaction in Logic Programming*; MIT Press; U.S.A.; p.t. 224.

Capítulo 4

Conclusiones

A lo largo del presente trabajo, se ha podido comprobar la utilidad de la programación lógica en la resolución de problemas basados en restricciones, entre los cuales se encuentran los problemas de calendarización.
El uso de diferentes métodos derivados de la programación lógica, en la resolución del tipo de problemas que nos ocupa, proporcionó la posibilidad de contrastar a dichos métodos, unos con otros. Esta comparación permitió generar algunas observaciones intreresantes. A partir de la implantación en PROLOG y de la emulación de CHIP se pudo observar que fuera de las facilidades que proporciona CHIP en el manejo de dominios enteros, no proporciona elementos adicionales (en este dominio), que nos permitan ponderarlo como una herramienta con posibilidades de velocidad y eficiencia superiores a PROLOG. En cuanto a la manipulación de problemas basados en restricciones a través de la Aritmética de Intervalos, se advirtió que, si bien constituye una herramienta que aporta elementos técnicos muy interesantes, resultó poco viable debido a que su puesta en marcha provocó que la corrida fuera lenta. La implantación con Aritmética de Intervalos invirtió en su ejecución una cantidad de tiempo exponencial con respecto al número de restricciones, en las pruebas que nosotros llevamos a cabo.
Una posibilidad para reducir este inconveniente, consiste en buscar una heurística que se adecue al manejo de redes de restricciones y que reduzca la complejidad de este proceso y que también garantice que todas las restricciones queden satisfechas, con la finalidad de que dicha heurística sustituya al algoritmo de relajación.
Otra alternativa viable podemos encontrarla en el lenguaje de programación

CLP($\Re$) [1], el cual se basa, también, en la programación lógica y está orientado a la resolución de problemas basados en restricciones. Una de las diferencias más significativas con el lenguaje de programación PROLOG consiste en que la unificación es sustituida por un mecanismo de resolución de restricciones.

Notas Bibliográficas.

[1] J. Jaffar, S. Michaylov, P. J. Stuckey, R. Yap; The CLP($\Re$) Language and System; *acm Transactions on Programming Languages and Systems*, XIV(3):339-395; 1992.

Apéndice

```
reinas(X) :- reticula(X), solucion(X).
reticula([(1,X1),(2,X2),(3,X3),(4,X4),(5,X5),(6,X6),(7,X7),(8,X8)]).
solucion([]).
solucion([(X,Y)|Otras]):- solucion(Otras),
                          miembro(Y,[1,2,3,4,5,6,7,8]),
                          noataques((X,Y),Otras).
miembro(X,[X|Y]).
miembro(X,[Y|Ys]) :- miembro(X,Ys).
noataques(_,[]).
noataques((X,Y),[(X1,Y1)|Otras]) :- Y =\= Y1,
                                    Y1-Y =\= X1-X,
                                    Y1-Y =\= X-X1,
                                    noataques((X,Y),Otras).
```

Programa no. 1

```
reinas(X) :- solucion( X,
                       [1,2,3,4,5,6,7,8],  [1,2,3,4,5,6,7,8],
                       [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
                       [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]).
solucion([],[],[],M,N).
solucion([R|Rs],[X|Xs],Y,M,N) :- borra(R,Y,Ys),
                                 K is X + R - 1,
                                 borra(K,M,Ms),
                                 L is R - X + 8,
                                 borra(L,N,Ns),
                                 solucion(Rs,Xs,Ys,Ms,Ns).
borra(X,[X|Xs],Xs).
borra(X,[Y|Ys],[Y|Xs]) :- borra(X,Ys,Xs).
```

Programa no. 2

```
todas([pa,ai,a2,a3,a4,a5,a6,ue,fic,s1,s2,s3,s4,s5,s6,b1,b2,b3,
       b4,b5,b6,ab1,ab2,ab3,ab4,ab5,ab6,m1,m2,m3,m4,m5,m6,l]).
pert(Lista) :- todas(AN), programa(AN,[],Lista).
programa([],Acts,Acts).
programa(AN,AP,R) :- borra(A,AN,Acts),
                     actividad(A,_,Pred,_),
                     subconj(Pred,AP),
                     programa(Acts,[A|AP],R).
borra(X,[X|Xs],Xs).
borra(X,[Y|Ys],[Y|Z]) :- borra(X,Ys,Z).
subconj([],_).
subconj([X|Xs],Y) :- miembro(X,Y), subconj(Xs,Y).
miembro(X,[X|Xs]).
miembro(X,[Y|Ys]) :- miembro(X,Ys).
```

