PROGRAMACION CONCURRENTE CON RESTRICCIONES EN MOZART

JULIO HERNANDO VARGAS
Profesor Asociado
Departamento de Matemáticas
Universidad Tecnológica de Pereira.
jhvargas@utp.edu.co

RESUMEN

En este artículo daremos una breve introducción a la programación concurrente con restricciones en Mozart. La tarea central (considerada un arte) al hacer programación con restricciones es encontrar un buen modelo para describir el problema y diseñar propagadores y estrategias de distribución que minimicen la complejidad del árbol de búsqueda. Con unos pocos ejemplos (combinatorios y de planificación) mostramos como usar algunas de las herramientas básicas (pero poderosas) que se tienen en el sistema Mozart.

PALABRAS CLAVES: Programación concurrente con restricciones, problemas combinatorios, planificación y optimización.

ABSTRACT

In this article we will give a brief introduction to the concurrent constraint programming whit Mozart. The central task (considered as an art) in doing a constraint program is to find a good model to describe the problem and device propagators and distribution strategy that minimize the complexity of the search tree. Whit a few examples (combinational and of scheduling) we illustrate how to use some of the most basic (but powerful) tools that we have in the Mozart system.

KEYWORDS: *Concurrent constraint programming, combinatoria problems, scheduling and optimization*

1. INTRODUCCIÓN

La programación concurrente con restricciones es una tecnología para resolver problemas combinatorios. Estos problemas se caracterizan porque el tamaño del espacio de búsqueda crece exponencialmente con el tamaño de los datos de entrada dificultando el diseño de algoritmos "rápidos" para solucionarlos.
El método de generar e ir verificando todas las posibles soluciones a un problema es casi siempre impracticable por lo que se hace necesario el diseño de estrategias que superen ese método. Una de las principales estrategias que han ayudado en la búsqueda de soluciones consiste en hacer tanta inferencia formal como sea posible usando el modelo del problema de modo que el espacio formado por las posibles soluciones se reduzca, y solo entonces aplicar un método de búsqueda controlada. Un sistema de programación concurrente con restricciones debe entonces proporcionar:

- Un conjunto de procedimientos para definir propagadores.
- Toda la maquinaria necesaria para hacer propagación y distribución.

El sistema Mozart, desarrollado por el consorcio Mozart formado por grupos de investigacion del Laboratorio de Sistemas de Programación de Saarbrucken, del Instituto Sueco para las Ciencias de la Computación y de la Universidad Católica de Lovaina, proporciona la tecnología mas innovativa comparado con los demás lenguajes, no solamente en el área de la programación con restricciones. Además es de dominio publico, esta extensamente documentado y es completamente operacional. En este artículo presentamos en una forma sencilla las ideas básicas de la propagación y la distribución y tres ejemplos, muy comunes en las aplicaciones industriales, escritos con el fin de mostrar la forma en que se pueden usar los módulos del sistema. La primera característica notable del sistema Mozart que estos ejemplos van a mostrar es la sencillez de su sintaxis, pues programas que constan de unas pocas líneas de código hacen tareas complejas que en otros lenguajes son mucho mas difíciles de programar.

2. PROPAGADORES Y DISTRIBUCION

Consideremos el problema de encontrar dos números enteros que cumplan las siguientes condiciones:

$$p1 : x + y = 20$$
$$p2 : x > 3y$$

Fecha de recepción: 29 Marzo de 2004
Fecha de aceptación: 16 Abril de 2004

¿Qué sabemos inicialmente? Obviamente podemos suponer que $x,\ y \in \{0, \ldots 20\}$. Imaginemos a cada una de las anteriores expresiones como un "agente" que puede razonar acerca de la condición que ella impone. $p1$ todavía no puede decirnos nada. Pero $p2$ puede preguntarse: ¿puede x ser igual a cero? Claramente no, pues el valor mínimo de y es 0 y no es cierto que $0 > 0$. Pero si puede ser alguno de los otros valores. Y y, ¿podría ser 20? No hay un valor de x que sea mayor a 60. Entonces y no puede ser igual a 20. Por esta misma razón se pueden ir eliminando para y los valores $19,\ 18, \ldots$ hasta 7. Por lo tanto $p2$ afirma:

