PROGRAMACIÓN LINEAL Y ENTERA CON OPL PARA LA NUEVA EMPRESA

Por
RAMÓN GONZÁLEZ DEL CAMPO RODRÍGUEZ BARBERO
Profesor Asociado en la Universidad Complutense de Madrid

SUMARIO: 1. INTRODUCCIÓN. 1.1. Programación Lineal. 1.2. Programación Entera.- 2. PROGRAMACIÓN LINEAL. 2.1 Programación Lineal y Entera. 2.2 Programación Lineal «a intervalos» (piecewise).- 3. PLANIFICACIÓN CON OPL. 3.1. Introducción. 3.2. Conceptos. 3.3. Actividades en OPL. 3.4. Origen y horizonte. 3.5. Declaración de Actividades en OPL. 3.6. Restricciones con Actividades. 3.7. Recursos. 3.7.1. Recursos Unarios. 3.7.2. Recursos Discretos. 3.7.3. Reservoir (recursos renovables). 3.7.4. Recursos con Estado. 3.7.5. Recursos alternativos.

1. INTRODUCCIÓN

1.1. PROGRAMACIÓN LINEAL

- La programación lineal supone una importante herramienta para resolver problemas a través de unas estructuras muy cercanas a las del lenguaje natural.
- La programación lineal consiste en minimizar una función lineal cuyos argumentos deben satisfacer un conjunto de ecuaciones lineales.
- Los argumentos son variables reales y su valor tiene que ser no negativo.
- En un lenguaje matemático podríamos representar establecer el siguiente modelo de programación lineal:

$$\text{minimizar: } \Sigma_k\, c_k x_k \quad (1 \le k \le n)$$
$$\text{sujeto a:}$$
$$\Sigma_k\, a_{qk} x_k = b_j \quad (1 \le q \le m)$$
$$x_k \ge 0 \qquad (1 \le k \le n)$$

- Nótese que el considerar la minimización de una función no es una condición restrictiva. La maximización se puede considerar como un caso particular de la minimización.
- Hay una gran cantidad de problemas que pueden ser resueltos eficientemente usando este enfoque.

1.2. PROGRAMACIÓN ENTERA

- Supone aplicar el mismo esquema de la programación lineal a un conjunto de restricciones formadas por variables enteras.
- Aplicando la misma notación nos quedaría el siguiente esquema para la programación entera:

$$\text{minimizar: } \Sigma_k\, c_k x_k \quad (1 \le k \le n)$$
$$\text{sujeto a:}$$
$$\Sigma_k\, a_{qk} x_k = b_j \quad (1 \le q \le m)$$
$$x_k \ge 0 \qquad (1 \le k \le n)$$
$$\mathbf{x_k \text{ enteros}}$$

- Desgraciadamente el número de problemas que se pueden resolver eficientemente desde este enfoque es mucho menor que con la programación lineal. Esto se debe a que la restricción de que las variables sean enteras hace que el algoritmo de resolución realice una búsqueda combinatoria.

Una vez que ya conocemos la sintaxis de OPL estamos en disposición de abordar aquellos programas más generales que podemos resolver eficientemente mediante estas técnicas.

2. PROGRAMACIÓN LINEAL

Ejemplo 1. El problema de la producción.

Supongamos que cierta actividad industrial necesita alcanzar un determinado nivel de producción. Las características de la producción van a ser las siguientes:

- Tenemos dos tipos de producción: la interna y la externa.
- En la producción interna elaboramos los productos a partir de sus materias primas a determinado coste. Las materias primas son limitadas.
- La producción externa consiste simplemente en comprar a otras empresas, factorías, etc. los productos en cuestión a un coste más alto que en la producción interna.
- El objetivo consiste en minimizar el coste de la producción.

En principio tenemos que establecer una caracterización de las variables del problema de forma que nos posible expresar tanto la función a minimizar como las restricciones a las que va a estar sujeta de una forma sencilla.

Mediante los siguientes tipos enumerados establecemos los productos y recursos necesarios para ellos:

```
enum Productos ...;
enum Recursos ...;
```

Con las siguientes declaraciones:

```
float+ consumo[Productos,Recursos] = ...,
float+ capacidad[Recursos] = ...,
float+ demanda[Productos] = ...;
float+ insideCoste[Productos] = ...;
float+ outsideCoste[Productos] = ...,
```

tratamos el consumo de cada uno de los recursos necesarios para crear un determinado producto, la cantidad de recursos, el nivel de producción requerido y el coste (tanto interno como externo) de cada uno de los productos.