Programa no. 3

```
dominios([rec(aux1,[]),    rec(aux10,[1,2]),
          rec(exc,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]),
          rec(aux2,[1,2,3,4,5,6,7,8,9,10]),  rec(aux3,[1,2,3,4,5,6]),
          rec(car1,[1,2,3,4,5,6,7,8]),  rec(car2,[1,2,3,4]),
          rec(car3,[1,2,3,4]),  rec(car4,[1,2,3,4]),
          rec(car5,[1,2,3,4]),  rec(car6,[1,2,3,4,5,6,7,8,9,10]),
          rec(rev,[1,2,3,4,5,6]),  rec(aux4,[1]),  rec(aux5,[1]),
          rec(aux6,[1]),  rec(aux7,[1]),  rec(aux8,[1]),  rec(aux9,[1]),
          rec(mas1,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]),
          rec(mas2,[1,2,3,4,5,6,7,8]),  rec(mas3,[1,2,3,4,5,6,7,8]),
          rec(mas4,[1,2,3,4,5,6,7,8]),  rec(mas5,[1,2,3,4,5,6,7,8]),
          rec(mas6,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])  ]).
actividad(pa,[],[],aux1).   actividad(fic,[1,2,3,4,5,6],[pa],aux3).
actividad(a1,[1,2,3,4],[pa],exc).  actividad(a2,[1,2],[pa],exc).
actividad(a3,[1,2],[pa],exc).  actividad(a4,[1,2],[pa],exc).
actividad(a5,[1,2],[pa],exc).  actividad(a6,[1,2,3,4,5],[pa],exc).
actividad(ue,[1,2,3,4,5,6,7,8,9,10],[pa],aux2).
actividad(s1,[1,2,3,4,5,6,7,8],[fic],car1).
actividad(s2,[1,2,3,4],[fic],car2).   actividad(s3,[1,2,3,4],[fic],car3).
actividad(s4,[1,2,3,4],[fic],car4).   actividad(s5,[1,2,3,4],[fic],car5).
actividad(s6,[1,2,3,4,5,6,7,8,9,10],[fic],car6).
actividad(b1,[1],[s1],rev).  actividad(b2,[1],[s2],rev).
actividad(b3,[1],[s3],rev).  actividad(b4,[1],[s4],rev).
actividad(b5,[1],[s5],rev).  actividad(b6,[1],[s6],rev).
actividad(ab1,[1],[b1],aux4).  actividad(ab2,[1],[b2],aux5).
actividad(ab3,[1],[b3],aux6).  actividad(ab4,[1],[b4],aux7).
actividad(ab5,[1],[b5],aux8).  actividad(ab6,[1],[b6],aux9).
actividad(m1,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],[ab1],mas1).
actividad(m2,[1,2,3,4,5,6,7,8],[ab2],mas2).
actividad(m3,[1,2,3,4,5,6,7,8],[ab3],mas3).
actividad(m4,[1,2,3,4,5,6,7,8],[ab4],mas4).
actividad(m5,[1,2,3,4,5,6,7,8],[ab5],mas5).
actividad(m6,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],[ab6],mas6).
actividad(1,[1,2],[m1,m2,m3,m4,m5,m6],aux10).
```

Base no. 1

```
todas([pa,a1,a2,a3,a4,a5,a6,ue,fic,s1,s2,s3,s4,s5,s6,b1,b2,b3,
       b4,b5,b6,ab1,ab2,ab3,ab4,ab5,ab6,m1,m2,m3,m4,m5,m6,l]).
pert(Lista) :- todas(AN), dominios(Doms), programa(AN,Doms,[],Lista).
programa([],_,Acts,Acts).
programa(AN,Doms,AP,R) :- borra(A,AN,Acts),
                          actividad(A,Durac,_,Rec),
                          clc_en_dom(Rec,Durac,Doms,Lst_tpo_us,Doms1),
                          programa(Acts,Doms1,[act(A,Lst_tpo_us)|AP],R).
borra(X,[X|Xs],Xs).
borra(X,[Y|Ys],[Y|Z]) :- borra(X,Ys,Z).
clc_en_dom(Recs,Durac,Doms,[Rec|Restante],[rec(Rec,X)|DomAux]) :-
                          borra(rec(Rec,Y),Doms,DomAux),
                          saca(Y,Durac,[],Restante,X).
saca(X,[],Restante,Restante,X).
saca([A|B],[C|D],Rest,Rsp,X) :- saca(B,D,[A|Rest],Rsp,X).
```