$$x \in \{1,\ \ldots,\ 20\} \qquad y \in \{0,\ \ldots,\ 6\}$$

Ahora $p1$ se da cuenta del cambio ocurrido y analiza esta nueva información. ¿Puede ser x igual a 1? No, pues no hay un valor de y que sumado con uno sea igual a 20. Pero entonces tampoco puede ser 2, ni 3, y así hasta 13. Ahora puedo afirmar algo, dice $p1$:

$$x \in \{14,\ \ldots,\ 20\} \qquad y \in \{0,\ \ldots,\ 6\}$$

El anterior ejemplo es muy elemental pero sirve para ilustrar uno de los conceptos fundamentales de la programación con restricciones: la *propagación*. Un *propagador* es un agente que impone una condición y con ella trata de reducir los dominios de las variables que aparecen en su condición. Los propagadores $p1$ y $p2$ han conseguido reducir su espacio de posibilidades, de 441 iniciales quedan 49.
Un esquema general de la forma en que se hace propagación en Mozart es el siguiente: Hay un almacén de restricciones básicas, como $y \in \{1,\ \ldots,\ 20\}$ en el ejemplo anterior. Sobre este almacén operan en forma concurrente (cada uno en su propio hilo) un conjunto de propagadores, cuyo número puede llegar a ser mayor a 100.000.

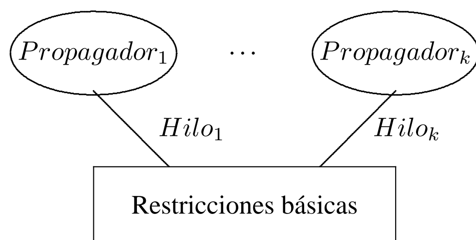

Figura 1. Propagación en Mozart.

Computación especulativa

Los propagadores $p1$ y $p2$ redujeron los dominios de las variables x y y pero no encontraron una solución al problema. Entonces aparece en escena el señor D quien es informado del problema. El trabajo del señor D es hacer especulaciones, así que sacando un pequeño tablero escribe $x = 14$.
Pero si $x = 14$, entonces tengo que decir que $y = 6$, dice $p1$, escribiendo esto último en el mismo tablero.

Lo siento pero 14 no es mayor que 3(6), interviene $p2$, y el señor D junto con su pequeño tablero desaparecen.
Simultáneamete, en un universo paralelo, han aparecido sendos clones de $p1$ y $p2$, junto con el señor D' (muy parecido a D) quien en su pequeño tablero escribe $x \neq 14$.

$$x \in \{15,\ \ldots,\ 20\} \qquad y \in \{0,\ \ldots,\ 5\}$$

propagan los clones de $p1$ y $p2$. Pero no pueden deducir nada más.
Entonces aparece el señor D" (muy parecido a D y a D') quien en su pequeño tablero escribe ...
En la programación con restricciones la propagación no es suficiente y hay que completarla con técnicas de búsqueda, que son llamadas *estrategias de distribución*. El guión principal de un modelo con restricciones se ejecuta en un *espacio de cómputo*. En ese espacio se hace toda la propagación posible hasta que se llega a un estado *estable*. Si aún no se tiene una solución, o si se quieren otras, un hilo en este espacio puede hacer distribución creando subespacios de cómputo, uno de los cuales agrega una condición ϕ y el otro su condición complementaria $\neg\phi$. Como en el ejemplo, un subespacio de cómputo puede *fallar* y entonces desaparece, o puede tener éxito en cuyo caso es *mezclado* con el espacio de nivel superior. El proceso es recursivo de modo que los sucesivos subespacios creados van formando un árbol de búsqueda para la solución del problema.

3. UN PROBLEMA COMBINATORIO

Consideremos el siguiente problema: dada una región de $m * n$ unidades de área, encontrar todos los subrectángulos de esa región que tengan un área determinada.