La declaración de variables donde representaríamos la distribución interna y externa para cierto nivel de producción serían:

```
var float+ inside[Productos] = ...;
var float+ outside[Productos] = ...;
```

de forma que la función a minimizar tendría el siguiente aspecto:

```
minimize
        sum(p in Productos)  (insideCoste[p]*inside[p] +
outsideCost[p]*outside[p])
```

con las siguientes restricciones:

```
subjetc to {
        forall(r in Recursos)
                sum(p in Productos) consumo[p,r] * inside[p] <=
capacidad[r];
        forall(p in Productos)
                inside[p] + outside[p] >= demanda[p];
};
```

juntándolo todo tendríamos el siguiente programa:

```
enum Productos ...;
enum Recursos ...;
float+ consumo[Productos,Recursos] = ...,
float+ capacidad[Recursos] = ...,
float+ demanda[Productos] = ...;
float+ insideCoste[Productos] = ...;
float+ outsideCoste[Productos] = ...,
var float+ inside[Productos] = ...;
var float+ outside[Productos] = ...;
minimize
        sum(p in Productos)  (insideCoste[p]*inside[p] +
outsideCost[p]*outside[p])
subjetc to {
        forall(r in Recursos)
                sum(p in Productos) consumo[p,r] * inside[p] <=
capacidad[r];
        forall(p in Productos)
                inside[p] + outside[p] >= demanda[p];
};
```

Una posible inicialización de los datos del programa podría ser la siguiente:

```
Productos = { kluski capellini fettucine };
Recursos = { harina huevos };
consumo = [[0.5 0.2] [0.4 0.4] [0.3 0.6]];
capacidad = [20, 40];
demanda = [100, 200, 300];
insideCoste = [0.6, 0.8, 0.3];
outsideCoste = [0.8, 0,9, 0.4];
```

donde manejamos 3 productos a partir de 2 recursos. Para cada producto tenemos que tener en cuenta cuántos recursos consume. El resultado del programa con estos datos sería el siguiente:

```
Optimal Soluction with Objetive Value 372.0000
    inside[kluski] = 40.0000
    inside[capellini] = 0.0000
    inside[fettucine] = 0.0000
    outside[kluski] = 60.0000
    outside[capellini] = 200.0000
    outside[fettucine] = 300.0000
```

Es de destacar que los costes, tanto externos como internos, influyen sobre qué producto será aquel que tenga más prioridad en su ejecución.

Ejemplo 2. Gasolinas.

Siguiendo con los planteamientos del ejemplo anterior podemos considerar el siguiente ejemplo que nos muestra cómo la adición de restricciones, que en otro lenguaje hubiese supuesto una complicación del código excesiva, en OPL son relativamente fáciles de manejar.

Tenemos una refinería en la cual producimos tres tipos de gasolina (super, regular y diesel) mediante la mezcla de tres tipos de crudo (que denominaremos respectivamente: crudo1, crudo2 y crudo3). Los tipos de crudo tienen un precio de compra y los tipos de gasolina de venta. Además, los tipos de gasolina deben cumplir unos criterios de calidad en cuanto a su octanaje y a su contenido en plomo:

	Octanos	Cont. plomo
super	≥10	≤1
regular	≥8	≤2
diesel	≥6	≤1

crudo1	12	0.5
crudo2	6	2.0
crudo3	8	3.0

Otras restricciones a satisfacer:

- La compañía debe producir al menos 3.000 barriles de super, 2.000 barriles de regular y 1.000 de diesel.
- Puede comprar como máximo 5.000 barriles de cada tipo de crudo
- y puede procesar hasta 14.000.
- Además, la compañía puede invertir en publicidad de forma que por cada dólar gastado la demanda en ese tipo de gasolina sube en 10 barriles.
- Y, por último, el procesar cada barril le cuesta 4 dólares.

El objetivo es obtener el mayor beneficio eligiendo convenientemente las mezclas y la cantidad de publicidad.

La forma más inmediata de representar la información de cada una de las mezclas es una matriz. Y, análogamente, para representar el gasto en publicidad por cada tipo de gasolina utilizaremos un vector. Con lo cual tendríamos:

```
enum Gasolinas ...,
enum Crudos ...;
struct TipoGas { float+ demanda; float+ precio; float+ octanos;
float+ plomo; };
struct TipoCrudo { float+ capacidad; float+ precio; float+ octanos;
float+ plomo; };
TipoGas gas[Gasolinas] = ...,
TipoCrudo crudo[Crudos] = ...;
float+ maxProduccion = ...;
float+ prodCoste = ...;
var float+ a[Gasolinas];
var float+ mezcla[Crudos,Gasolinas];
```

donde además, agrupamos para cada tipo de gasolina y crudo, sus características relevantes.