Programa no. 4

```
todas([pa,a1,a2,a3,a4,a5,a6,ue,fic,s1,s2,s3,s4,s5,s6,b1,b2,b3,
       b4,b5,b6,ab1,ab2,ab3,ab4,ab5,ab6,m1,m2,m3,m4,m5,m6,l]).
pert(Lista) :- todas(AN), dominios(Doms), programa(AN,Doms,[],Lista).
programa([],_,Acts,Acts).
programa(AN,Doms,AP,R) :- borra(A,AN,Acts),
                          actividad(A,Durac,Pred,Rec),
                          subconj(Pred,AP),
                          clc_en_dom(Rec,Durac,Doms,Lst_tpo_us,Doms1),
                          programa(Acts,Doms1,[act(A,Lst_tpo_us)|AP],R).
borra(X,[X|Xs],Xs).
borra(X,[Y|Ys],[Y|Z]) :- borra(X,Ys,Z).
subconj([],_).
subconj([X|Xs],Y) :- miembro(X,Y), subconj(Xs,Y).
miembro(X,[act(X,_)|Xs]).
miembro(X,[Y|Ys]) :- miembro(X,Ys).
clc_en_dom(Rec,Durac,Doms,[Rec|Restante],[rec(Rec,X)|DomAux]) :-
                          borra(rec(Rec,Y),Doms,DomAux),
                          saca(Y,Durac,[],Restante,X).
saca(X,[],Restante,Restante,X).
saca([A|B],[C|D],Rest,Rsp,X) :- saca(B,D,[A|Rest],Rsp,X).
```

Programa no. 5

```
dominios([rec(aux1,i(0,0)),
          rec(exc,i(0,17)), rec(aux2,i(0,10)), rec(aux3,i(0,6)),
          rec(car1,i(0,8)), rec(car2,i(0,4)), rec(car3,i(0,4)),
          rec(car4,i(0,4)), rec(car5,i(0,4)), rec(car6,i(0,10)),
          rec(rev,i(0,6)), rec(aux4,i(0,1)), rec(aux5,i(0,1)),
          rec(aux6,i(0,1)), rec(aux7,i(0,1)), rec(aux8,i(0,1)),
          rec(aux9,i(0,1)), rec(mas1,i(0,16)), rec(mas2,i(0,8)),
          rec(mas3,i(0,8)), rec(mas4,i(0,8)), rec(mas5,i(0,8)),
          rec(mas6,i(0,15)), rec(aux10,i(0,2)) ]).
actividad(pa,i(0,0),[],aux1).
actividad(a1,i(0,4),[pa],exc).   actividad(a2,i(0,2),[pa],exc).
actividad(a3,i(0,2),[pa],exc).   actividad(a4,i(0,2),[pa],exc).
actividad(a5,i(0,2),[pa],exc).   actividad(a6,i(0,5),[pa],exc).
actividad(ue,i(0,10),[pa],aux2).   actividad(fic,i(0,6),[pa],aux3).
actividad(s1,i(0,8),[fic],car1).   actividad(s2,i(0,4),[fic],car2).
actividad(s3,i(0,4),[fic],car3).   actividad(s4,i(0,4),[fic],car4).
actividad(s5,i(0,4),[fic],car5).   actividad(s6,i(0,10),[fic],car6).
actividad(b1,i(0,1),[s1],rev).   actividad(b2,i(0,1),[s2],rev).
actividad(b3,i(0,1),[s3],rev).   actividad(b4,i(0,1),[s4],rev).
actividad(b5,i(0,1),[s5],rev).   actividad(b6,i(0,1),[s6],rev).
actividad(ab1,i(0,1),[b1],aux4).   actividad(ab2,i(0,1),[b2],aux5).
actividad(ab3,i(0,1),[b3],aux6).   actividad(ab4,i(0,1),[b4],aux7).
actividad(ab5,i(0,1),[b5],aux8).   actividad(ab6,i(0,1),[b6],aux9).
actividad(m1,i(0,16),[ab1],mas1).   actividad(m2,i(0,8),[ab2],mas2).
actividad(m3,i(0,8),[ab3],mas3).   actividad(m4,i(0,8),[ab4],mas4).
actividad(m5,i(0,8),[ab5],mas5).   actividad(m6,i(0,15),[ab6],mas6).
actividad(l,i(0,2),[m1,m2,m3,m4,m5,m6],aux10).
```