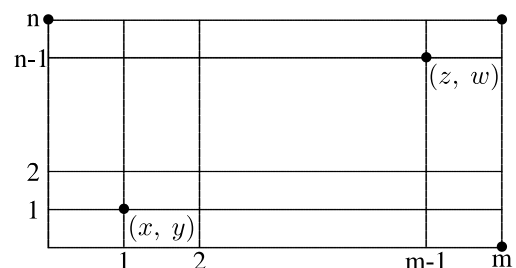

Figura 2. El problema de los rectángulos.

Veamos como modelar el problema. Un rectángulo está determinado por dos de sus vértices opuestos, que llamaremos $(x,\ y)$ y $(z,\ w)$.
La primera condición que se puede imponer a las variables $x,\ y,\ z$ y w tiene que ver con su dominio.

$$x,\ z,\ \in \{0,\ \ldots,\ m\} \quad y,\ w\ \in \{0, \ldots,\ n\},$$

Para evitar respuestas simétricas se pone como condición:

$$x < z \qquad y < w$$

Para que el área de un subrectángulo sea igual al valor A dado la condición es:

$$(z - x) * (w - y) = A$$

Definimos una función *Subr* cuyos parámetros son los valores de m, n y A. Está función devuelve un procedimiento de un sólo parámetro *Sol* llamado un solucionador para cada instancia del problema. *Sol* es una lista con las coordenadas de los puntos $(x,\ y)$ y $(z,\ w)$. Como estrategia de distribución se va a usar la estrategia ff (primero en fallar) que hace computación especulativa con la variable que tenga el menor dominio y con el menor valor de ese dominio.

```
declare
fun {Subr M N A}
   proc {$ Sol}
          X Y Z W
   in
             Sol=[X Y Z W]
             X::0#M
             Z::0#M
             Y::0#N
             W::0#N
             X<:Z
             Y<:W
             (Z-X)*(W-Y)=:A
             {FD.distribute ff Sol}
   end
end
```

Ahora se puede invocar la función *Subr* con el Explorador de Mozart. Al ejecutar

{ExploreAll {Subr 5 4 12}}

Se abre la ventana del explorador en la que se muestra el árbol de búsqueda para esta instancia del problema. En este árbol podemos ver siete hojas verdes en forma de diamante que corresponden a las siete soluciones encontradas. Al hacer doble clic sobre cada una de esas hojas en la ventana del Browser aparece la solución correspondiente. Por ejemplo, el nodo solución 6 corresponde a la solución $(0,\ 1)$, $(4,\ 4)$. Al extremo derecho aparece un triángulo rojo que corresponde a un subárbol en el que el proceso de búsqueda no encontró ninguna solución. También en la parte inferior de la ventana del explorador se nos informa que el sistema demoró menos de un milisegundo en la exploración del árbol. Si cambiamos el llamado a la función por:

{ExploreAll {Subr 10 10 12}}

Vemos que el árbol de búsqueda para esta instancia del problema es mucho más grande (Figura 3), su profundidad es 19 y en el hay 202 nodos solución. El tiempo que demora la búsqueda es de 16 milisegundos.

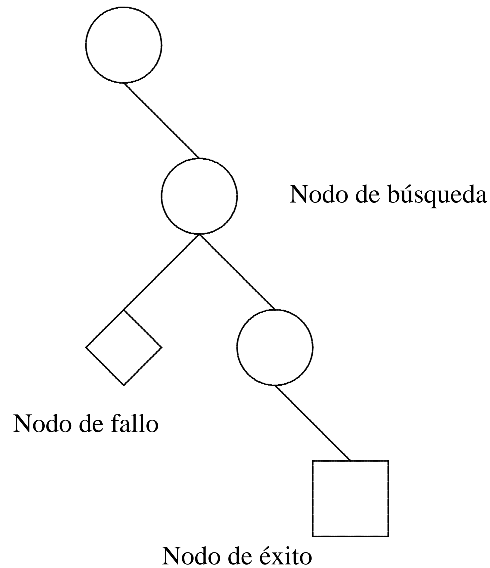

Figura 3. Árbol de búsqueda.