La influencia de la publicidad en la demanda la codificaríamos mediante la siguiente restricción:

```
forall(g in Gasolines)
                sum(o in Crudos) mezcla[o,g] = gas[g].demanda +
10*a[g];
```

mientras que la capacidad de procesamiento limitada nos llevaría a escribir:

```
forall(o in Crudos)
        sum(g in Gasolinas) mezcla[o,g] <= oil[o].capacidad;
```

donde mezcla[o,g] representa la cantidad, y no la proporción, de crudo o usada en la producción de gasolina de tipo g.

El límite de la producción nos llevaría a plantear la restricción:

```
sum(o in Crudos, g in Gasolinas) mezcla[o,g] <= max Produccion;
```

Y, por último, los criterios de calidad de cada tipo de gasolina establecerían que:

```
forall(g in Gasolinas)
        sum(o in Crudos)(crudo[o].octano – gas[g].octano) * mezcla[o,g]
>= 0;
forall(g in Gasolinas)
        sum(o in Crudos)(crudo[o].plomo – gas[g].plomo) * mezcla[o,g]
<= 0;
```

La función a maximizar, el beneficio, sería:

```
maximize
        sum(g in Gasolines, o in Crudos)
            (gas[g].precio – crudo[o].precio – prodCoste) * mezcla[o,g]
- sum(g in Gasolines) a[g]
```

En conjunto el programa en OPL quedaría como:

```
enum Gasolinas ...,
enum Crudos ...;
struct TipoGas { float+ demanda; float+ precio; float+ octanos;
float+ plomo; };
struct TipoCrudo { float+ capacidad; float+ precio; float+ octanos;
float+ plomo; };
TipoGas gas[Gasolinas] = ...,
TipoCrudo crudo[Crudos] = ...;
float+ maxProduccion = ...;
float+ prodCoste = ...;
var float+ a[Gasolinas];
var float+ mezcla[Crudos,Gasolinas];
maximize
        sum(g in Gasolines, o in Crudos)
            (gas[g].precio – crudo[o].precio – prodCoste) * mezcla[o,g]
```

```
- sum(g in Gasolines) a[g]
subject to {
        forall(g in Gasolines)
                sum(o in Crudos) mezcla[o,g] = gas[g].demanda +
10*a[g];
        forall(o in Crudos)
                sum(g in Gasolinas) mezcla[o,g] <= oil[o].capacidad;
sum(o in Crudos, g in Gasolinas) mezcla[o,g] <= max Produccion;
forall(g in Gasolinas)
        sum(o in Crudos)(crudo[o].octano-gas[g].octano)* mezcla[o,g]
>= 0;
forall(g in Gasolinas)
        sum(o in Crudos)(crudo[o].plomo-gas[g].plomo)* mezcla[o,g]
<= 0;
};
```

de forma que inicializando los datos

```
Gasolinas = { super regular diesel };
Crudos = { crudo1 crudo2 crudo3 };
gas = [
        #<demanda:3000 precio:70 octano:10 plomo:1>#
        #<demanda:2000 precio:60 octano:8 plomo:2>#
        #<demanda:1000 precio:50 octano:6 plomo:1>#];
crudo = [
        #<capacidad:5000 precio:45 octano:12 plomo:0.5>#
        #<capacidad:5000 precio:35 octano:6 plomo:2>#
        #<capacidad:5000 precio:25 octano:8 plomo:3>#];
maxProduccion = 14000;
prodCoste = 4;
```

obtendríamos la siguiente solución.

```
Optimal Soluction with Objetive Value 287750.0000
a[super] = 0.0000
a[regular] = 750.0000
a[diesel] = 0.0000
mezcla[crudo1,super] = 2000.0000
mezcla[crudo1,regular] = 2200.0000
mezcla[crudo1,diesel] = 800.0000
```

mezcla[crudo2,super] = 1000.0000
mezcla[crudo2,regular] = 4000.0000
mezcla[crudo2,diesel] = 0.0000
mezcla[crudo3,super] = 0.0000
mezcla[crudo3,regular] = 3300.0000
mezcla[crudo3,diesel] = 200.0000

Set covering.

Podemos considerar la construcción de una casa como la ejecución de un conjunto de tareas. Para realizar cada tarea es necesario tener cierta destreza que cada uno de los trabajadores de que disponemos puede tener o no. A su vez, cada trabajador va a cobrar una cantidad, independientemente del número de tareas que haga.

El objetivo consiste en establecer una asignación de trabajadores válida a cada una de las tareas que minimice el coste total.

Para poder representar los datos de este problema sobre variables enteras es útil definir un vector booleano donde establecer si un trabajador ha sido contratado o no:

```
Boolean hire[trabajadores];
```

donde Boolean tenemos que definirlo. Esta declaración, junto con el coste de cada trabajador, nos ayuda a establecer la función a minimizar:

```
var Boolean hire[trabajadores];
minimize
        sum(c in trabajadores) cost[c]*hire[c]
```

con la siguiente declaración de variables:

```
range Boolean 0..1;
int ntrabajadores = ..;
range trabajadores 1..ntrabajadores;
enum tareas ...;
{trabajadores} cualificado[tareas] = ...;
int cost[trabajadores] = ...;
```

La declaración: {trabajadores} cualificado[tareas] = ...; es especialmente útil porque nos permite representar de una forma cómoda qué trabajadores van a estar capacitados para realizar cada una de las tareas.