Base no. 2

```
todas([pa,a1,a2,a3,a4,a5,a6,ue,fic,s1,s2,s3,s4,s5,s6,b1,b2,b3,
       b4,b5,b6,ab1,ab2,ab3,ab4,ab5,ab6,m1,m2,m3,m4,m5,m6,1]).
pert(Lista) :- todas(AN), dominios(Doms), programa(AN,Doms,[],Lista).
programa([],_,Acts,Acts).
programa(AN,Doms,AP,R) :- borra(A,AN,Acts),
                          actividad(A,Durac,Pred,Rec),
                          subconj(Pred,AP),
                          clc_en_dom(Rec,Durac,Doms,Lst_tpo_us,Doms1),
                          programa(Acts,Doms1,[act(A,Lst_tpo_us)|AP],R).
borra(X,[X|Xs],Xs).
borra(X,[Y|Ys],[Y|Z]) :- borra(X,Ys,Z).
subconj([],_).
subconj([X|Xs],Y) :- miembro(X,Y), subconj(Xs,Y).
miembro(X,[act(X,_)|Xs]).
miembro(X,[Y|Ys]) :- miembro(X,Ys).
clc_en_dom(Rec,Durac,Doms,[Rec,Restante],[rec(Rec,X)|DomAux]) :-
                          borra(rec(Rec,Y),Doms,DomAux),
                          saca(Y,Durac,Restante,X).
saca(i(A,B),i(C,D),i(A,F),i(F,D)) :- F is A + D - C.
```