4. UN PROBLEMA DE PLANIFICACIÓN

Vamos a considerar el esquema general de un problema de planificación que consiste en la programación en secuencia de un conjunto de tareas {T1 ... TN}. Cada tarea tiene asociado 3 valores:

Minicio(Tk). El mínimo tiempo en que se puede iniciar Tk.
TiempoE(Tk). El tiempo que demora la ejecución de Tk.
TiempoF(Tk). El tiempo en que la tarea Tk termina.
Además un diagrama de precedencia determina un orden parcial sobre el conjunto de tareas.
El problema consiste en asignar a cada tarea un tiempo de inicio *inicio(Tk)* de modo que las tareas se pueden realizar secuencialmente cumpliendo todas las restricciones, en un tiempo T dado.
Para resolver este tipo de problemas vamos a agrupar las diferentes restricciones. Primero reunimos las restricciones que establecen que el tiempo de inicio de la tarea Tk no puede superar al tiempo final menos el tiempo de ejecución y tampoco puede ser inferior a *Minicio (Tk)*

$$A = \{Minicio(Tk) \leq T - tiempoE(Tk) \quad k = 1, \ldots, N\}$$

Después se tienen las restricciones que indican que todas las tareas deben finalizar antes del tiempo T

$$B = \{tiempoF(Tk) \leq T \quad k = 1, \ldots, N\}$$

También se agrupan las restricciones determinadas por el diagrama de precedencia D.

$$C = \{inicio(T_i) \geq tiempoF(T_j) \quad si \quad (T_j,\, T_i) \text{ está en } D\}$$

Finalmente están las restricciones que impiden que dos tareas se superpongan.

$$inicio(T_k) + tiempoE(T_i) \leq inicio(T_j)$$

$$inicio(T_j) + tiempoE(T_j) \leq inicio(T_i)$$

para todo $i \neq j$

El problema se puede generalizar asumiendo que hay más de un proceso, también llamado recurso, para realizar tareas. Cada tarea pasa por un único proceso y los procesos son independientes.

Como un primer ejemplo elemental, veamos como programar en Mozart un problema en el que hay varias tareas y 2 procesos, pero en el que no hay relaciones de precedencia. En la siguiente tabla se dan los datos del problema.

Tk	a	b	c	d	e	f	g	h	i	j
Pr	p1	p1	p1	p2	p2	p2	p2	p3	p3	p3
Dr	3	4	2	6	5	6	4	5	7	4

En Mozart, la información acerca de las restricciones del problema se pueden guardar en las siguientes estructuras de datos:

Inicior. Un registro de enteros de dominio finito que representan los tiempos de inicio de las tareas.

Dur. Un registro de enteros que corresponden a la duración de las tareas. Las aridades de *Inicior* y *Dur* deben ser iguales.

Dur = dur(a:3 b:4 c:2 d:6 e:5 f:6 g:4 h:5 i:7 j:4)
Inicio = {FD.record inicio [a b c d e f g h i j] 0# FD.sup}

Aquí se crea un registro con la etiqueta *inicio* y con características $a \ldots j$. Las variables de este registro se declaran inicialmente en el intervalo 0# 134217726.

Tareas= tareas(puno:[a b c] pdos:[d e f g])

las tareas a, b, y c se realizan en el proceso uno y las tareas d, e, f, y g en el dos.

{Schedule.serialized Dis Tareas Inicio Dur}

se crea un propagador para que las tareas realizadas por un mismo proceso no se crucen.

{FD.distribute ff Inicio}

se escoge usar la estrategia de distribución "primero en fallar".

Supongamos que ahora se tiene el siguiente diagrama de precedencia:

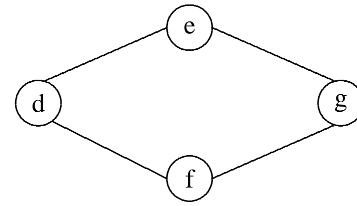

Figura 4. Diagrama de precedencia.

agregando las restricciones determinadas por este diagrama se obtiene el siguiente programa:

```
declare
proc Plan Inicio
   Dur = dur(a:3 b:4 c:2 d:6 e:5 f:6 g:4)
   Tareas = tareas(puno:[a b c] pdos:[d e f g])
in
   Inicio = {FD.record inicio [a b c d e f g] 0#FD.sup}
   {Schedule.Serialized disj Tareas Inicio Dur}
   Inicio.d + Dur.d <: Inicio.e
   Inicio.d + Dur.d <: Inicio.f
   Inicio.e + Dur.e <: Inicio.g
   Inicio.f + Dur.f <: Inicio.g
   {FD.distribute ff Inicio}
end
```