La única restricción que vamos a tener va a ser aquella en la cual exijamos que todas las etapas van a ser cubiertas por algún trabajador. Si tenemos que cuenta que cualificado[q] va a contener a todos los trabajadores habilitados para realizar

la etapa q, podemos escribir la restricción de la siguiente manera:

```
forall(j in Tareas)
        sum(c in cualificado[j]) hire[c] >= 1;
```

para cada tarea j tiene que existir, al menos, un trabajador cualificado.

El programa completo quedaría así:

```
range Boolean 0..1;
int ntrabajadores = ..;
range trabajadores 1..ntrabajadores;
enum tareas ...;
{trabajadores} cualificado[tareas] = ...;
int cost[trabajadores] = ...;
var Boolean hire[trabajadores];
minimize
        sum(c in trabajadores) cost[c]*hire[c]
subject to
        forall(j in Tareas)
                sum(c in cualificado[j]) hire[c] >= 1;
{Trabajadores} crew = { c | c in trabajadores : hire[c] = 1};
display crew;
```

donde "volcamos" en crew aquellos trabajadores contratados para mostrarlo por pantalla.

2.1. PROGRAMACIÓN LINEAL Y ENTERA

Naturalmente, la división entre programación lineal y entera no se traduce en una división estricta de los problemas que tenemos que abordar. En la mayoría de los casos ocurre que necesitamos que algunas variables tomen valores enteros mientras que otras sean reales.

OPL resuelve internamente este tipo de programas restringiendo la tecnología de la programación entera a aquellas variables enteras. El programador no nota el diferente tratamiento interno de cada tipo de variable.

Consideremos una modificación del problema de la producción que hemos visto anteriormente. Ahora vamos a suponer que la elaboración de un producto a partir de sus recursos va a necesitar una máquina. Esta máquina tendrá un coste y en el cálculo de la solución óptima tendremos que decidir, entre otras cosas, si la alquilamos o no. Si decidimos no alquilarla tendremos que recurrir a

la producción externa. El programa sería muy parecido a la versión inicial:

```
range Boolean 0..1;
enum Productos ...;
enum Recursos ...;
enum Maquinas ...;
float+ maxProduccion = ...;
struct TipoProducto {
        float+ demanda;
        float+ incost;
        float+ outcost;
        float+ use[Recursos];
        Maquinas maquina;
};
TipoProducto[Productos] = ...;
float+ capacidad[Recursos] = ...;
float+ rentCost[Maquinas] = ...;
var Boolean rent[Maquinas];
var float+ inside[Productos];
var float+ outside[Productos];
minimize
        sum(p in Productos) (producto[p].incost*inside[p] +
producto.outcost*outside[p])+
        sum(m in Maquinas) rentCost[m] * rent[m]
subject to {
        forall(r in Recursos)
                sum(p in Productos) producto[p].use[r]*inside[p] <=
capadicad[r];
        forall(p in Productos)
                inside[p] + outside[p] >= producto[p].demanda;
        forall(p in Productos)
                inside[p] <= maxProduccion *
rent[producto[p].maquina];
};
```

donde:

```
        float+ rentCost[Maquinas] = ...;
```

representa el coste de alquiler de cada máquina,

```
var Boolean rent[Maquinas];
```

la opción de alquilar (o no cada una de las máquinas)

```
sum(m in Maquinas) rentCost[m] * rent[m]
```

es el coste, a minimizar, de las máquinas alquiladas. Y, por último, tenemos que representar la necesidad para cada producto de tener su máquina:

```
forall(p in Productos)
        inside[p] <= maxProduccion * rent[producto[p].maquina];
```

2.2. PROGRAMACIÓN LINEAL "a intervalos" (piecewise)

Este tipo de programación es en el fondo un azúcar sintáctico que, simplemente, hace más cómoda la programación. Todo programa escrito mediante esta técnica puede ser transformado en un problema de programación lineal o entera (o mezcla de ambos).
La idea consiste en poder representar fácilmente funciones lineales a intervalos.

Astilleros

Supongamos que ciertos astilleros tienen que hacer frente a una serie de pedidos que conocidos de antemano. Para poder planificar la producción es necesario saber que:

- Cada pedido se corresponde con un periodo de producción en el astillero.
- Para cada periodo la producción de barcos tiene un determinado coste por unidad. Si el pedido supera cierta cantidad de barcos este coste se incrementa para aquellos barcos que se produzcan de más.
- Existe la posibilidad de almacenar parte de la producción de un periodo para ser vendida en otro. Esto conlleva un coste de almacenaje.