Programa no. 6

```
todas([pa,a1,a2,a3,a4,a5,a6,ue,fic,s1,s2,s3,s4,s5,s6,b1,b2,b3,
       b4,b5,b6,ab1,ab2,ab3,ab4,ab5,ab6,m1,m2,m3,m4,m5,m6,l]).
ejecuta(M) :- dominios(Q), todas(L), reesc(Q,L,L1),
              separa(L1,L2), ejec_aux([],L2,[],M).
reesc(_,[],[]).
reesc(Q,[A|T],[act(A,Rec,si(i(Y,Z),d(X,X),f(Y,Z)),P)|L]) :-
                  actividad(A,i(_,X),P,Rs), borra(Rec,Rs,_),
                  miembro(rec(Rec,i(Y,Z)),Q), reesc(Q,T,L).
borra(X,[X|Y],Y).
borra(X,[W|Y],[W|Z]) :- borra(X,Y,Z).
miembro(A,[A|_]).
miembro(A,[_|B]) :- miembro(A,B).
separa([],[]).
separa([H|T],L) :- separa(T,L1), acomoda(H,L1,L).
acomoda(H,[],[[H]]).
acomoda(act(A,B,C,P),[[act(D,B,E,P1)|T]|Ts],
                     [[act(A,B,C,P),act(D,B,E,P1)|T]|Ts]).
acomoda(act(A,B,C,P),[[act(D,E,F,P1)|T]|Ts],[[act(D,E,F,P1)|T]|L]) :-
                     B \= E, acomoda(act(A,B,C,P),Ts,L).
ejec_aux(_,[],R,R).
ejec_aux(I,L,N,R) :- selecciona(I,L,K,M), perm(K,H), relaja(I,I1,H,[],J),
                     pega(J,N,O), ejec_aux(I1,M,O,R).
selecciona(I,[L|Ls],L,Ls) :- revisa(I,L), !.
selecciona(I,[L|Ls],K,[L|M]) :- selecciona(I,Ls,K,M).
revisa(I,[]).
revisa(I,[act(N,R,S,P)|L]) :- subconj(P,I), revisa(I,L).
subconj([],_).
subconj([X|Xs],Y) :- miembro(X,Y), subconj(Xs,Y).
perm([],[]).
perm([W|X],Z) :- perm(X,Y), borra(W,Z,Y).
relaja(S,S,[],M,M).
relaja(S,S1,[act(N,R,A,P)],C,M) :- rest(sum,R,A,A1,_),
       rest(min,R,A1,A2,_), pega(C,[act(N,R,A2,P)],D),
       relaja([N|S],S1,[],D,M).
relaja(S,S1,[act(N1,R,A,P),act(N2,R,B,P1)|C],D,M) :-
   goal([rest(sum,R,A,_,[]),
         rest(sum,R,B,si(_,Y,Z),_),
```

```
        rest(me,R,_,si(_,X),[]),
        rest(min,R,_,A1,[]),_]),
   relaja([N1|S],S1,[act(N2,R,si(X,Y,Z),P1)|C],[act(N1,R,A1,P)|D],M).
relaja(S,S1,[act(N1,R,A,P),act(N2,R,B,P1)|C],D,M) :-
          goal([rest(sum,R,A,_,_),
                rest(sum,R,B,si(_,Y,Z),_),
                rest(me,R,_,si(_,X),_),
                rest(min,R,_,A1,_),
                parcial(_,_,_,[H|T]) ]),
          traslada(H,[act(N2,R,si(X,Y,Z),P1)|C],D,E,F),
   relaja([N1|S],S1,E,[act(N1,R,A1,P)|F],M).


goal(X) :- red(X), proceso(X).
red([ rest(sum,Rec,si(X1,Y1,Z1),si(X1p,Y1p,Z1p),Q),
      rest(sum,Rec,si(X2,Y2,Z2),si(X2p,Y2p,Z2p),_),
      rest(me,Rec,si(Z1p,X2p),si(Z1q,X2q),S),
      rest(min,Rec,si(X1p,Y1p,Z1q),si(X1r,Y1r,Z1r),T),
      parcial(Q,S,T,U)  ]).
proceso([]).
proceso([X|Y]) :- X, proceso(Y).
parcial(Q,S,T,U) :- pega(Q,S,M), pega(M,T,U).


rest(sum,Rec,si(i(A,B),d(C,D),f(E,F)),si(i(M,N),d(C,D),f(Q,R)),[]) :-
          K is E - D, L is F - C, inter(i(A,B),i(K,L),i(M,N)),
          O is A + D, P is B + C, inter(i(E,F),i(O,P),i(Q,R)),
          [A,B,E,F] == [M,N,Q,R].
rest(sum,Rec,si(i(A,B),d(C,D),f(E,F)),si(i(M,N),d(C,D),f(Q,R)),[Rec]) :-
          K is E - D, L is F - C, inter(i(A,B),i(K,L),i(M,N)),
          O is A + D, P is B + C, inter(i(E,F),i(O,P),i(Q,R)),
          [A,B,E,F] \= [M,N,Q,R].
rest(me,Rec,si(f(E,F),i(M,N)),si(f(E,E),i(N,N)),[Rec]) :- E = N.
rest(me,Rec,si(f(E,F),i(M,N)),si(f(E,F),i(M,N)),[]) :- F =< M.
rest(me,Rec,si(f(E,F),i(M,N)),si(f(G,H),i(G,H)),[Rec]) :-
          E < N, F > M, inter(i(E,F),i(M,N),i(G,H)).
rest(min,Rec,si(i(A,A),d(C,D),f(E,F)),si(i(A,A),d(C,D),f(E,F)),[]).
```

```
rest(min,Rec,si(i(A,B),d(C,D),f(E,F)),X,Y) :- A \= B,
          rest(sum,Rec,si(i(A,A),d(C,D),f(E,F)),X,Y).
inter(i(_,E),i(E,_),i(E,E)).
inter(i(A,B),i(C,D),i(E,F)) :- C < B, comp(A,C,E,_),
                               comp(B,D,_,F), F >= E.
comp(A,B,A,B) :- A >= B.
comp(A,B,B,A) :- A < B.
traslada(H,A,C,F,G) :- miembros(H,C,B,G), pega(B,A,F).
miembros(_,[],[],[]).
miembros(Rec,[H|T],R,[H|S]) :- H \= act(_,Rec,_,_), miembros(Rec,T,R,S).
miembros(Rec,[H|T],[H|R],S) :- H = act(_,Rec,_,_), miembros(Rec,T,R,S).
pega([],Y,Y).
pega([W|X],Y,[W|Z]) :- pega(X,Y,Z).
```

Programa no. 7