La notación <: nos indica que cada una de esas relaciones va a actuar como un agente propagador. Para ejecutar este guión se usa una de las máquinas de búsqueda del sistema. Con *Search.base.one* se obtiene la primera solución que se encuentra usando la técnica de búsqueda llamada *primero en profundidad*:

Xs={Search.base.one Plan}
{Browse Xs}

La ventana del Browser muestra una lista con un único elemento:

[inicio(a:0 b:3 c:7 d:0 e:13 f:7 g:19)]

en el registro que aparece en la lista están los tiempos de inicio de cada una de las tareas. Se pueden usar otras máquinas de búsqueda para encontrar todas las soluciones o para encontrar la mejor solución definiendo adecuadamente un orden que permita establecer entre dos posibles soluciones cuál se prefiere.

5. UN PROBLEMA DE OPTIMIZACIÓN

En el análisis de la complejidad algorítmica un problema muy estudiado es el problema del empaquetamiento (conocido en inglés como *the bin packing problem*). En este problema se tiene un conjunto finito de objetos $C = \{0_1,\ 0_2, \ldots,\ 0_n\}$ y cada objeto tiene un "tamaño" racional $t(O_i) \in [0,\ 1]$. El problema consiste en encontrar una partición de C en subconjuntos disyuntos $C_1, \ldots C_k$ tal que la suma de los tamaños de los objetos en cada subconjunto C_i no sea mayor a 1 y tal

que k sea el menor entero posible. En [1] se muestra que este problema es NP-duro en el sentido más fuerte por lo que hay pocas esperanzas de encontrar algoritmos eficientes, si bien “hay algoritmos de aproximación que vale la pena considerar”.

Consideremos un problema diferente en el que los tamaños de los objetos son valores enteros y en el que la suma de los tamaños en cada subconjunto C_i no puede superar un valor límite L, i.e., vamos a tratar de optimizar el empaque de N objetos de tamaños enteros en paquetes de capacidad L. Este ejemplo es una adaptación del problema de guardar archivos en discos de la sección 4 de [2].

Cada objeto se va a representar como un registro de la forma *obj(nombre:N tam:T)*. Se define una función *Empacar* que tiene como parámetros una lista con los objetos y el valor de L y que devuelve un solucionador (**proc** {$ Paquetes }) para cada instancia del problema.

Se define una variable TamObjetos que es una variable que referencia una lista con los tamaños de los objetos

```
TamObjetos ={Map Objetos fun {$ F} F.tam end }
```

También se define una variable de dominio finito *Numpaq* que corresponde al número de paquetes en los que se van a agrupar los objetos

```
Numconj={FD.int 1#FD.sup}
```

el rango de esta variable se declara entre 1 y el mayor valor entero manejado por el sistema, pero ese dominio se puede reducir poniendo como cota inferior para esta variable un valor V igual a la suma de todos los tamaños dividida por la capacidad (mas uno si el residuo no es cero). Sobre esta variable se aplicará una estrategia de distribución ingenua en la que se van ensayando sucesivos valores para las variables partiendo de su cota mínima.

Como las restricciones de conjuntos finitos en Mozart solo pueden operar con enteros no negativos, el solucionador va a representar internamente los objetos individuales como enteros. Se crea entonces una lista llamada *LosObjetos* que es la lista [1 2 ... N].

```
LosObjetos = {List.number 1 {Length Objetos} 1 }
```

Ahora, usando el módulo FS se va a declarar una lista Ls de longitud *Numpaq* tal que todos los elementos L de esta lista cumplan:

$$\emptyset \subseteq L \subseteq conj(LosObjetos)$$

En seguida se declara que Ls debe ser una partición del conjunto formado por los elementos de la lista *LosObjetos*:

```
{FS.partition Ls {FS.value.make LosObjetos}}
```