Se trata de hacer frente a la demanda con el mínimo coste. Se trataría de un típico caso de programación lineal:

```
int+ nPeriodos = ...;
range Periodos 1..nPeriodos;
float+ demanda[Periodos] = ...;
float+ regularCoste = ...;
float+ extraCoste = ...;
asser regularCoste <= extraCoste;
float+ capacidad = ...;
float+ inventario = ...;
```

```
float+ inventarioCoste = ...;
var float+ regBarco[Periodos];
var float+ extBarco[Periodos];
var float+ inv[o..nPeriodos];
minimize
        regCoste * (sum(t in Periodos) regBarco[t]) +
        extCoste * (sum(t in Periodos) extBarco[t]) +
        inventarioCoste * (sum(t in Periodos) inv[t])
subject to {
        inv[0] = inventario;
        forall(t in Periodos)
                regBarco[t] <= capacidad;
        forall(t in Periodos)
                regBarco[t] + extBarco[t] + inv[t-1] = inv[t] +
demanda[t]
};
```

La posibilidad de usar poder manejar cómodamente funciones lineales a intervalos a través de la herramienta "piecewise" en OPL permitiría escribir:

```
int+ nPeriodos = ...;
range Periodos 1..nPeriodos;
float+ demanda[Periodos] = ...;
float+ regularCoste = ...;
float+ extraCoste = ...;
float+ capacidad = ...;
float+ inventario = ...;
float+ inventarioCoste = ...;
var float+ Barco[Periodos];
var float+ inv[o..nPeriodos];
minimize
        sum(t in Periodos) piecewise{ regCoste -> capacidad ; extCoste
} boat[t] +
        inventarioCoste * (sum(t in Periodos) inv[t])
subject to {
        inv[0] = inventario;
        forall(t in Periodos)
                Barco[t] + inv[t-1] = inv[t] + demanda[t]
};
```

3. PLANIFICACIÓN CON OPL

3.1. INTRODUCCIÓN

Una de las novedades más significativas de OPL es el soporte que proporciona para problemas de planificación. Para ello la implementación de OPL consta de algoritmos eficientes que reducen considerablemente el espacio de búsqueda de la solución.

3.2. CONCEPTOS

Podemos reducir todas las nociones que manejamos en los problemas de planificación a dos conceptos fundamentales: actividad y recurso.

Una actividad representa una tarea, un proceso que tiene una determinada duración de tiempo.

Un recurso es aquello necesario para que se pueda llevar a cabo una actividad.

OPL proporciona a través de estas estructuras la simulación de las características y relaciones de los elementos que encontramos en cualquier problema de planificación.

3.3. ACTIVIDADES EN OPL

Desde un punto de vista formal podemos definir una actividad con tres datos sujetos a una restricción:

- El comienzo de la actividad, su duración y su finalización.
- La finalización tiene que ser igual al valor que representa su comienzo mas el que representa su duración.

Por lo tanto, podemos concebir una actividad como un intervalo de tiempo.

3.4. ORIGEN Y HORIZONTE

Todas las actividades que tengamos definidas en nuestro programa van a estar situadas dentro del intervalo global:
[sheduleOrigin,sheduleHorizon)

Este intervalo nos va a proporcionar los límites a nuestra búsqueda. En principio tiene unos valores por defectos pero es altamente recomendable

redefinirlos en función de las características de nuestro problema. Los elementos que determinan una actividad son enteros y la búsqueda no es demasiado eficiente. Con ello reducimos el espacio de búsqueda.

Para acotar este intervalo podemos establecer instrucciones con la siguiente forma:

```
sheduleOrigin = 0;
sheduleHorizon = 364;
```

3.5. DECLARACIÓN DE ACTIVIDADES EN OPL

La declaración más típica de una actividad en OPL es aquella en la que le especificamos su duración mediante la instrucción:

```
Activity carpinteria(10);
```

Su comienzo y su final son variables que, a lo largo del programa, podrán tomar valores dentro de este rango global consistentes con las restricciones que establezcamos.

También podemos declarar actividades de duración variable mediante la declaración análoga:

```
int  curacionCarpinteria in 8..10;
Activity Carpinteria[duracionCarpinteria);
```

o incluso no especificar nada sobre su duración:

```
Actividad Carpinteria;
```

Lo más habitual será disponer de un conjunto de actividades sujetas a una serie de restricciones y una función a minimizar. La representación de estas actividades la podemos establecer a través de un array de actividades:

```
enum Tarea =...;
int duration[Tarea];
Activity tareas[t in 1..10](duration[t]);
```

donde declaramos 10 actividades con duración: duration[t].

OPL nos proporciona también el concepto de actividad interrumpible, es decir, aquella que, debido a las restricciones del programa, no necesita seguir ejecutándose hasta el final una vez iniciada. Su declaración sería:

```
Activity Carpinteria breakable;
```

O, como en el caso anterior, podemos declarar 10 actividades interrumpibles:

```
Activity Tareas[t in 1..10](duration[t]) breakable;
```

OPL es tan versátil que permite mezclar actividades interrumpibles con otras que no lo son:

Activity Tareas[t in 1..10](duration[t]) breakable if t in breakableSet;

donde breakableSet es un conjunto de enteros.

3.6. RESTINCIONES CON ACTIVIDADES

La restricción por excelencia entre actividades es aquella en la cual establecemos que para comience una debe haber terminado la otra:

a precedes b;

También podemos acceder a los valores de comienzo y fin:

b.start >= a.end;

donde establecemos que para que comience b tiene que haber terminado a. O equivalentemente:

b.start >= a.start + a.duration;

3.7. RECURSOS

La idea de recurso es el otro concepto básico sobre el que se apoya la planificación en OPL. Para describir formalmente sus características OPL implementa varios tipos de recursos.

3.7.1. *Recursos Unarios*

Representan la implementación más simple de la idea de recurso. Es decir, para la ejecución de cierta actividad es necesario disponer de un elemento que no puede ser usado por otra actividad hasta que no finalice la ejecución de la primera.

Mediante la siguiente instrucción:

UnaryResource machine;

Establecemos que disponemos de cierto recurso: machine, que no puede ser compartido por dos actividades. Generalizando, podemos declarar:

UnaryResource machine[1..10];

donde declaramos 10 recursos.

Una vez que sabemos como declarar recursos tenemos que codificar de algún modo las tareas que requieren un determinado recurso:

excavation require machine;

con esta instrucción estamos diciendo que cierta actividad, previamente declarada, necesita para su ejecución ese recurso. Si no dispone de él su comienzo se retrasará hasta que esté disponible.

Ejemplo. Trabajos y máquinas.

Objetivo: minimizar el tiempo de ejecución de un conjunto de trabajos. Descompondremos cada trabajo en una serie de tareas que tendrán que ser efectuadas por cierta máquina.

Restricciones:

- Las tareas de cada trabajo tienen que realizarse en un determinado orden.
- Cada una de las máquinas solamente podrá ser utilizada simultáneamente por una tarea.

Tendremos que manejar información sobre trabajos, tareas y máquinas:

```
int nbMachines = ..;
range Machines 1..nbMachines;
int nbJobs = ...;
range Jobs 1..nbJobs;
int nbTasks = ...;
range Task 1..nbTasks;
```

Además, deberemos especificar qué máquinas son necesarias para cada una de las tareas y cual va a ser su duración:

```
Machines resource[Jobs,Task] = ...;
int+ duration[Jobs,Tasks] = ...;
```

Con las siguiente instrucciones simplemente queremos buscar una solución más eficiente. Lo hacemos restringiendo el dominio de las variables:

```
int totalDuration = sum(j in Jobs, t in Tasks) duration[j,t];
SheduleHorizon = totalDuration;
```

con ellas nos hemos puesto en el caso más extremo: aquel en el cual no hay procesamiento paralelo de ninguna de las tareas. Aun así, podemos acotar el tiempo de toda la planificación.

Pasemos a declarar las actividades:

```
Activity task[i in Jobs, t in Tasks] (duration[j,t]);
Activity makespan(0);
```

tenemos un conjunto de actividades que van a representar cada una de las tareas de nuestros trabajos. Además, disponemos de una actividad de duración cero.

Su comienzo representa el fin la última tarea del conjunto de trabajos. Es muy útil a la hora de establecer las restricciones. Nos queda codificar la condición sobre el uso exclusivo de cada una de las máquinas:

```
UnaryResource tool[Machines];
```

Nos queda aún que el final de la planificación es posterior a cada una de las tareas, que cada trabajo necesita la ejecución de sus tareas en cierto orden y que cada tarea necesita su máquina:

```
forall(j in Jobs)
        task[j,nbTasks] precedes makespan;
forall(j in Jobs)
        forall(t in 1..nbTasks-1)
                task[j,t] precedes task[j,t+1];
forall(j in Jobs)
        forall(t in Tasks)
                task[j,t] requires tool[resource[j,t]];
```