También se debe verificar que la la suma de los tamaños de los elementos en cada subconjunto en Ls no supere al parámetro *Capacidad*. Para lograr esto se usa la función *reified.areIn* que devuelve una lista ES binaria que determina cuales de los elementos de LosObjetos están en un subconjunto D, y finalmente también se hace distribución ingenua sobre la variable Ls.

```
fun {Empacar Objetos Capacidad}
  proc {$ Paquetes}
        TamObjetos={Map Objetos fun {$ F} F.tam end}
        Numpaq={FD.int 1#FD.sup}
        LosObjetos={List.number 1 {Length Objetos} 1}
        Ls
        in
        {FD.distribute naive [Numpaq]}
        {FS.var.list.upperBound Numpaq LosObjetos Ls}
        {FS.partition Ls {FS.value.make LosObjetos}}
        {ForAll Ls proc {$ D} ES in
            {FS.reified.areIn LosObjetos D ES}
            {FD.sumC TamObjetos ES '=<:' Capaci
                                              dad}
        end}
        {FS.distribute naive Ls}
end
```

En el anterior programa solamente falta la definición de un procedimiento que se encarge de mostrar el resultado de la búsqueda de una forma que se pueda leer fácilmente pero, ¡eso es todo!

Para hacer una prueba de este algoritmo se pueden crear N objetos cuyos pesos sean N valores enteros aleatoriamente distribuidos entre 1 y 100, y hacer un llamado a la función con un parámetro de capacidad.

```
{OS.srand 0}
fun {Objetos L}
  if L==0 then nil else
        obj(nombre:101-L tam:(({OS.rand} div 328)+1))
                                        |{Objetos L-1} end
end
Pr={Objetos 10}
{ExploreOne {Empacar Pr 200}}
```

La complejidad del problema tratado resulta evidente cuando se ejecuta el anterior programa con 100 objetos y con paquetes de capacidad 100. Es posible que, en un computador de escritorio, despúes de explorar un árbol con más de 600 000 nodos se supere la capacidad normal de memoria del sistema.

6. CONCLUSIONES

En la vida práctica surgen muchas situaciones que se pueden modelar como problemas combinatorios. Muchos de estos problemas no pertenecen a la clase P y por lo tanto (hasta el momento) no existen algoritmos “rápidos” para solucionarlos. Parte de la investigación que se realiza en el área conocida como Inteligencia Artificial está orientada al desarrollo de sistemas que puedan proporcionar herramientas eficaces para manejar esa clase de problemas, y dentro de esos sistemas se destaca la programación concurrente con restricciones. Mediante tres ejemplos mostramos las técnicas básicas para ha-

cer programación concurrente con restricciones en Mozart. Los ejemplos usan solo algunas de las herramientas del sistema, pero muestran que en ese paradigma de programación la tarea fundamental consiste en construir un buen modelo matemático del problema de modo que los propagadores reduzcan sensiblemente los dominios y la distribución explore en la mejor forma el árbol de búsqueda. El sistema Mozart incluye el paradigma de programación funcional y una implementación "ultraliviana" de la concurrencia lo que permite hacer programación de alto orden lo cual se refleja en la sencillez de su sintaxis; como se mostró en los ejemplos, los programas constan de pocas líneas de código. Siendo software de licencia abierta, es importante su estudio para la transferencia del conocimiento científico.

BIBLIOGRAFÍA

[1] GAREY Michael, JOHNSON David. Computers and Intractability, A Guide to the Theory of NP-Completeness, 339 páginas, W. H. FREEMAN AND COMPANY, New York, 1979.

[2] MÜLLER Tobias. Problem Solving with Finite Set Constraints in Oz. A Tutorial, Versión 1.2.4, 23 páginas, documentación del sistema Mozart, 2002.

[3] SCHULTE Christian, SMOLKA Gert. Finite Domain Constraint Programming in OZ. A Tutorial, Versión 1.1.0, 112 páginas, documentación del sistema Mozart, 2000.

[4] SCHULTE Christian. Programming Constraint Services, 159 páginas,Springer-Verlag Berlín, 2002.

[5] VAN ROY Peter, HARIDI Seif. Concepts, Techiques, and Models of Computer Programming, 820 páginas, 2002.

[6] En la página Web www.mozart-oz.org se encuentra toda la documentación sobre el lenguaje Mozart.