El programa en su conjunto quedaría:

```
int nbMachines = ..;
range Machines 1..nbMachines;
int nbJobs = ...;
range Jobs 1..nbJobs;
int nbTasks = ...;
range Task 1..nbTasks;
Machines resource[Jobs,Task] = ...;
int+ duration[Jobs,Tasks] = ...;
int totalDuration = sum(j in Jobs, t in Tasks) duration[j,t];
SheduleHorizon = totalDuration;
Activity task[i in Jobs, t in Tasks] (duration[j,t]);
Activity makespan(0);
UnaryResource tool[Machines];
minimize
        makespan.end
subject to {
        forall(j in Jobs)
                task[j,nbTasks] precedes makespan;
        forall(j in Jobs)
                forall(t in 1..nbTasks-1)
```

```
                    task[j,t] precedes task[j,t+1];
        forall(j in Jobs)
            forall(t in Tasks)
                    task[j,t] requires tool[resource[j,t]];
};
```

3.7.2. *Recursos discretos*

Nos permiten declarar un tipo de recurso y, a la vez, establecer de cuántas copias disponemos. Su declaración es análoga a la anterior:

```
DiscreteResource machine(5);
```

donde hemos establecido que disponemos de 5 copias del tipo de recurso: machine. Si lo que queremos es codificar varios tipos de máquinas de las cuales, a su vez, disponemos de un determinado número de copias de cada una, podemos crear un array de recursos discretos:

```
DiscreteResource TipoMaquina[t in 1..10](cap[t]);
```

con esta declaración decimos que de la máquina de tipo t disponemos de cap[t] copias.

Podemos generalizar la instrucción donde codificábamos que una tarea necesita de un recurso al caso de los recursos discretos. Ahora vamos declarar que cierta actividad: a, 2 máquinas de tipo 1:

```
a requires(2) TipoMaquina[1];
```

Hasta ahora hemos visto cómo las actividades pueden monopolizar recursos. Sin embargo, en nuestros problemas de planificación también se da el caso de que nuestras tareas consumen recursos. Con la instrucción:

```
a consume(2) r;
```

indicamos que la actividad consume dos copias del recurso discreto r. Esta idea de implementar el consumo de recursos se podría extender al caso de los recursos unarios aunque su utilidad es mínima.

Ejemplo. Construcción de una casa (1ª parte).

Suponemos que la construcción de una casa la descomponemos en un conjunto de tareas. Cada una de estas tareas debe efectuada por un trabajador. Cada trabajador solamente puede realizar una tarea a la vez. El objetivo consiste en minimizar el tiempo de construcción de la casa.

La declaración de datos sería algo así:

```
enum Tasks = {masonry,carpentry,plumbing,ceiling,roofing,painting,
windows,facade,garden,moving};
int duration[Tasks] = ...;
enum Workers { Thomas, Maite, Antoine };
DiscreteResource pool(3);
Activity a[t in Tasks](duration[t]);
UnaryResource worker[Workers];
var int use[Task,Workers] in 0..1;
```

donde representamos las actividades y los recursos de que disponemos. var int use[Task,Workers] in 0..1 significa que ese trabajador ha realizado esa tarea. La función a minimizar es:

```
minimize
        a[moving].end
```

donde moving es la última tarea. Más adelante codificaremos las relaciones de precedencia de las tareas.

Para que comience una tarea tiene que haber un trabajador desocupado:

```
forall(t in Tasks, w in Workers)
        a[t] requires(use[t,w]) worker[w];
```

Cada tarea tiene que ser efectuada por un solo trabajador:

```
forall(t in Tasks)
        sum(w in Workers) use[t,w] = 1;
```

El programa completo sería:

```
enum Tasks = {masonry,carpentry,plumbing,ceiling,roofing,painting,
windows,facade,garden,moving};
int duration[Tasks] = ...;
enum Workers { Thomas, Maite, Antoine };
DiscreteResource pool(3);
Activity a[t in Tasks](duration[t]);
UnaryResource worker[Workers];
var int use[Task,Workers] in 0..1;
minimize
        a[moving].end
subject to{
        task[[masonry] precedes task[carpentry];
```

```
        task[painting] precedes task[moving];
        forall(t in Tasks)
                a[t] requires(1) pool;

        forall(t in Tasks, w in Workers)
                a[t] requires(use[t,w]) worker[w];
        forall(t in Tasks)
                sum(w in Workers) use[t,w] = 1;
};
```

3.7.3. *Reservoir (recursos renovables)*

De manera análoga a como implementamos los recursos consumibles podemos establecer que disponemos de un tipo de recurso renovable. Con:

```
Reservoir plumbing(3);
```

declaramos el tipo de recurso plumbing del cuan disponemos de 0 copias inicialmente y, como máximo, podemos disponer de 3 copias. Si al comienzo de la planificación disponemos de alguna copia del recurso, 2 por ejemplo, podemos escribir:

```
Reservoir plumbing(3,2);
```

La instrucción que nos permite incrementar el número de copias de un recurso es análoga a la del consumo:

```
b produces(2) plumbing;
```

3.7.4. *Recursos con estado*

En muchas ocasiones nuestros problemas de planificación nos llevan a codificar que una determinada tarea para poder ser efectuada debe disponer de un recurso en un estado concreto. OPL proporciona un soporte elegante para poder tratar estas situaciones.

Si queremos declarar recursos con estado tenemos que declarar el tipo de estados posibles previamente el conjunto de estados posibles:

```
enum estados { hot, warm, cold };
StateResource horno(estados);
```

donde hemos especificado que el recurso horno tiene 3 estados posibles: hot,

warm, cold. Para exigir a una actividad que el recurso horno esté en estado warm escribiremos:

```
a requiresState(warm) horno;
```

OPL también nos permite trabajar con conjuntos de estados:

```
{estados} notCold = { hot, warm};
a requireAnyState(notCold) horno;
```

Con este par de instrucciones establecemos que la actividad a requiere al recurso horno en estado cold. También podemos la relación de exclusión de una actividad de un estado de un recurso:

```
a excludeState(cold) oven;
a excludeAllStates(notCold) oven;
```

donde excluimos un estado y un conjunto de estados, respectivamente.

3.7.5. *Recursos alternativos*

Existe una implementación de la noción de conjunto de recursos unarios. Mediante las siguientes declaraciones establecemos el conjunto de recursos worker:

```
UnaryResources worker[Worker];
AlternativeResources s(worker);
```

Para exigir que cierta actividad use algún recurso del conjunto escribiremos:

```
a requires s;
```

Existe, además, una restricción especialmente útil si una actividad a ha seleccionado el recurso u del conjunto de recursos unarios s:

```
activityHasSelectedResource(a,s,u)
```

Ejemplo. Construcción de una casa (2ª versión).

Queremos reelaborar el programa de construcción de una casa cambiando algunos elementos:

- No todos los trabajadores pueden realizar todas las tareas.
- Deseamos minimizar el tiempo de trabajo de cada uno de los empleados.

Ahora vamos a ver a los trabajadores desde dos puntos de vista: como recursos y como sujetos que van a realizar una tarea. La declaración de datos con las nuevas restricciones sería:

```
enum Tasks = {masonry,carpentry,plumbing,ceiling,roofing,painting,
windows,facade,garden,moving};
```

```
int duration[Tasks] = ...;
int totalDuration = sum(i in Tasks) duration[t];
scheduleHorizon = totalDuration;
Activity task[t in Tasks](duration[t]);
enum Workers {joe, jack, jim};
{Workers} cannotperform[Tasks] = ...;
UnaryResource worker[Workers];
AlternativeResource s(worker);
var int durationWorkers[Workers] in 0..totalDuration;
Activity attendance[w in Workers](durationWorkers[w]);
```

el cambio se traduce en que ahora la variable de la búsqueda va a ser la duración del trabajo de cada empleado. Con las nuevas restricciones el objetivo es:

```
minimize
        max(w in Workers) durationWorkers[w]
```

La restricción que exige que los empleados solo puedan hacer las tareas para las cuales están preparados es:

```
forall(t in Tasks)
        forall(w in cannotperform[t])
                not activityHasSelecResource(task[t],s,worker[w]);
```

Además, el periodo de ejecución de una tarea por un empleado debe estar comprendida en el intervalo de tiempo en el que éste está trabajando:

```
forall(t in Tasks)
        forall(w in Workers)
                activityHasSelectedResource(tast[t],s,worker[w]) =>
                        attendance[w].start <= task[t].start &
                        attendance[w].end. >= task[t].end;
```

Programa completo:

```
enum Tasks = {masonry,carpentry,plumbing,ceiling,roofing,painting,
windows,facade,garden,moving};
int duration[Tasks] = ...;
int totalDuration = sum(i in Tasks) duration[t];
scheduleHorizon = totalDuration;
Activity task[t in Tasks](duration[t]);
enum Workers {joe, jack, jim};
{Workers} cannotperform[Tasks] = ...;
```

```
UnaryResource worker[Workers];
AlternativeResource s(worker);
var int durationWorkers[Workers] in 0..totalDuration;
Activity attendance[w in Workers](durationWorkers[w]);
minimize
        max(w in Workers) durationWorkers[w]
subject to{
        task[[masonry] precedes task[carpentry];
                .
                .
                .
        task[painting] precedes task[moving];
        forall(i in Tasks) task[t] requires s;
        forall(t in Tasks)
        forall(w in cannotperform[t])
                not activityHasSelecResource(task[t],s,worker[w]);
forall(t in Tasks)
        forall(w in Workers)
                activityHasSelectedResource(tast[t],s,worker[w]) =>
                        attendance[w].start <= task[t].start &
                        attendance[w].end. >= task[t].end;
};
```